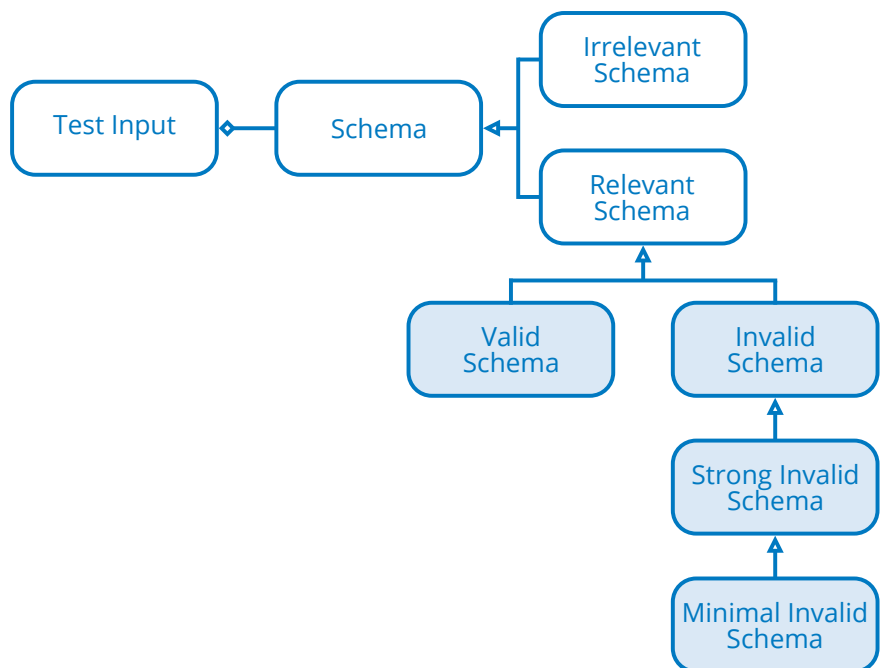Konrad Anton Fögen

Combinatorial Robustness Testing based on Error-Constraints

Combinatorial Robustness Testing based on Error-Constraints

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Konrad Anton Fögen, M. Sc.
aus Höxter

Berichter: Universitätsprofessor Dr. rer. nat. Horst Lichter
              Professor Angelo Gargantini, Ph.D.

Tag der mündlichen Prüfung: 10. Februar 2021

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek verfügbar.

Konrad Anton Fögen
RWTH Aachen University

Combinatorial Robustness Testing
based on Error-Constraints

Printed in Germany.

Kurzfassung

Robustheit ist eine wichtige Eigenschaft einer Software, welche zusätzlich zur Funktionalität getestet werden muss. Dies erfordert ungültige Werte und ungültige Wertekombinationen, um die Reaktion einer Software auf die Eingabe dieser Werte beobachten zu können.

Kombinatorisches Testen (CT) ist eine effektive spezifikationsbasierte Testmethode, die auf einem Eingabeparametermodell (IPM) basiert, mit dem Eingaben für das Testen ausgewählt werden. Die Effektivität von CT verschlechtert sich jedoch, wenn ungültige Werte oder ungültige Wertekombinationen vorhanden sind. Dieses Phänomen wird als Maskierungseffekt ungültiger Eingaben bezeichnet und ist in der Forschung bereits bekannt. Das Phänomen führte zu Erweiterungen der CT Testmethode, welche wir als Kombinatorisches Testen für Robustheit (CRT) bezeichnen. Das Ziel von CRT ist es, das Erkennen von Fehlern zu verbessern, indem der Maskierungseffekt ungültiger Eingaben vermieden wird. Vermieden wird er durch die Trennung von Tests mit gültigen Werten und gültigen Wertekombinationen und Tests mit ungültigen Werten und ungültigen Wertekombinationen.

Obwohl CRT eine vielversprechende Erweiterung von CT ist, argumentieren wir, dass sie immer noch unzureichend erforscht ist. Zum Beispiel wird in verwandten Arbeiten das IPM mit semantischen Informationen erweitert, um Werte als ungültig markieren zu können. Wertekombinationen können allerdings nicht direkt als ungültig markiert werden.

In dieser Arbeit entwickeln wir daher die Idee des CRT weiter. Das Ziel ist eine neue CRT Testmethode mit einem Modellierungsansatz, um ungültige Werte und ungültige Wertkombinationen gleichermaßen gut zu spezifizieren. Dieser Modellierungsansatz soll auch in explizite Testabdeckungskriterien und Testauswahlstrategien integriert werden. Zudem soll dieser Modellierungsansatz durch automatisierte Techniken weiter unterstützt werden.

Zunächst führen wir ein kontrolliertes Experiment durch, um zu überprüfen, ob CRT als Erweiterung notwendig ist, oder ob CT bereits geeignet ist, um Robustheit von Software zu testen. Basierend auf den Ergebnissen verfeinern wir das t-Faktor Fehlermodell, sodass Robustheitsfehler und der inhärente Maskierungseffekt ungültiger Eingaben berücksichtigt werden.

Dann entwickeln wir eine neue Testmethode für CRT und führen das Robustheits-Eingabeparametermodell (RIPM) ein, welches die Struktur der IPMs um das Konzept der Error-Constraints erweitert. Dabei handelt es sich um einen zusätzlichen Satz logischer Ausdrücke zur Beschreibung der Gültigkeit von Werten und Wertekombinationen.

Mit dem verfeinerten t-Faktor Fehlermodell und der neuen RIPM Struktur werden neue Testabdeckungskriterien, welche die zusätzlichen semantischen Informationen einbeziehen, und neue Testauswahlstrategien, welche die Testabdeckungskriterien erfüllen, entwickelt.

Das neue Konzept der Error-Constraints erfordert zusätzlichen Aufwand. Daher entwickeln wir zwei zusätzliche Techniken, welche die Modellierung der Error-Constraints unterstützen. Zuerst entwickeln wir eine Technik zur Identifizierung und Reparatur von Inkonsistenz innerhalb der Error-Constraints. Zudem entwickeln wir eine Technik zur automatischen Generierung von Error-Constraints, welche auf der Konformität mit einem anderen System basiert.

Zu guter Letzt werden alle oben genannten Konzepte und Techniken operationalisiert und in ein Framework zu Testautomatisierung integriert, welches einen Prozess, eine Architektur und eine Java-basierte Referenzimplementierung umfasst.

Abstract

Robustness is an important property of a software, which must be tested in addition to a software's functionality. This requires invalid values and invalid value combinations to be able to observe a software's reaction to them.

Combinatorial testing (CT) is an effective specification-based test method that is based on an input parameter model (IPM) with which test inputs are selected. But its effectiveness deteriorates in the presence of invalid values or invalid value combinations. This phenomenon is called invalid input masking effect and is already acknowledged in some research. The phenomenon led to extensions of CT that we call combinatorial robustness testing (CRT). The objective of CRT is to improve the fault detection by avoiding invalid input masking. This is achieved by separating the testing of valid values and valid value combinations from the testing of invalid values and invalid value combinations.

While CRT is a promising extension of CT, it is still insufficiently researched. For instance, in related work, IPMs are extended with additional semantic information to specify invalid values. However, invalid value combinations cannot be specified directly.

Therefore, the objective of this work is to further expand the idea of CRT. The aim is to develop a new CRT test method with a modeling approach to specify invalid values and invalid value combinations equally well. This modeling approach should also be incorporated into explicit test adequate criteria and test selection strategies. Furthermore, this modeling approach shall be supported by automated techniques.

First, we conduct a controlled experiment to check if CRT is necessary at all or if CT is already appropriate to test robustness. Based on the result, we continue and develop a refined t-factor fault model that incorporates robustness faults and the inherent invalid input masking effect.

Next, we develop a new test method for CRT and introduce a new structure that extends the structure of IPMs. It is called robustness input parameter model (RIPM) and contains the concept of error-constraints which is an additional set of logical expressions to describe the validity of values and value combinations.

With the refined t-factor fault model and the new RIPM structure, new test adequacy criteria that incorporate the additional semantic information and new test selection strategies that satisfy the test adequacy criteria are developed.

The new concept of error-constraints requires additional effort in modeling. Therefore, we develop two techniques to support the modeling of them. First, we develop a technique to identify and repair inconsistencies among error-constraints. Second, we develop a technique to automatically generate error-constraints based on the conformance to another system.

Last but not least, all aforementioned concepts and techniques are operationalized and integrated in a test automation framework which includes a process, an architecture, and a Java-based reference implementation.

Contents

Part I.

Foundations

Chapter 1.

Introduction

Contents

1.1. Research Motivation

Today, software is omnipresent and increasingly important for society and business. The whole world highly depends on software. As SOMMERVILLE [Som16, p.18] points out, "national infrastructures and utilities are controlled by computer-based systems, and most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerized, as is the financial system". Nowadays, software and its programs are increasingly integrated and connected. We as people, organizations, and society become more and more dependent on the behavior of programs. In fact, the daily life of almost anyone is affected by the dependability of programs [AO16, p.1].

An important characteristic of programs is *robustness* which can be defined as "the degree to which [programs] can function correctly in the presence of invalid inputs or stressful environmental conditions" [IEE90b]. Whether an input is valid or invalid is context-dependent: Valid inputs lie "within the bounds of normal operation" of a program whereas invalid inputs "lie outside of the normal operating range" [GOA05]; [AO16]. Examples of syntax-level invalid inputs are omissions of values, values containing invalid characters, or values provided in an invalid format. On the semantic-level, inputs can also be logically invalid. For instance, a number that exceeds a limit or begin and end dates of a period that are flipped. Stressful environmental conditions are also context-dependent. Other programs might be unavailable or do not respond within an expected time frame. Invalid inputs and stressful environmental conditions can be summarized as *external faults* since their origin is external to the program's boundary [Avi+04].

External faults can have a severe impact on the program's robustness because they can infect the program's state and propagate to *robustness failures* [Avi+04]. Robustness failures can lead to unavailable programs because the program or even the entire operating system crashes [Koo+97]. Robustness failures can also lead to security breaches when an attacker uses invalid

inputs to gain privileges by exploiting a buffer overflow [BE11]. Furthermore, robustness failures can cause a loss of revenue when business rules or legal requirements are violated. For instance, allowing a customer to reserve a car for the 29th of February in a non-leap year or accepting customer bookings with flipped begin and end dates results in downstream business problems when asked to fulfill the service or when asked to create an invoice.

To prevent robustness failures, a program must become tolerant towards external faults such that external faults are detected and corresponding errors are eliminated from the program's state before they can propagate to robustness failures [Avi+04]. To realize fault tolerance, programs can be developed defensively [BF04]. *Defensive programming* is a principle that is based on mistrusting all influences that are external to the program's boundary [Mey92]; [BE11]; [MSB11]. External influences that are invalid or even dangerous must be neutralized or rejected [TBL17]. Therefore, external faults must be anticipated and dedicated computer instructions must be executed to detect errors in the program's state that are caused by external faults [BF04]; [Avi+04]. Once an error is detected, further computer instructions must be executed to recover the state by eliminating the error. If computer instructions for the detection or recovery are incorrect, a *robustness fault* is manifested which is an internal fault that enables an external fault to cause a robustness failure.

To develop defensively and to support the detection and recovery of errors caused by external faults, *exceptions* allow representing errors in the program's state and exception handling (EH) is used to detect errors in the state and to recover from them [DG94]; [MT97]; [LS98]; [Gar+01]. Often, a significant proportion of a system under test (SUT)'s source code is devoted to EH. According to CRISTIAN [Cri89], up to two thirds of a SUT's source code is devoted to EH. A more recent study conducted by SINHA & HARROLD [SH00] examined Java programs and identified that between 8.9% and 66.7% of all examined classes detect and propagate exceptions. WEIMER & NECULA [WN08] analyzed open-source programs also written in Java with 4,000 to 16,000 lines of source code and between 1% to 5% of source code was directly devoted to EH. Furthermore, between 3% and 46% of source code was indirectly linked to EH, e.g. a method to clean up resources was invoked as a response to an exception. These numbers further show that EH is frequently used and an important part of programs.

Even though defensive programming in general and EH in particular are intended to support the development of robust programs, they are themselves significant sources of robustness faults and robustness failures. For instance, MARINESCU [Mar11] inspected Java classes to find correlations between the exceptions and faults. The inspection shows that classes with EH are more likely to contain faults. SAWADPONG et al. [SAW12] investigated six releases of the Eclipse[1] integrated development environment (IDE) and compared the exception-related fault density to the overall fault density. The study shows that source code devoted to EH has a fault density that is approximately three times higher than the overall fault density of source code that is also devoted to the standard functions.

The argumentation of MILLER & TRIPATHI [MT97] further shows that the goals of dedicated language features devoted to EH often conflict with the goals of other language features in modern object-oriented programming languages which complicates EH even more. In addition, EH is not limited to isolated modules but is known to be a global issue that affects all modules

---

[1]See https://www.eclipse.org (Last access on 1st December 2020).

[Fil+09]. Due to the dedicated language feature devoted to EH, an *implicit* control flow for exceptions is established which is detached from the standard control flow and complicates the EH even more [RM00]; [Cac+08]; [WN08]. EH is considered to be one of the least understood parts of programming languages and source codes. Developers often focus on the standard behavior and EH is often oversimplified or neglected (cf. [SGH10]; [BGB14]; [BG18]).

A famous example is the case of the Ariane 5 maiden flight which occurred in 1996 where exceptions caused a robustness failure and a loss of 1.9 billion French francs [Lan97]; [Ben01]. The identified cause was a function within the inertial navigation system which is supposed to compute the position of the rocket. The function attempted to compute a conversion from a 64-bit floating point to a 16-bit signed integer without checking if 16 bits were sufficient [Lan97]. The conversion failed and an exception was propagated to the central system. While this function worked for the previous model Ariane 4, the trajectory of Ariane 5 was much higher than the trajectory of its predecessor. From the perspective of the function, the trajectory as input was beyond the standard input domain and the function initiated exception behavior and propagated an exception. From the perspective of the entire system, the function did not satisfy the standard specification of the Ariane 5 model. Due to the propagated exception, a report of the exception rather than an actual position of the rocket was sent to the central system [Ben01]. The central system did not validate its input and treated the report as valid but extreme input which resulted in a turn that was too sharp and the rocket started to break.

To conclude, it is not only important to check that a program realizes its standard functions as intended. It is also important to check the behavior of a program in exceptional situations.

1.2. Research Context

Testing is the primary approach in industry to evaluate and ensure quality of programs [AO16]. It is a dynamic activity where the SUT is stimulated with selected test inputs and the behavior of the SUT is observed and compared to the expected behavior [Bar+15]; [LO17].

Combinatorial testing (CT) is a specification-based test method. The selection of test inputs is based on an input parameter model (IPM) [GOA05]; [NL11a]. IPMs are models of the SUT's specification that structure the input domain of a SUT via parameters and domains of

A test adequacy criterion defines a predicate that must be satisfied by a test suite [WO80]; [WJ91] and a test selection strategy describes how to select test inputs such that the test adequacy criterion is satisfied [Gou83]; [ZHM97]; [Hie02]. In CT, test adequacy criteria and test selection strategies are defined relative to an IPM. A structure of IPMs together with a corresponding fault model, test adequacy criteria, and test selection strategies denote a CT test method.

The general idea of CT is based on combinatorics which allows testing with relatively small test suites as opposed to exhaustive testing with large unfeasible test suites [KKL13]. CT is relying on the t-factor fault model which assumes that a fault is caused by the interaction of t parameters [DM98]. The t-wise coverage test adequacy criterion is only satisfied if each t-sized combination of values of an IPM appears in at least one test input [KKL13]. Consequently, a test suite that satisfies t_1-coverage contains at least one t_1-sized combination of values that activates a t_2-factor fault as long as $t_1 \geq t_2$. Typically, t takes on a value of one to six which is much lower than the number of all parameters [KKL13].

Various studies demonstrate that CT is an effective test method [Gri+06]. But the effectiveness depends on the quality of the IPM that must model all important parameters and values [Bor+13]; [GO07]; [Kuh+20]. Combinations of values that are, for instance, irrelevant and uninteresting to test or not executable, must also be identified and excluded from the test suite [Gri+06]; [GOM07]; [Pet15] [Wu+19b]; [Wu+19a]. Otherwise, those *irrelevant value combinations* can prevent test inputs from being tested [Yil+14a]. Values and value combinations that only appear in a test input that also covers irrelevant value combinations remain untested and faults may remain undetected. They are *masked* by some irrelevant value combinations. This phenomenon is denoted as the *input masking effect* [Yil+14a]; [Wu+19b]. Therefore, the irrelevant value combinations must be identified and excluded from the final test suite. Early approaches that deal with irrelevant value combinations are based on manual remodeling of IPMs such that the modified IPMs do not model any irrelevant value combinations [Wu+19a]. The final test suite is the result of several iterations of remodeling and test selection. However, this approach is not competitive because of the larger test suite size caused by multiple iterations and the manual effort to remodel IPMs. Nowadays, irrelevant value combinations are most commonly described with additional semantic information in IPMs and test selection strategies are able to automatically exclude them from the final test suite by avoiding their selection [Wu+19a].

When testing exceptional situations, test inputs with an invalid value or an invalid value combination are required to consciously trigger EH. Due to the nature of EH, an alternative control flow path for recovery is taken through the SUT once the occurrence of an exception is detected. Then, other values and value combinations of the same test input do not exercise the SUT as expected. These values and value combinations may remain untested and faults may remain undetected when they only appear in a test input that triggers EH. This is another instance of the input masking effect – the invalid input masking effect – which requires a different solution. Avoidance of invalid values and invalid value combinations is not an option because it is necessary to check the behavior in exceptional situations.

This difficulty is already acknowledged in some research (cf. [She94]; [Coh+94]; [Coh+97]; [GOA05]; [Cze06]). It led to extensions of CT that we call combinatorial robustness testing (CRT): CRT denotes extensions of CT where the IPM is enriched with additional semantic information and test adequacy criteria as well as test selection strategies are extended to incorporate the additional semantic information. The main idea of CRT is to improve the fault detection by avoiding the invalid input masking effect. Therefore, valid and invalid test inputs are separated such that all values and value combinations are tested as expected. Only valid values as well as valid value combinations are used in valid test inputs to test the standard behavior and invalid values as well as invalid value combinations are only used in invalid test inputs to test the exceptional behavior.

1.3. Research Objective

Even though some test methods for CRT exist, CRT is still insufficiently researched[2]. First, it is not clear if CRT test methods are necessary or if CT test methods are already appropriate

---

[2]Please refer to Chapter 4 for an in-depth discussion of related work.

because only qualitative argumentation and examples but no quantitative assessment exists yet.

Second, CT relies on a generic fault model – the t-factor fault model – from which test adequacy criteria like t-wise coverage are derived. The existing CRT test methods have no explicated fault model that describes the faults they are targeting and the test adequacy criteria of different CRT test methods target different faults.

Third, the test selection strategies of CRT test methods are insufficiently automated. Early approaches again rely on manual efforts to remodel the IPMs. Although remodeling is a universal approach that allows influencing test selection in many different ways, it results in larger test suites and requires additional manual effort. Current approaches rely on semantic information that is used to annotate invalid values of IPMs. Then, test selection strategies exclude the invalid values when selecting valid test inputs and select some invalid test inputs that contain exactly one invalid value for each annotated invalid value. The current approaches do not directly support invalid value combinations. Although, workarounds exist to integrate invalid value combinations. The workarounds allow modeling invalid value combinations such that they do not appear in valid test inputs and such that invalid test inputs that cover them are selected.

However as we will point out in Chapter 4, the workaround increases the complexity of IPMs and makes mistakes in IPMs more likely. Further on, the test selection strategies treat invalid value combinations similar as invalid values. Even though invalid value combinations consist of values for two or more parameters, no combinatorics between them can be utilized because the test adequacy criteria and test selection strategies are unaware of invalid value combinations. In that sense, the CRT test methods are incomplete.

Finally, CRT test methods require more semantic information than CT test methods since invalid values and potentially invalid value combinations must be annotated. Therefore, supportive techniques to check the consistency of existing annotations and supportive techniques to support the activity of annotating values and value combinations are desirable. Until now, no supportive techniques for CRT test methods exist. Due to the lack of direct support for invalid value combinations, the development of supportive techniques for existing CRT test methods is further complicated.

Therefore, the overall goal of this work is to develop a CRT test method that consists of (1) a refined t-factor fault model that incorporates robustness faults and input masking, (2) an extended structure of IPMs that allows annotating values and value combinations as invalid, (3) a set of test adequacy criteria that takes the refined fault model into account, and (4) a set of test selection strategies that describes how to select small test suites such that the test adequacy criteria are satisfied. Furthermore, the goal is to provide supportive techniques (5) to automatically check IPMs for consistency and to repair inconsistencies among the annotations of invalid values and invalid value combinations and (6) to automatically infer and annotate values and value combinations as invalid.

The first research objective of this work is to assess the necessity of CRT test methods. Since the annotation of invalid values and invalid value combinations means additional effort and additional sources of mistakes, shortcomings of CT test methods must be recognizable.

Research Question 1 How effective is CT and what are its shortcomings in detecting faults when invalid input masking is present?

To answer this research question, we conduct a controlled experiment in which we inject faults in test scenarios with input masking and evaluate the fault detection effectiveness of test suites selected by a CT test method.

The controlled experiment indicates shortcomings and possible improvements arise.

As a first step towards an improved CT test method that mitigates the identified shortcomings, a refined fault model that incorporates the input masking effect is necessary. Therefore, the second research objective of this work is to refine the t-factor fault model accordingly.

Research Question 2 How can the t-factor fault model be refined such that it explicitly describes robustness faults and the inherent invalid input masking effect?

To answer this research question, we first identify issues related to EH and derive a classification of robustness fault characteristics. Afterwards, the t-factor fault model is refined to integrate the findings of our classification.

Given the refined t-factor fault model and its additional information, a specific CRT test method can be designed. Therefore, the third research objective of this work is to develop a CRT test method.

Research Question 3 How can a CRT test method capture and utilize the additional information for annotating invalid values and invalid value combinations to select test suites that activate faults in the presence of invalid input masking?

This research question is based on the refined t-factor fault model and affects the structure of IPMs, test adequacy criteria and test selection strategies. Therefore, the research question can be further broken down.

Research Question 3.1 How can the structure of IPMs be extended such that IPMs capture semantic information that annotate invalid values and invalid value combinations?

Research Question 3.2 How can test adequacy criteria be defined such that their satisfaction ensures the activation of faults as described by the refined t-factor fault model?

Research Question 3.3 How can test selection strategies be defined such that they select small test suites that satisfy the test adequacy criteria?

To answer these three research questions, a constructive approach is taken where we introduce an extended IPM structure that captures additional semantic information, formally define the t-wise coverage test adequacy criterion and systematically extend it to match the refined t-factor fault model, and extend a well-researched test selection strategy. To answer research question 3, the constructed CRT test method is compared to a CT test method.

Testing with the newly designed CRT test method is possible in general. However as discussed before, the annotation of invalid values and invalid value combinations requires additional effort and introduces additional sources of mistakes. To mitigate these disadvantages of the CRT test methods, we propose supporting techniques to check the consistency of existing annotations and to identify and repair inconsistent annotations if present.

Research Question 4 How can inconsistent annotations of invalid values and invalid value combinations be detected, identified, and resolved?

To answer this research question, we utilize ideas from the research on constraint satisfaction problems (CSPs) to detect the existence of inconsistencies, to identify annotations that cause inconsistencies, and to provide strategies to resolve the inconsistencies.

Further on, we propose supporting techniques to reduce the extra effort that is required by CRT test methods, i.e. to initially annotate values and value combinations as invalid.

Research Question 5 How can values and value combinations that trigger EH of a SUT be identified during testing and annotated as invalid?

To answer this research question, we utilize ideas of fault characterization (FC) to search for test inputs that trigger EH of a SUT when testing them and to identify the values or value combinations that are responsible for the trigger.

The answers to the previous research questions are theoretical concepts and algorithms that describe how to select test inputs, how to analyze models, and how to search for invalid test inputs that induce EH. The answers are abstract and not directly applicable to practical tasks. To operationalize the concepts and algorithms, a test automation framework that integrates the concepts and algorithms and that allows an implementation to realize the algorithms is desirable.

Therefore, we design a software architecture and develop a prototype `coffee4j` that realizes the test automation framework including all the concepts and algorithms.

1.4. Structure of this Work

This work is structured as follows. In Part I, a running example is introduced that is used throughout this work. Afterwards in Chapter 3, the basic taxonomy of systems, programs, and dependability as well as the fundamental concepts of CT are presented. In addition, the basic taxonomy is extended to precisely describe correctness and robustness. Moreover, we transfer the concepts to the testing domain to describe the issues related to CT and the extensions of the CRT test method. Afterwards, existing CRT test methods and other related works are introduced and contrasted with our work (Chapter 4).

In Part II, an assessment to evaluate the effectiveness of CT when testing for robustness faults is conducted and a combinatorial robustness fault model is developed in Chapter 5. The structure of an extended IPM is defined in Chapter 6 and test adequacy criteria that are based on the fault model and the extended IPM are presented in Chapter 7. Afterwards, test selection strategies are introduced in Chapter 8.

Part III presents techniques that support the use of the CRT test method. First, a technique to check and repair the consistency of existing annotations is presented in Chapter 9. Afterwards, Chapter 10 introduces a technique to initially annotate values and value combinations as invalid. Finally, Chapter 11 covers a test automation framework that operationalizes all previously discussed concepts.

In Part IV, the evaluation of all previously discussed techniques is reported. First, an overview of the evaluations is given in Chapter 12. In Chapter 13, the test selection strategies are assessed.

Then, the supportive techniques are evaluated in Chapter 14 and in Chapter 15. Afterwards, we conclude this work in Chapter 16 with a summary and an outlook.

Running Example   Throughout the dissertation text, we use boxes to highlight when we reference and apply the running example for further illustration.

List of Symbols   To provide an overview of mathematical symbols, we use boxes to summarize newly introduced symbols at the end of sections.

1.5. List of Publications

Parts of this work have already been published. In the following, these publications and my contributions are listed.

[FL18b] Konrad Fögen and Horst Lichter. "Combinatorial Testing with Constraints for Negative Test Cases". In: *Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*. Apr. 2018, pp. 328–331

Level of contribution: High – main investigator, led the work and paper writing. The content is mainly incorporated into Chapters 6, 8, and 9.

[FL18a] Konrad Fögen and Horst Lichter. "A Case Study on Robustness Fault Characteristics for Combinatorial Testing - Results and Challenges". In: *Proceedings of the 6th International Workshop on Quantitative Approaches to Software Quality co-located with 25th Asia-Pacific Software Engineering Conference (APSEC 2018), Nara, Japan, December 4, 2018*. Dec. 2018, pp. 22–29

Level of contribution: High – main investigator, led the work and paper writing. The content is mainly incorporated into Chapter 5.

[BFL19] Joshua Bonn, Konrad Fögen, and Horst Lichter. "A Framework for Automated Combinatorial Test Generation, Execution, and Fault Characterization". In: *Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*. Apr. 2019, pp. 224–233

Level of contribution: Medium – investigator, supervisor of a Bachelor thesis, led the paper writing.
The content is mainly incorporated into Chapter 11.

[FL19b] Konrad Fögen and Horst Lichter. "Combinatorial Robustness Testing with Negative Test Cases". In: *Proceedings of the 19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019*. July 2019, pp. 34–45

Level of contribution: High – main investigator, led the work and paper writing. The content is mainly incorporated into Chapters 6, 7, 8, and 13.

[FL19c] Konrad Fögen and Horst Lichter. "Repairing Over-Constrained Models for Combinatorial Robustness Testing". In: *Proceedings of the 19th IEEE International Conference on Software Quality, Reliability and Security Companion, QRS Companion 2019, Sofia, Bulgaria, July 22-26, 2019*. July 2019, pp. 177–184

Level of contribution: High – main investigator, led the work and paper writing. The content is mainly incorporated into Chapters 9 and 14.

[FL19d]   Konrad Fögen and Horst Lichter. "Semi-Automatic Repair of Over-Constrained Models for Combinatorial Robustness Testing". In: *Proceedings of the 26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019*. Dec. 2019, pp. 110–117

            Level of contribution: High – main investigator, led the work and paper writing. The content is mainly incorporated into Chapters 9 and 14.

[FL19a]   Konrad Fögen and Horst Lichter. "An Experiment to Compare Combinatorial Testing in the Presence of Invalid Values". In: *Proceedings of the 7th International Workshop on Quantitative Approaches to Software Quality co-located with 26th Asia-Pacific Software Engineering Conference (APSEC 2019), Putrajaya, Malaysia, December 2, 2019*. Dec. 2019, pp. 27–36

            Level of contribution: High – main investigator, led the work and paper writing. The content is mainly incorporated into Chapter 5.

[FL20b]   Konrad Fögen and Horst Lichter. "Generation of Invalid Test Inputs from Over-Constrained Test Models for Combinatorial Robustness Testing". In: *Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2020, Porto, Portugal, October 24-28, 2020*. IEEE, Oct. 2020, pp. 171–180

            Level of contribution: High – main investigator, led the work and paper writing. The content is mainly incorporated into Chapters 9 and 14.

[FFL20]   Torben Friedrichs, Konrad Fögen, and Horst Lichter. "A Comparison Infrastructure for Fault Characterization Algorithms". In: *Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2020, Porto, Portugal, October 24-28, 2020*. IEEE, Oct. 2020, pp. 201–210

            Level of contribution: Medium – investigator, supervisor of a Master thesis, led the paper writing.
            The content is mainly incorporated into Chapter 15.

[FL20a]   Konrad Fögen and Horst Lichter. "An Industrial Case Study on Fault Detection Effectiveness of Combinatorial Robustness Testing". In: *Proceedings of the 8th International Workshop on Quantitative Approaches to Software Quality co-located with 27th Asia-Pacific Software Engineering Conference (APSEC 2020), Singapore, December 1, 2020*. Dec. 2020, pp. 29–36

            Level of contribution: High – main investigator, led the work and paper writing. The content is mainly incorporated into Chapter 13.

Chapter 2.

Running Example: Ordering Web Service

Throughout the dissertation text, we will be using an ordering web service as a running example. Therefore, let's suppose a newly established coffee roaster that is located in London, UK plans to sell coffee beans worldwide via an e-commerce system. The most crucial part of the e-commerce system is the ordering web service which is, therefore, our SUT.

To place an order, customers must select the number of coffee packages. They must also provide an address for billing and shipping as well as a shipping method which is either *Standard* for delivery via a local parcel services or *Express* for fast and direct delivery via a globally operating parcel service. In particular, the SUT asks for the following input fields: packages, address, phone, shipping.

The input to the ordering web service has to be validated to ensure entered values actually match the intended semantics of the input fields and to avoid invalid orders and corrupt databases. Only valid inputs are further processed. Invalid inputs are answered with an error message asking the user to correct the inputs. Therefore, the following business rules should be validated by the web service:

- All input fields are non-optional and must not be empty.

- For the `Packages` input field, only odd numbers of packages should be accepted because an even number of packages cannot be wrapped into one box.

- For the `Country` input field, only valid country codes such as UK or USA are accepted.

- For the `Phone` input field, inputs should follow the common format of numeric country code, area code and local number.

- To avoid contact problems, the parcel services require that the numeric country code of the `Phone` and the symbolic country code of the `Country` must not belong to different regions.

A Java implementation of the ordering web service is depicted in Listing 2.1. Different approaches to implement the EH exist. A simple approach is the so-called *return code* pattern[1] where different return values either indicate valid results of an operation or code that represent exception occurrences. For the sake of simplicity, we use the return code pattern in our examples.

---

[1] See Subsection 3.2.2 for more information.

```
1 enum ShippingType { Standard, Express };
2
3 String processOrder(int packages, String country,
4                     String phone, ShippingType shipping){
5   if(isInvalidNumberOfPackages(packages)) return "INV_PACKAGES";
6   if(isInvalidCountry(country)) return "INV_COUNTRY";
7   if(isInvalidPhoneNumber(phone)) return "INV_PHONE";
8   if(isInvalidCountryCode(country, phone)) return "INV_CODE";
9
10  if(isDomesticCountry(country)) {
11    if(isStandardShipping(shipping)) {
12      processDomesticStandardOrder(packages, phone);
13    } else {
14      processDomesticExpressOrder(packages, phone);
15    }
16  } else {
17    if(isStandardShipping(shipping)) {
18      processForeignStandardOrder(packages, country, phone);
19    } else {
20      processForeignExpressOrder(packages, country, phone);
21    }
22  }
23
24  return "SUCCESS";
25 }
```

Listing 2.1: Implementation of the Ordering Web Service Example

```
Parameters:
  p_1 : Packages:  1, 3, 123
  p_2 : Country:   UK, USA, 123
  p_3 : Phone:     +44, +1, 123
  p_4 : Shipping:  Standard, Express
Constraints:
  c_1 : Packages ≠ 123
  c_2 : Country ≠ 123
  c_3 : Phone ≠ 123
  c_4 : ¬(Country = UK ∧ Phone = +1) ∧ ¬(Country = USA ∧ Phone = +44)
```

$$p_1 : \text{Packages:} \quad 1, 3, 123$$
$$p_2 : \text{Country:} \quad \text{UK, USA, 123}$$
$$p_3 : \text{Phone:} \quad \text{+44, +1, 123}$$
$$p_4 : \text{Shipping:} \quad \text{Standard, Express}$$
$$c_1 : \text{Packages} \neq 123$$
$$c_2 : \text{Country} \neq 123$$
$$c_3 : \text{Phone} \neq 123$$
$$c_4 : \neg(\text{Country} = \text{UK} \wedge \text{Phone} = \text{+1}) \wedge \neg(\text{Country} = \text{USA} \wedge \text{Phone} = \text{+44})$$

Figure 2.1.: IPM for the Ordering Web Service Example

To test the web service, Figure 2.1 depicts the corresponding IPM which consists of parameter values that should be tested and *constraints* that describe relations among those parameter values. In Section 3.1.2, the concept is explained in more detail. By convention, we use `123` to commonly indicate an invalid value that is, for instance, malformed or empty. For the sake of simplicity, the identified parameter values are considered to be exhaustive and testing beyond those values is not necessary. `UK` is selected to represent the domestic country and `USA` to represent another foreign country. Further on, `+1` represents a phone number that is located in the `USA` and `+44` represents a phone number that is located in the `UK`.

Constraints are expressed in propositional logic and describe the business rules in terms of relations between parameter values. For instance, c_1 describes that `Packages` other than 1 or 3 are not valid. As another example, c_4 describes that `UK` as a country of destination requires a phone number with the country code `+44`. Each order that satisfies all conditions is expected to pass the validation and to be processed. If at least one condition is not satisfied, an error message is expected to be returned from the web service. Constraints like these are later used to either exclude certain value combinations from testing or to separate them into different test suites.

Chapter 3.

Conceptual Foundations

Contents

In this chapter, we first describe the basic taxonomy of dependable programs that is used throughout this work. Since this work is about improving the CT test method in the context of EH and input masking, a precise description and clear distinction of robustness and related concepts is important. Therefore, we extend the initial taxonomy to describe robust programs and to distinguish robust programs from correct programs. At the end of both sections, summaries of key concepts are given.

3.1. A Taxonomy of Dependable Programs

First, the basic concepts of systems, programs, and dependability are introduced. The concepts are mainly derived from the taxonomy of dependable and secure computing by AVIŽIENIS et al. [Avi+04] and the IEEE standard glossary of software engineering terminology [IEE90b]. Afterwards, the foundations of testing and the foundations of combinatorial testing are discussed.

3.1.1. Dependable Programs

System

The generic theory of systems allows to uniformly describe hardware, software, programs, humans, and also the physical world including its natural phenomena [Avi+04].

Definition 1 (System) *A system is an entity that interacts with other entities, i.e. other systems [IEE90b]; [Avi+04].*

In general, any identifiable mechanism can be considered as a system if it shares one or more interfaces with its environment at which the system interacts with its environment [AL85]. To

interact means to share information between systems in order to stimulate the behavior of other systems and to receive responses from other systems.

A system is hierarchically structured [AL85]. It is either atomic where the internal structure is not examined any further or it consists of components interacting with each other where each component is itself a system [Avi+04]. Every system can be viewed as a component of another system. The containing system is the super-system and the contained system is the sub-system [Teu99, p.10].

The concepts of interface and environment require a system to have a boundary to distinguish and separate it from other systems. For a given system, its components are internal systems within the boundary. All other systems are external to its boundary.

Definition 2 (Environment) *For a given system, the other external systems constitute the environment [Avi+04].*

A system interacts with other systems of its environment via interfaces. Interfaces are shared boundaries that connect two or more systems to pass information from one system to the others [IEE90b]. When passing information, one system can be user of other systems and provider for other systems.

Definition 3 (User) *A user is a system that receives services from providing systems [Avi+04].*

Definition 4 (Provider) *A provider is a system that delivers a service to its users [Avi+04].*

Definition 5 (Interface) *An interface is a place of a provider's system boundary where service is requested by users and service is delivered to users [Avi+04].*

Where the boundary between a system and its environment is drawn and how a system is organized internally depends on the intensity of interactions between sub-systems and the intended functions of the system that it is expected to realize [IEE90b], [Teu99, p.10].

Definition 6 (Function) *A function of a system is what a system is intended to do [Avi+04].*

Definition 7 (Behavior) *The behavior of a system is what a system does to realize its functions [Avi+04].*

A system can also be described in terms of its state which consists of computation, communication, stored information, interconnection, and physical condition [Avi+04].

Definition 8 (State) *The state of a system is the set of "[...] values assumed at a given instant by the variables that define the characteristics of a system [...]" [IEE90b].*

In order to realize its functions, a system performs a sequence of state transitions [Avi+04]; [AL85]. Therefore, the behavior of a system can be described as the sequence of states that a system passes through.

A system – as it is viewed here – engages in a stimulus-response behavior such that it provides a response to each stimulus [AL85], [Wie03, p.23].

Definition 9 (Stimulus) *A stimulus of a system is a specific event at the interface of the system that is caused by the environment [Wie03, p.26].*

Definition 10 (Response) *A response is a specific event at the interface of a system that is caused by the system [Wie03, p.31].*

A system is stimulated from the environment in order to receive a response from the system. When stimulated, the system passes through a sequence of state transitions and responds to the stimulus.

Program

Programs are the central subjects that are under investigation in this work. Since they are part of software and executed on hardware, these concepts are introduced first.

Software is abstract and intangible [Som16, p.18]. To define software, it can be opposed to hardware which encompasses everything tangible [LL13]. Following the IEEE standard glossary of software engineering [IEE90b], we use the subsequent definitions of hardware and software.

Hardware is the "physical equipment used to process, store, or transmit computer programs or data" [IEE90b]. Hardware not only encompasses computers and internals of computers, e.g. CPU, RAM, or graphic board, but also encompasses peripheral devices for input, e.g. keyboards, microphones, or sensors, and output, e.g. displays, speakers, or printers [IEE90b], that allow interacting with humans and other hardware [Teu99, p.23]. Further on, hardware encompasses infrastructure to physically connect different hardware, e.g. network cable, router, or repeater, is also included [IEE90b], [Teu99, p.23].

Software is a set of "programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system" [IEE90b] and a *computer system* is a super-system that consists of "[...] one or more computers and associated software" [IEE90b]. The definitions of hardware and software are based on the further concepts which are also defined by the IEEE standard glossary of software engineering [IEE90b]. The *possibly associated documentation* of software is neglected in this work since we focus on the behavior and functions of software and in particular on the behavior and functions of programs.

Definition 11 (Program) *A program is "a combination of computer instructions and data definitions that enables [...] hardware to perform computational or control functions" [IEE90b].*

A *computer instruction* is an expression written in a programming language that can be executed by hardware [IEE90b]. *Data definitions* are descriptions of format, structure and other properties of data that is processed by the computer program [IEE90a]. For the sake of simplicity, we consider data definitions as special types of computer instructions. Also, we do not distinguish between computer instructions that are readable by humans and machine instructions that can be interpreted and executed by hardware [IEE90b].

A *procedure* is "a portion of a program that is named and that performs a specific action" [IEE90b]. It is considered as means to structure computer instructions within a program.

From a *structural point of view*, a program can be decomposed into sub-systems of which each sub-system is a program as well. Computer instructions are the most basic and atomic

sub-systems of a program. Depending on the programming language and its language features, different types of *blocks*, e.g. components, modules, packages, classes, methods, subroutines, or paragraphs, are used to structure atomic sub-systems and to compose procedures and programs [Ran75].

From a behavioral point of view, the execution of a program can be described by its state and a sequence of state transitions [PY08, p.58]. In contrast to the generic definition of a system's state (See Definition 8, Page 18), AMMANN & OFFUTT [AO16] define the state of a program more specifically "during execution [...] as the current value of all live variables and the current location, as given by the program counter".

A program engages in stimulus-response behavior and interacts with other systems of its environment. The interactions encompass hardware, other programs, and humans that stimulate or provide responses to the program [Som16, p.144].

Other programs with which a program may interact belong to the category of system and support programs such as the operating system, hardware drivers, databases, or middleware systems [Bal09, p.5], [Teu99, p.23]. This is the environment in which programs are executed [Som16, p. 57]. Application system is another type of software that consists programs with which a program may interact. The intention of application systems is to support or realize operational tasks of its users [IEE90b]; [Bal09, p.5]; [Teu99, p.23]. Examples of application systems are enterprise resource planning systems and workflow management systems [Wie03, p.3]. An enterprise resource planning system provides an integrated view on the production process of an enterprise. Its functionality can range from recommending production schedules to ordering of spare parts from suppliers. Workflow management systems can be used to semi-automate business processes, to monitor tasks and allocate manual tasks to employees.

Programs are not only part of a computer system where they interact with hardware and other programs. They are also embedded in a more extensive and broader socio-technical system where they interact with humans. A socio-technical system is a super-system of programs that involves technical elements, i.e. computer systems, but it also involves non-technical elements such as people, processes, and regulations [Som16, p.291]. It follows human, social, or organizational purposes that generate or constrain the functions that a program is intended to realize.

The lifecycle of a program has two different phases, i.e. development and usage, which also determine how other programs and humans interact with the program [Avi+04]. During the development phase, humans develop programs by writing computer instructions and by composing them into paragraphs, components, programs, etc. in order to realize the intended functions. Therefore, humans use hardware and other programs to write, organize and store computer instructions. During the usage phase, humans administrate and use programs [Avi+04]. To administrate is to prepare, deploy, and maintain a program such that it is ready for use in its environment [IEE90b]. To use is to interact with a program that is executed on hardware following the patterns of stimulus and response.

Since a program is intangible, it is not limited by properties of material, physical laws, or manufacturing processes [Som16, p.18]. Each copy of a program is identical to the original and reuse of a program can be highly advantageous [LL13]. In order to reuse a program, it can be made configurable so that it can be adapted to the needs of different computer systems and socio-technical systems [Som16, p.453]. Examples of configurable programs are software fam-

ilies [Par76], highly-configurable systems [CDS07] or software product lines [PBL05]. Then, a program is developed generically to cover, for instance, a particular business activity or business type [Som16, p.454]. The structure of a program can be divided into a common part which is always relevant and a variable part that is only relevant for the usage in certain computer systems and socio-technical systems [AG01]. The variable part is parameterized and configuration mechanisms are built into the program to enable or disable features and to choose between different implementations of the same feature [Cza99, p.8]; [Som16, p.453].

Different implementation techniques exist to realize configuration mechanisms at compile-time or run-time [AG01]. Examples are conditional compilation where source code regions are enabled or disabled at compile-time, dynamic class loading where classes are chosen and loaded at run-time, polymorphism and design patterns where implementations are exchanged at run-time, or aspect-orientation where meta-programming is used to modify and inject additional instructions at run-time.

In this investigation on programs, the state of a program, its configuration and also other aspects can be important. Therefore, we use *input* and *output* as an abstraction over the stimulus, response, configuration, and state spaces of a program.

Definition 12 (Input Domain) *The input domain I denotes the set of all possible inputs.*

Definition 13 (Input) *An input $\tau \in I$ may encompass anything explicable that can actually stimulate or affect the behavior of a program and anything explicable that can be expected to stimulate or affect the behavior of a program.*

For instance, BINDER [Bin94] considers configuration, state of a program, and also responses of external systems as input. BARR et al. [Bar+15] consider "configuration and platform settings, database table contents, device states, resource constraints, preconditions, typed values at an input device, inputs on a channel from another system, sensor inputs, and so on" as input.

Definition 14 (Output Domain) *The output domain O denotes the set of all possible outputs.*

Definition 15 (Output) *An output $o \in O$ encompasses anything explicable that can be actually observed after a program has been stimulated and anything explicable that can be expected to be observed after a program has been stimulated.*

Output may not only include response of a program but also include the state of the program as well as stimuli to other systems and their states. For instance, Barr et al. [Bar+15] consider "anything that can be discerned and ascribed a meaning significant to the purpose of testing – including values that appear on an output device, database state, temporal properties of the execution, heat dissipated during execution, power consumed, or any other measurable attributes of its execution".

Formally, a program can be modeled and viewed as a system P and represented as a mathematical function $P : I \rightarrow O$ that maps inputs $\tau \in I$ to outputs $o \in O$. $P(\tau)$ denotes the execution of a program P with input $\tau \in I$ producing an output $o \in O$. This view on systems is referred to as a *transformational* view because it emphasizes the input-output relationship of

the functions that are realized by the system [HP85]. However, this view is not limited to systems that receive one stimulus and produce one response. While the system passes through a sequence of state transitions, it may interact with other systems to stimulate them and to request responses that are required to fulfill the system's functions.

The transformational view is often contrasted with a reactive view on systems that emphasizes a system's continuous interaction and maintenance of relationships with its environment [HP85], [Wie03, p.5]. The reactive view on systems covers more aspects of the actual system but is also more complex. Here, the transformational view is chosen because it is simpler and sufficient.

In order to be able to reason about programs, two properties of programs that this work relies on are observability and controllability. Observability describes the ability of observing the behavior of a program in terms of its output [Bin94]. If the output cannot be observed, no statements about a program's behavior can be made [Bin94]. Controllability describes the ability to which the input of a program can be controlled [Bin94]. If the input cannot be controlled, it is unclear what has caused the output [Bin94]. In other words, if an input is used to stimulate a program several times, the output should be the same every time [Fre91]. If the output is not the same, not all parts of input are correctly identified. It is described as the so-called *input inconsistency effect* where the output is not only functionally-related to the identified input but also to some other parts which are not yet identified [Fre91].

Dependability

Since programs are deeply embedded in our lives, it is important that programs are trustworthy [Som16, pp.283-286]. Programs should operate as intended without undesirable side effects. Dependability is an established concept to describe "the ability of a system to deliver [an output] that can be justifiably trusted" [Avi+04]. It is an integrating concept that consists of several attributes such as availability, reliability, safety, and others [Avi+04].

In different contexts, different characteristics of dependability are appropriate [PY08, p.10]. In this work, we first focus on dependability in general and later introduce correctness and robustness as the main attributes of dependability.

Definition 16 (Specification) *A specification describes the functions that a program must realize and constrains its development and execution [IEE90b], [Som16, p.53].*

Definition 17 (Dependable Program) *A program is dependable if and only if its behavior is consistent with its specification [PY08]; [Avi+04, p.43].*

A specification is typically used to decide whether the behavior of a program is dependable or not. In practice, however, specifications may be incomplete, imprecise, and ambiguous which leaves room for interpretation and makes it impractical to come to a decision [AL85]. Nevertheless, an authority must exist that ultimately decides whether or not the behavior of a program is consistent with its specification. Therefore, we consider the specification as an *authority* which cannot be challenged and functions as the ground truth.

To describe a specification S, we adapt the notation of CRISTIAN [Cri82]; [Cri84]; [Cri89] and partition the input domain I into anticipated and unanticipated input domains to describe

Figure 3.1.: Input Domain Structure Diagram

the intended behavior. See Figure 3.1 (Page 23) for a structure diagram of the input domain that is based on the notation of UML 2.5.2 [Coo+17]. Since the input domain I is broadly defined, it covers all inputs that can be processed by the program that is under investigation and all inputs for which the specification defines an intended behavior. The anticipated input domain $AI \subseteq I$ is the *main* subset of all inputs in I for which the specification defines intended behavior. The unanticipated input domain $UI \subset I$ is the possibly remaining subset of inputs in I that a program can technically process but for which no intended behavior is defined by the specification.

Formally, a specification S is a relation $S \subseteq I \times O$ between the input domain I and output domain O. The functions of a program are described via pairs $(\tau, o) \in S$ and each pair describes an intended behavior in the form of expected output $o \in O$ for an input $\tau \in I$. The notation also allows modeling non-determinism via several pairs that consist of the same input but map to different outputs.

Please note, the following concepts are used to structure the input domain as described by the specification. Later on, a test-specific model presents a view on the specification which is used to ultimately guide the selection of inputs for testing. Since a test-specific model is the result of a creation process, it may contain faults that cause a deviation from the authoritative specification. Whenever necessary, we use *truly* as a prefix to distinguish the ground truth concepts that are based on the authoritative specification from similar concepts that are based on a test-specific model which may deviate from the authoritative specification.

Definition 18 (Anticipated Input Domain) *The anticipated input domain $AI \subseteq I$ is a subset of the input domain for which some expected output is defined by the specification S, i.e. $\forall \tau \in AI, \exists o \in O$ such that $(\tau, o) \in S$.*

Definition 19 (Unanticipated Input Domain) *The unanticipated input domain $UI \subset I$ is a proper subset of the input domain for which no expected output is defined by the specification S, i.e. $\forall \tau \in UI, \forall o \in O, (\tau, o) \notin S$.*

The anticipated and unanticipated input domains are complete and disjoint subsets of the input domain, i.e. $I = AI \cup UI$ and $AI \cap UI = \emptyset$. Further on, truly-relevant and truly-irrelevant inputs can be distinguished according to whether they belong to the anticipated or unanticipated input domain.

Definition 20 (Truly-Relevant Input) *An input τ is truly-relevant if and only if it is anticipated by the specification S, i.e. $\tau \in AI$.*

Definition 21 (Truly-Irrelevant Input) *An input τ is truly-irrelevant if and only if it is not anticipated by the specification S, i.e. $\tau \in UI$.*

A program P is dependable if and only if its behavior realizes the functions and if the program's outputs match the expected outputs of specification S for the entire anticipated input domain AI, i.e. $\forall \tau \in AI, (\tau, P(\tau)) \in S$. A program P is not dependable if at least one truly-relevant input exists that is *failure-causing* and for which the output of the program does not match the expected output of the specification S, i.e. $\exists \tau \in AI, (\tau, P(\tau)) \notin S$.

Definition 22 (Failure-Causing Input) *A truly-relevant input $\tau \in AI$ is a failure-causing input if and only if the output of the program $P(\tau)$ does not match the expected output of the specification, i.e. $(\tau, P(\tau)) \notin S$.*

When a program is not dependable, the concepts of failure, error, and fault describe the inconsistencies in more detail.

Definition 23 (Failure) *A failure is an event that occurs if the output of a program deviates from the expected output as described by its specification [Avi+04].*

Definition 24 (Error) *During the execution of a program, an error is the difference between the state of the program and the hypothetically correct state [Avi+04].*

Since the state of a program is defined by the value of all variables [AO16], an error can also be described as the "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition" [IEE90b].

An error in a program's state can lead to errors in subsequent states during execution [Avi+04]. When an error infects the output of a program, it causes a failure. However, many errors do not infect the output and may even remain unnoticed.

Definition 25 (Fault) *A fault is the identified or hypothesized cause of an error [Avi+04]. It is a static defect that manifests as an incorrect computer instruction [IEE90b]; [AO16]; [LO17].*

Faults can often be considered as human-made either caused by the absence of actions or caused by the performance of wrong actions when administrating, developing or using the program [Avi+04]. Malicious and non-malicious human-made faults can be distinguished [Avi+04]. Non-malicious human-made faults can be introduced by *mistakes* that the human is not aware of and by *bad decisions* that the human could not foresee, or simply by incompetence. The important distinction is that the faults are introduced accidentally without malicious objectives. In contrast, malicious human-made faults are introduced with malicious objectives like disrupting or stopping a service or accessing confidential information.

Symbols

$P : I \rightarrow O$	program
I	input domain

$\tau \in I$	input
O	output domain
$o \in O$	output
$P(\tau)$	execution of program P
$AI \subseteq I$	anticipated input domain
$\tau \in AI$	truly-relevant input
$UI \subset I$	unanticipated input domain
$\tau \in UI$	truly-irrelevant input
$S \subseteq I \times O$	specification
$(\tau, o) \in S$	specified output $o \in O$ for truly-relevant input $\tau \in AI$

3.1.2. Combinatorial Testing

Foundations of Testing

Testing is a primary approach to evaluate and ensure dependable programs [AO16]. In this context, programs and sub-systems of programs are also called SUT when they are the subject under investigation. The purpose of testing is to evaluate the conformance of a SUT to its specification and to determine if a SUT contains any faults [GG75]; [Bou85].

Testing can be understood as an activity that follows a test method. In general, a method is a goal-oriented and systematic approach to solve a theoretical or practical task [Mit13]. To achieve its goals, a method encompasses a process model that describes the necessary activities and the order in which they have to be performed [Bra+05]. Techniques are another important part of methods which are detailed instructions on how to perform the necessary activities [Bra+05]. In addition, tools support a test method by automating and assisting the performances of techniques. Following this terminology, a test method can be defined as follows.

Definition 26 (Test Method) *A test method is a particular method with the objective of evaluating the conformance of a SUT to its specification.*

A test method consists of a testing process, testing techniques and testing tools. The testing process defines necessary activities of preparation, execution, and post-processing and the order in which these activities should be performed. Testing techniques provide detailed guidelines and describe how to perform necessary activities. Testing tools automate or assist the execution of testing techniques.

Testing is a dynamic activity where the evaluation of a SUT is based on actual executions [IEE90b]. Therefore, a SUT is ultimately stimulated with inputs and the outputs of the SUT are evaluated.

Definition 27 (Test Input) *An input τ that is used for testing is a test input.*

Definition 28 (Test Suite) *A test suite T is a set of test inputs from the input domain $T \subseteq I$ that is used for testing.*

Definition 29 (Test Execution) *Test execution is the activity of stimulating a SUT with test inputs $\tau \in T$, observing its outputs $o \in O$ and checking if the outputs conform to the specification [IEE90b]; [LO17].*

Whether testing detects failures depends on two key factors which are test oracles and the selected test inputs [LO17]. Checking if the output conforms to the specification is the task of a test oracle. A test oracle assigns a label to each test input which is either `pass` or `fail`.

Definition 30 (Test Oracle) *A test oracle is a mechanism that checks the observed outputs of a test and decides whether the outputs match the expected outputs of the specification [Wey82]; [LO17].*

We use the predicate $OK(\tau, P(\tau))$ to indicate a pass of test input τ because the test oracle determined that the observed output $P(\tau)$ conforms to the specification [GG75]. A test input τ for which the test oracle determines that the output of program $P(\tau)$ does not conform to the specification is a failure-causing test input which exposes a fault [WWH91]. A failure-causing test input is indicated by $\neg OK(\tau, P(\tau))$.

Testing requires the existence of test oracles since no conclusions can be drawn otherwise [AL85]. The *oracle assumption* captures the belief that correct test oracles which reveal all failures of a SUT exist or can be constructed with reasonable effort [Wey82]; [AL85]; [LO17]. However, the oracle assumption does not necessarily hold in practice. But, even when the oracle assumption does not hold, the concepts of testing are still applicable with limited informative value and less exploratory power. In fact, construction of test oracles is another dedicated branch within the area of research on testing. For more information, please refer to related work (cf. [Wey82]; [Bar+15]; [LO17]; [PH18]).

Further on, testing involves several executions of the SUT with various test inputs and implies that each execution terminates within a reasonable amount of time [Gou83]; [Mor90]. If it does not terminate, it is assumed that the test oracle labels the test input as failed after time constraints as defined by the specification are exceeded.

Failure-causing inputs and failure-causing test inputs could be further distinguished because a failure-causing input is defined in terms of the authoritative specification, i.e. $(\tau, P(\tau)) \notin S$, while a failure-causing test input is defined in terms of the test oracle, i.e. $\neg OK(\tau, P(\tau))$.

But for the sake of simplicity, we do not consider discrepancies. We assume the existence of correct test oracles and consider $OK(\tau, P(\tau))$ to be equivalent to $(\tau, P(\tau)) \in S$. The main difference between a failure-causing input and a failure-causing test input is that the latter is selected for testing. Furthermore, a failure-causing input is always a truly-relevant input because no expected output is specified for truly-irrelevant inputs. However, test-specific models define a classification of relevant and irrelevant test inputs subsequently. The classification may deviate from the classification as imposed by the authoritative specification and not all failure-causing

test input may be classified as relevant. Then, test inputs can be failure-causing even if they are incorrectly classified as irrelevant.

Assuming that test oracles reveal all propagated failures, the important factor in testing is the selection of test inputs such that faults are triggered and propagate to failures. Testing with all inputs, i.e. $T = I$, is said to be exhaustive [GG75]. But it is almost always infeasible due to a too large input domain. Therefore, a proper subset $T \subset I$ is typically used as a test suite.

While using a non-exhaustive test suite makes testing feasible, it also reduces the informative value and exploratory power of testing. When at least one test input fails, i.e. $\exists \tau \in T, \neg OK(\tau, P(\tau))$, it can be concluded that the program contains faults. However, when all test inputs pass, i.e. $\forall \tau \in T, OK(\tau, P(\tau))$, it cannot be concluded that the program does not contain any fault. This would be an infinite conclusion from finite knowledge, i.e. $\forall \tau \in T, OK(\tau, P(\tau)) \Rightarrow \forall \tau \in I, OK(\tau, P(\tau))$ [GG75]; [How76]; [WO80]; [Gou83]; [Bou85].

GOODENOUGH & GERHART [GG75] introduced and formalized the concept of ideal testing with ideal test suites that allow drawing that infinite conclusion. However, HOWDEN [How76] showed that ideal testing is not practical since no general approach exists to construct ideal test suites for arbitrary programs and specifications. In addition, WEYUKER & OSTRAND [WO80] discussed additional flaws and showed that only the exhaustive test suite fulfills the properties of an ideal test suite. Hence, the question how to decide that a program has been tested thoroughly enough such that its conformance to its specification can be judged with reasonable confidence remains an open problem in research on testing [Wey86].

The common approach to deal with this problem is to systematize the selection of test inputs via fault models, test adequacy criteria and test selection strategies which are explained subsequently. Test adequacy criteria allow guiding the systematic selection of test inputs and to assess the quality of test suites. Based on test adequacy criteria, conclusions whether a program is tested sufficiently can be made by assessing whether test suite T satisfies a test adequacy criterion and by comparing a test adequacy criterion to other test adequacy criteria [Gou83]; [Wei89]; [Wei90]; [WWH91]; [FW93b]; [FW93a]; [Zhu96]; [ZHM97]; [Hie02]; [Wey02].

Definition 31 (Test Adequacy Criterion) *A test adequacy criterion is a predicate that specifies conditions that must be satisfied by a test suite T [WO80]; [WJ91].*

Test adequacy criteria are usually defined relative to a SUT, its specification, or both [Wey86]; [ZHM97]. A test adequacy criterion is a black-box test adequacy criterion when it is defined independently from the SUT and only relative to the specification [ZHM97].

Test adequacy criteria define "what properties of a [SUT] must be exercised to constitute a *thorough* test" [GG75]. The properties are usually based on hypothesized characteristics and faults which are believed to be contained by the program [Bin00]; [Hie02].

Definition 32 (Fault Model) *A fault model is a description of hypothesized characteristics and faults that the SUT is believed to contain [Bin00].*

The test adequacy criterion is typically defined such that its satisfaction by test suite T implies – or at least improves the likelihood – that testing with test suite T activates the hypothesized faults if they are present in the program [Bou85]; [Bin00, p.65] [Pre+13].

A test selection strategy is a particular technique that provides detailed instructions on how to choose test inputs from the anticipated input domain and when to stop.

Definition 33 (Test Selection Strategy) *A test selection strategy is a technique that describes how to systematically select a subset of test inputs from the input domain $T \subseteq I$ such that one or more test adequacy criteria are satisfied [Gou83]; [ZHM97]; [Hie02].*

Just as with test adequacy criteria, a test selection strategy can be defined relative to a SUT, its specification, or both. A test selection strategy is a black-box test selection strategy when it is defined independently from the SUT and only relative to the specification.

When using a test selection strategy, the test adequacy criterion functions as a stopping rule that determines when enough test inputs are selected [Wey86]; [ZHM97]. Typically, a test adequacy criterion describes a lower boundary of test inputs that must be selected. Once the lower boundary of test inputs is reached, no additional test inputs are needed [ZHM97]. Therefore, test selection strategies usually aim to select a minimum number of test inputs such that the corresponding test adequacy criteria are satisfied.

Fault model, test adequacy criterion and test selection strategy form a functional unit [Gou83]; [ZHM97]. Following the instructions of the test selection strategy, test suites are generated that satisfy the corresponding test adequacy criterion. These test suites exhibit characteristics such that they can expose faults that are hypothesized by the fault model.

Foundations of Combinatorial Testing

CT is a test method with particular black-box test adequacy criteria and particular black-box test selection strategies that depend on a model of the SUT's input domain which is called IPM [GOA05]. Figure 3.2 depicts the idealized CT test process which we adopted from GRINDAL & OFFUTT [GO07]. As a first activity, the tester has to choose one or more test selection strategies. Then, the tester has to model an IPM. In activity three, the test selection strategies are applied to select test inputs based on the IPM. Afterwards, the test suite can be evaluated. If it is considered inadequate, the IPM and test selection strategies can be adjusted. Otherwise, the test suite is used in the test execution activity. Afterwards, the results of the test execution are evaluated and conclusions are drawn.

An IPM can be understood as a model of the specification that describes a subset of the input domain with the intention of selecting test inputs from it. Formally, an IPM is a 3-tuple $IPM = \langle \tilde{P}, V, C^{ex} \rangle$ [GOA05]; [NL11a]. It consists of a set of n input parameters $P = \{p_1, ..., p_n\}$ and a set of n value domains $V = \{V_1, ..., V_n\}$ that structure a subset of the input domain I of the SUT. Each parameter $p_i \in \tilde{P}$ represents an explicable unit of the input domain and is associated with a domain of values, i.e. a nonempty set V_i of m_i discrete values. C^{ex} is a set of exclusion-constraints which is further explained in subsequent paragraphs.

Definition 34 (Input Parameter Model) *An IPM is a specification-based model of the input domain that is used by the test selection strategy to select test inputs [GOA05]. It structures the input domain via parameters, associated value domains, and exclusion-constraints.*

Following the structure of the input domain via parameters and values, test inputs can be described as sets of parameter-value pairs such that exactly one value for each parameter is

Figure 3.2.: Idealized CT Test Process

contained. Let $\pi = (p_i, v_j)$ be a parameter-value pair that denotes an assignment of value $v_j \in V_i$ for parameter p_i. A test input is a set of parameter-value pairs for n distinct parameters: $\tau = \{\pi_1, , .., \pi_n\} = \{(p_{i_1}, v_{j_1}), ..., (p_{i_n}, v_{j_n})\}$.

The IPM as a model of a specification describes an input domain from which test inputs are selected by combinatorial test selections strategies. We denote this input domain as the exhaustive test suite E which is a discrete subset of the SUT input domain $E \subseteq I$ and a superset $T \subseteq E$ of each test suite T.

Definition 35 (Exhaustive Test Suite) *The exhaustive test suite E contains all test inputs that are modeled by the IPM, i.e. $E = V_1 \times ... \times V_n$.*

The concept of schemata generalizes this formalism and allows to represent test inputs and also *partial* test inputs that do not contain parameter-value pairs for all parameters. A test input is a schema with degree $d = n$, a value is a schema with degree $d = 1$, and a value combination is a schema with degree $d > 1 \leq n$. We use τ as a symbol to describe schemata and test inputs in general. When the distinction between schemata and test inputs is important, s is used as another symbol to describe schemata.

Definition 36 (Schema) *A schema τ is a set of parameter-value pairs for d $(1 \leq d \leq n)$ distinct parameters $\tau = \{(p_{i_1}, v_{j_1}), ..., (p_{i_d}, v_{j_d})\}$ [NL11b]; [Yil+14a].*

Definition 37 (Degree of Schema) *The number of parameter-value pairs that a schema covers denotes the degree of a schema, i.e. $d = |\tau|$.*

An important relation between two schemata is the coverage relation where one schema covers another schema. Please note, the relation is reflexive such that each schema τ covers itself.

Definition 38 (Coverage) *Coverage denotes a relation between two schemata τ_a and τ_b where schema τ_a covers schema τ_b (denoted as $\tau_b \subseteq \tau_a$) if and only if for every parameter-value pair (p_i, v_j) of τ_b the same parameter-value pair exists in τ_a, i.e. $\forall (p_{i_b}, v_{j_b}) \in \tau_b, \exists (p_{i_a}, v_{j_a}) \in \tau_a$ such that $i_a = i_b \wedge j_a = j_b$.*

Test modeling is an important activity as it defines the input domain for testing [NL11a]. If a failure-causing test input is not considered by the IPM, it cannot be selected by the test selection strategy and cannot fail during test execution. As a consequence, the fault that is linked to the failure-causing test input may not be detected. Therefore, all *important*[1] values should be modeled. Different test modeling techniques can be used to identify parameters and values for each parameter [GOM07]. For parameters with value domains that are continuous or potentially infinite, a discrete subset of values must be selected, for instance, by using equivalence partitioning or boundary value analysis [GOA05].

To further analyze causes of failures, the concept of failure-causing (test) inputs is extended to failure-causing schemata.

Definition 39 (Failure-causing Schema) *A truly-relevant schema s is a failure-causing schema if and only if each truly-relevant input $\tau \in AI$ that covers $s \subseteq \tau$ is a failure-causing input, i.e. $s \subseteq \tau \Rightarrow (\tau, P(\tau)) \notin S$ and $s \subseteq \tau \Rightarrow \neg OK(\tau, P(\tau))$.*

Further on, the concept of minimality allows to narrow down the parameter values that are held responsible for causing the failure [NL11b].

Definition 40 (Minimal Failure-causing Schema) *A failure-causing schema s is a minimal failure-causing schema (MFCS) if and only if no proper subset schema $s' \subset s$ exists that is itself a failure-causing schema. Then, for each proper subset schema $s' \subset s$, a relevant input $\exists \tau \in AI$ exists that covers $s' \subset \tau$ and is not failure-causing, i.e. $(\tau, P(\tau)) \in S$ and $OK(\tau, P(\tau))$.*

A *combinatorial* test adequacy criterion commonly used in CT is t-wise coverage where t is a positive integer that denotes the testing strength [KKL13].

Definition 41 (t-wise Coverage) *t-wise coverage is a test adequacy criterion that is satisfied by a test suite if all schemata of an IPM with degree $d = t$ are covered by at least one test input of the test suite [GOA05].*

t-wise coverage can be understood as a generalization of the each-choice ($t = 1$), pair-wise ($t = 2$), and N-wise ($t = n$) test adequacy criteria [GOA05]. Each-choice is a simple test adequacy criterion that requires each value that is modeled by the IPM to appear in at least one test input. Pair-wise is a test adequacy criterion that considers the interaction of parameter values. It requires that each pair of parameter values appears in at least one test input. t-wise coverage is simply an extension of pair-wise that provides a generalization from pairs to tuples of size t. The N-wise coverage test adequacy criterion denotes the upper limit of t-wise coverage since each tuple encompasses values of all parameters. Testing with N-wise coverage can be considered as exhaustive testing with respect to the IPM which is usually infeasible [Gri+06].

The corresponding t-factor fault model hypothesizes that faults are caused by the interaction of t parameters and can be reached by combinations of t parameter values (MFCSs with degree

---

[1]The term 'important' is inspired by GRINDAL et al. [GOA05] who use the term 'interesting' to refer to any value that the tester decides to use. Here, 'important' refers to any value that may reveal important findings when used to test the SUT.

$d = t$) [DM98]. The t-factor fault model is substantiated by empirical research that analyzed bug reports for different types of programs [KKL16]. The empirical research indicates that (1) most faults are reached by MFCSs with degree $d = 1$ and $d = 2$, (2) progressively fewer faults require MFCSs with degree $d \leq 3$, and (3) no discovered fault involved requires more than 6 parameters [KKL16]. But, it is important to note that these findings do not imply that MFCSs with degree $d > 6$ may not exist [KKL13]. The conclusion that can be drawn from these findings is that *most* failures are likely reached by MFCSs with degrees lower than 6. Consequently, testing should focus on MFCSs with degrees between $1 \leq k \leq 6$.

When testing a SUT with a test suite that satisfies t-wise coverage, i.e. a test suite that includes all parameter value combinations of size t, at least one test input covers a MFCS with degree $d \leq t$ and the SUT is expected to fail. Therefore, CT is considered an effective testing method [KKL13]. Sometimes, CT is even called *pseudo-exhaustive* testing (cf. [KO06]) because all t-way combinations are tested.

Another important aspect is the avoidance of the so-called input masking effect since it can prevent the identification of failure-causing test inputs. We adapt the concept from YILMAZ et al. [Yil+14a]:

Definition 42 (Input Masking Effect) *"The input masking effect is an effect that prevents a test [input] from testing all combinations of input values, which the test [input] is normally expected to test" [Yil+14a].*

Different types of input masking can be identified. Irrelevant test inputs are one cause of the irrelevant input masking effect. The test selection should be guided to only select test inputs of the anticipated input domain. Otherwise, test inputs could be selected that are irrelevant which means, for instance, that their expected behavior is not defined, they are not executable or they are unimportant and just not of any interest. Therefore, truly-relevant and truly-irrelevant schemata should be distinguished. These concepts are refinements of Definition 20 (Truly-Relevant Input) and Definition 21 (Truly-Irrelevant Input) that incorporate the structure of parameters and values. A truly-relevant input is also a truly-relevant schema and a truly-irrelevant input is also a truly-irrelevant schema.

Again, the prefix *truly* is used to highlight the ground truth classification of the authoritative specification. Another classification that is based on the IPM is provided afterwards.

Definition 43 (Truly-Irrelevant Schema) *A schema s is truly-irrelevant if and only if $\nexists \tau \in AI$ such that $s \subseteq \tau$.*

Definition 44 (Truly-Relevant Schema) *A schema s is truly-relevant if and only if $\exists \tau \in AI$ such that $s \subseteq \tau$.*

While it is easy to exclude values that are truly-irrelevant by simply omitting them from the IPM, it is more complicated for truly-irrelevant value combinations because they are the result of combining relevant values that cannot be omitted individually. In fact, most IPMs contain combinations of parameter values that should not be combined [GO07].

Running Example   Let's suppose the ordering web-service contains a fault where `is-DomesticCountry(country)` fails for foreign countries. It is a known fault and restriction of the current implementation which has implications for the testing as well. Since the ordering web-service fails for orders with foreign countries as its destination, testing with foreign countries has negative consequences. It can even cause irrelevant input masking and should, therefore, be excluded.

A test input like `[Packages: 1, Country: USA, Phone: +1, Shipping: Standard]` fails because of `[Country: USA]`. If it is the only test input that covers `[Shipping: Standard]`, the functions `is-StandardShipping(shipping)` as well as `processDomesticStantardOrder(packages, phone)` are not sufficiently tested.

Any test input that covers a truly-irrelevant schema should not be tested because the truly-irrelevant schema falsifies the result of the test. For instance, because the test input cannot be executed or the intended behavior for the truly-irrelevant schema is not defined. As a consequence, all the other values and value combinations of a test input that covers a truly-irrelevant schema remain untested as long as there is no other test input that contains them. The truly-irrelevant schema *masks* all other values and value combinations of the same test input. Because truly-irrelevant schemata cause the input masking effect, a tester should identify and exclude truly-irrelevant schemata from test suites.

An IPM with parameters and values is a purely syntactic structure of the input domain without any semantic information [GOA05]. To exclude truly-irrelevant schemata automatically, additional semantic information must be added to the IPM. Since the authoritative specification does not exist in a machine-interpretable form, the aforementioned set of exclusion-constraints C^{ex} is used to add the semantic information to the IPM. It allows to distinguish relevant from irrelevant schemata.

Definition 45 (Exclusion-Constraints) *The set of exclusion-constraints C^{ex} is a set of constraints to distinguish relevant from irrelevant schemata.*

In general, constraints describe characteristics that may be true or false for a given schema. A schema for which a set constraints evaluates to true is said to *satisfy* the constraints. Formally, $\text{SAT} : C^* \times A \to Bool$ denotes the satisfiability function to check whether a schema $s \in A$ satisfies a set of constraints $C \subseteq C^*$. In analogy to the exhaustive test suite E, we introduce A as a set that covers all schemata that are modeled by the IPM. It encompasses all test inputs $\tau \in E$ with $d = n$, and all schemata that are subsets of the test inputs, i.e. $A = \{s | \tau \in E \wedge s \subseteq \tau\}$. In addition, C^* is a set that contains all possible sets of constraints as modeled by the IPM.

Different ways to specify constraints and to realize the satisfiability function exist (cf. [Yu+13a]; [Yu+15]). A constraint can be represented as a logical expression in propositional logic or as a set of predefined schemata. Although, it is important to note that both representations of constraints, are equally powerful and both can be transformed into the other.

Both representations allow for different ways of realizing the satisfiability function. When using the representation of logical expressions, the constraints C and the schema under evaluation s are transformed into a CSP and a CSP *solver* searches if a test input τ exists that satisfies

all constraints and covers $s \subseteq \tau$ [RN10]; [Yu+13a]. When using the representation via sets of predefined schemata, it is checked if the schema under evaluation s does not cover any predefined schemata and if a test input τ exists that covers $s \subseteq \tau$ while not covering any predefined schemata [Yu+15]. Optimizations of this approach involve the computation of *minimal* schemata to reduce the number of coverage checks [Yu+15].

In this work, we rely on logical expressions to model constraints without restricting the realization of the satisfiability function. In the case of exclusion-constraints, schemata that satisfy the set of exclusion-constraint are relevant. Otherwise, they are irrelevant.

Definition 46 (Irrelevant Schema) *A schema τ is an irrelevant schema if and only if at least one exclusion-constraint $c \in C^{ex}$ is not satisfied, i.e.* $\mathtt{SAT}(C^{ex}, \tau) = false$.

Definition 47 (Relevant Schema) *A schema τ is a relevant schema if and only if it satisfies all exclusion-constraints, i.e.* $\mathtt{SAT}(C^{ex}, \tau) = true$.

The classification of irrelevant and relevant schemata is defined with respect to the IPM and its exclusion-constraints. It is not necessarily equivalent to the classification that is imposed by the authoritative specification. As a consequence, exclusion-constraints that are too strict or loose may classify schemata incorrectly.

> **Running Example**   The conditions $c_1, ..., c_4$ of the example can be used as exclusion-constraints. Consequently, the corresponding schemata would be classified as irrelevant and excluded. This is an undesirable outcome because we want to test the EH as well. The partition of valid and invalid schemata follows later on.
>
> Let's suppose the development of the ordering web-service is not finished yet. In fact, the connection between the ordering web-service and other web-services of the local parcel service is still incomplete. As a consequence, the ordering web-service is failing when it processes orders from foreign countries that select the standard shipping method. It is a known fault and restriction of the current implementation which has implications for the testing as well. Since the ordering web-service fails for orders that include standard shipping to foreign countries, testing that particular schema is of no interest. The schema might even cause irrelevant input masking and should, therefore, not be excluded.
>
> The logical expression `Shipping = Standard ⇒ Country ≠ USA` can be used as an exclusion-constraint to exclude the known fault. Alternatively, the equivalent formulae `¬(Shipping = Standard ∧ Country = USA)` or `(Shipping = Standard ∧ Country = UK) ∨ (Shipping = Standard ∧ Country = 123)` can be used as well. The first alternative formula highlights the irrelevant schemata. The second alternative formula highlights `[Country:123]` as a relevant value.

Specifying exclusion-constraints and realizing the satisfiability function is not sufficient. Test selection strategies must exclude irrelevant schemata while still preserving the test adequacy criteria [GOM07].

Definition 48 (Relevant t-wise Coverage) *Relevant t-wise coverage is a test adequacy criterion that is satisfied by a test suite if all relevant schemata with degree $d = t$ are covered by at least one relevant test input of the test suite [GOA05].*

Different ways to exclude irrelevant schemata exist. GRINDAL & OFFUTT [GOM07] discuss them as *conflict handling strategies*. Two strategies (abstract parameter, sub-models) involve re-modeling of new IPMs that do not model any irrelevant schemata. The replace strategy proposes to clone and replace irrelevant schemata of existing test suites. The advantage of all three strategies is that the test selection strategies may remain simpler as they do not have to deal directly with irrelevant schemata. But, the test suites may become unnecessarily big. A fourth strategy is to already avoid irrelevant schemata during test selection by embedding the satisfiability function into the test selection strategy (cf. [Yu+13a]; [Yu+15]). As a disadvantage, the test selection strategies may become more complex and the execution time may increase. However, the strategy can be fully automated and the test suites are smaller.

In empirical studies conducted by PETKE et al. [Pet+13]; [Pet+15], test selection strategies following the avoid strategy are applied to real-world IPMs. Even though there is room for improvement, the results indicate that the avoid strategy is applicable in real-world contexts. Therefore, we follow the avoid strategy in this work.

Symbols

$T \subseteq E$	test suite
$\tau \in E$	test input
$OK(\tau, P(\tau))$	passing test input τ
$\neg OK(\tau, P(\tau))$	failure-causing test input
$IPM = \langle \tilde{P}, V, C^{ex} \rangle$	input parameter model
$\tilde{P} = \{p_1, ..., p_n\}$	set of all parameters modeled by an IPM
n	number of parameters modeled by an IPM
$p_i \in \tilde{P}$	parameter with label i ($0 \leq i \leq n$)
$V = \{V_1, ..., V_n\}$	set of all value domains of all parameters
V_i	domain of values associated with parameter p_i
$v_j \in V_i$	value with a unique index j of parameter p_i
m_i	number of values modeled by V_i
$\pi = (p_i, v_j)$	parameter-value pair π with $v_j \in V_i$ for parameter p_i parameter value pair
$E \subseteq I$	exhaustive test suite

A	set of all schemata
$\tau = \{(p_i, v_j), ...\}$	schema
$s = \{(p_i, v_j), ...\}$	schema (alternative symbol)
$d = |\tau|$	degree of schema τ
$\tau_a \subseteq \tau_b$	coverage relation between two schemata
t	testing strength
$\mathtt{SAT} : C^* \times A \to Bool$	satisfiability function
c	constraint
$C \in C^*$	set of constraints
C^*	set of all possible constraint sets
C^{ex}	set of exclusion-constraints

Summary of Key Concepts

To conclude the taxonomy of dependable programs, Figure 3.3 depicts a structure diagram that is based on the notation of UML 2.5.1 [Coo+17]. The objective of this summary is to provide an overview of key concepts and relations in order to function as a reference point and in order to highlight the distinction of specification-based and IPM-based concepts. The structure diagram consists of named elements, aggregations, and generalization sets (cf. [Coo+17, p.119]). For reasons of clarity, names of generalization sets and cardinalities are omitted. Further on, each generalization set is considered disjoint. The diagram is divided into four quadrants which are pictured by dashed circles and correspond to four different views on the central concept of input $\tau \in I$ and its input domain I.

The upper left quadrant no. 1 corresponds to the concepts that are established by the authoritative specification. A specification S partitions the input domain and all inputs into truly-relevant inputs $\tau \in AI$ that belong to the anticipated input domain AI and into truly-irrelevant inputs $\tau \in UI$ that belong to the unanticipated input domain UI.

For all truly-relevant inputs $\tau \in AI$, expected outputs $o \in O$ are defined by the specification $(\tau, o) \in S$. When the output of a program P differs from the expected outputs of specification S, i.e. $(\tau, P(\tau)) \notin S$, the truly-relevant input is failure-causing and program P is not dependable.

Testing is the primary approach to evaluate whether a program is dependable. In the context of CT, an IPM is used as an explicit model of the specification S. An IPM defines parameters and values that constitute a subset of the input domain $E \subseteq I$. A test input is a particular input $\tau \in E$ that is modeled by an IPM. In addition, the parameters and values of an IPM allow utilizing a schema as a finer-grained structure that generalizes over single values, combinations of values, and complete test inputs. To make the concepts of truly-relevant, truly-irrelevant,

and failure-causing inputs applicable in the context of CT, they are transferred to its schema counterparts.

In the upper right quadrant no. 2, the aforementioned concepts are transferred from a purely specification-based view to a view that incorporates the structure of parameters and values as defined by an IPM.

Each schema $s \in A$ can be partitioned according to the authoritative specification: A schema s is truly-relevant if a truly-relevant input τ exists that covers the schema, i.e. $\exists \tau \in AI, s \subseteq \tau$. A schema s is truly-irrelevant if no such truly-relevant input exists, i.e. $\nexists \tau \in AI, s \subseteq \tau$.

Further on, a truly-relevant schema s is failure-causing when all truly-relevant inputs τ that cover the schema are failure-causing, i.e. $\forall \tau \in AI, s \subseteq \tau \Rightarrow (\tau, P(\tau)) \notin S$.

The concept of minimal failure-causing schemata utilizes the finer-grained structure and provides more detailed information about the cause of a failure. A failure-causing schema s is minimal if no proper subset $s' \subset s$ is failure-causing.

Both upper quadrants display concepts that are based on the authoritative specification. However, not the authoritative specification but an IPM is used to select test inputs for testing. Since an IPM is the result of a creation process, it may contain faults that result in a deviation from the authoritative specification. As a consequence, an IPM may classify test inputs incorrectly. To acknowledge potential deviations, the concepts of relevant, irrelevant, and failure-causing test inputs are introduced as counterparts to truly-relevant, truly-irrelevant inputs, and failure-causing inputs. The concepts are depicted in the lower left quadrant no. 3.

The exclusion-constraints C^{ex} of an IPM allow a partitioning of all test inputs $\tau \in E$ into relevant and irrelevant test inputs. A test input is relevant if all exclusion-constraints are satisfied, i.e. $\texttt{SAT}(C^{ex}, \tau) = true$. Otherwise, the test input is irrelevant, i.e. $\texttt{SAT}(C^{ex}, \tau) = false$.

Since the concept of failure-causing inputs relies on truly-relevant inputs, a counterpart is required as well. A test input τ is failure-causing if a test oracle determines that the output $P(\tau)$ does not conform to the specification, i.e. $\neg OK(\tau, P(\tau))$.

The classification of failure-causing test inputs is not limited to relevant test inputs because a truly-relevant input that is failure-causing may be incorrectly classified as irrelevant.

In the lower right quadrant no. 4, the partitioning of exclusion-constraints C^{ex} is combined with the structure of parameters and values. Hence, the concepts of irrelevant, relevant, and failure-causing schemata are analogous to the irrelevant, relevant, and failure-causing test inputs. In addition, the concept of minimal failure-causing schemata introduces a finer-grained structure that provides more detailed information about the cause of a failure.

Figure 3.3.: Key Concepts of Dependable Programs and its Relations

3.2. An Extended Taxonomy of Robust Programs

The CT test method is build on the taxonomy of dependable programs as introduced in the previous section. It relies on a uniform treatment that does not distinguish between different types of failures. As already stated in the introduction of this work, CT has shortcomings and potential improvements in the context of EH and input masking exist.

Now, we extend the previously introduced taxonomy to precisely distinguish different types of failures. Furthermore, we refine the concepts of dependable programs to describe robust and correct programs. Then, EH is introduced as a common approach to develop robust programs and issues related to EH are identified to further motivate the necessity of testing with invalid inputs. Afterwards, the introduced concepts are transferred to the domain of CT.

3.2.1. Robust Programs

As stated before, programs should operate as intended without undesirable side effects. One can distinguish two different types of faults that can make a program not to operate as intended. The two different types depend on the location of a fault and whether the fault is internal or external with regards to the program [Avi+04].

Definition 49 (Internal Fault) *For a given program, an internal fault is a fault that is located within the boundaries [Avi+04].*

Internal faults are typically introduced during the development phase of a system or during the maintenance of a program [Avi+04]. Examples of internal faults are development faults where incorrect computer instructions or data definitions are introduced or configuration faults where wrong settings affect the security, network access or storage of a program.

Definition 50 (External Fault) *For a given program, an external fault is a fault that is located outside the boundaries, i.e. located in the program's environment [Avi+04].*

External faults occur during the use phase when interacting with a program [Avi+04]. Most of the time, external faults are the result of human-made faults in the program's environment either by interacting incorrectly with the program or by introducing an internal fault when developing another program. Even though external faults occur outside of a program's boundary, they can still infect state by interaction and interference [Avi+04]. Consequently, external faults cannot only lead to errors in the program, but these errors can also propagate to failures. Examples of external faults are inputs from humans or other programs that are malformed, or stressful environmental conditions, e.g. other programs that do not respond within an expected time frame [IEE90b].

The classification of internal and external faults depends on the perspective [Avi+04]. As an example, an error made by a developer during the development phase results in a failure to write correct computer instructions which, in turn, results in a fault of the program. During the use phase of the program, the fault can produce an error which might propagate to a failure that affects the program's output. From the perspective of program A, the fault is internal. However, from the perspective of another program that does not receive a correct response, the fault is

external. Another example is the inappropriate usage of a program by humans which is – from the perspective of the used program – perceived as an external human-made fault with either malicious or non-malicious intentions [Avi+04].

Internal and external faults are threats that can affect the dependability of every program. But faults can only be removed from programs over which the developers have a sufficient level of control. To detect internal faults in programs, testing is the primary approach in industry [AO16]. Other programs of the environment and especially humans are beyond the control of developers.

Therefore, the presence of external faults must be acknowledged and programs must become tolerant towards external faults by considering, specifying and implementing appropriate responses. External faults must be detected and corresponding errors must be eliminated before they can lead to errors and propagate to failures [Avi+04].

The tolerance of a program towards external faults is called *robustness* which IEEE [IEE90b] defines as "the degree to which [a program behaves] correctly in the presence of [external faults, i.e.] invalid inputs or stressful environmental conditions". AVIŽIENIS et al. [Avi+04] describe robustness as "a specializing secondary attribute" and as "dependability with respect to external faults". According to PEZZÈ & YOUNG [PY08, p.18], robustness involves weighing properties of a program depending on their criticality and deciding which properties should be maintained in exceptional situations where the complete functionality cannot be guaranteed. LIGGESMEYER [Lig09, p.10] understands robustness as a characteristic of programs to provide reasonable responses even in exceptional situations. Further on, KOOPMAN & DEVALE [KD00] emphasize the scope of robustness "to be more than just catastrophic system crashes and encompasses situations in which small, recoverable failures might occur". What all the different definitions of robustness have in common is the emphasis on a program's *acceptable* behavior in the presence of external faults [SF13].

Since a specification constitutes the dependable behavior of a program, robustness is, first of all, a property of the specification [Lig09, p.10]. The specification has to define acceptable behavior in the presence of external faults. Therefore, we divide the specification $S = STS \cup ES$ into a *standard specification STS* and an *exception specification ES* [Cri89].

Definition 51 (Standard Specification) *The standard specification STS defines the functions of programs for standard situations without external faults.*

Definition 52 (Exception Specification) *The exception specification ES defines the functions of programs for exceptional situations with external faults.*

Accordingly, standard and exception behavior of programs can be distinguished.

Definition 53 (Standard Behavior) *The standard behavior constitutes what a program does to realize its functions as defined by the standard specification.*

Definition 54 (Exception Behavior) *The exception behavior constitutes what a program does to realize its functions as defined by the exception specification.*

The anticipated input domain AI that is associated with the specification S is further divided into a standard input domain $SI \subseteq AI$ and an exception input domain $EI \subseteq AI$ which are

Figure 3.4.: Extended Input Domain Structure Diagram

disjoint and complete, i.e. $SI \cap EI = \emptyset$ and $AI = SI \cup EI$. See Figure 3.4 (Page 40) for a structure diagram of the extended input domain that is based on the notation of UML 2.5.2 [Coo+17]. Since the two domains are disjoint, the standard and exception specification cannot specify expected outputs for the same input.

Definition 55 (Standard Input Domain) *The standard input domain SI encompasses all inputs for which expected outputs are described by the standard specification, i.e. $\forall \tau \in SI, \exists o \in O$ such that $(\tau, o) \in STS$.*

Definition 56 (Exception Input Domain) *The exception input domain EI encompasses all inputs for which expected outputs are described by the exception specification, i.e. $\forall \tau \in EI, \exists o \in O$ such that $(\tau, o) \in ES$. The exception input domain covers inputs for all external faults that are anticipated by the exception specification.*

Further on, truly-valid and truly-invalid inputs can be distinguished according to whether they belong to the standard or exception input domain.

Definition 57 (Truly-Valid Input) *An input τ is truly-valid if and only if it belongs to the standard input domain $\tau \in SI$.*

Definition 58 (Truly-Invalid Input) *An input τ is truly-invalid if and only if it belongs to the exception input domain $\tau \in EI$.*

Consequently, a program is only dependable if it is consistent with its standard and exception specification. A program that is consistent with its standard specification is *correct* [PY08, p.45] and a program that is consistent with its exception specification is *robust*.

Definition 59 (Correct Program) *A program P is correct if and only if its behavior is consistent with its standard specification STS for the entire standard input domain SI, i.e. $\forall \tau \in SI, (\tau, P(\tau)) \in STS$.*

Definition 60 (Robust Program) *A program P is robust if and only if its behavior is consistent with its exception specification ES for the entire exception input domain EI, i.e. $\forall \tau \in EI, (\tau, P(\tau)) \in ES$.*

Please note, this is a narrow view on dependability that excludes further characteristics such as availability, reliability, and safety because of our focus on robustness. Further on, robustness is often understood as a gradual characteristic that encompasses different levels of robustness [Lig09, p.7]. Our definition of robust programs is not gradual but describes an absolute criterion because the distinction between different levels of robustness is not necessary here. Robustness should also not be confused with safety [SF13]. Safety is another characteristic of dependability that is typically a concern of critical programs such as in the avionics or medical that is concerned with the avoidance of catastrophic consequences [PY08, p.44]; [Avi+04]. Safety does not distinguish between truly-valid and truly-invalid inputs but defines safety requirements which are generally valid even in the presence of internal faults. In contrast, robustness is limited to truly-invalid inputs as it is concerned with the program's behavior in the presence of external faults. For truly-invalid inputs, robustness not only strives for the avoidance of catastrophic consequences. It may also strive for retaining partial or alternative functionality despite the presence of external faults [PY08, p.45].

Analogously to the distinction of correctness and robustness, ordinary and robustness faults and corresponding failures can be distinguished.

Definition 61 (Ordinary Fault) *An ordinary fault is an internal fault that is the identified or hypothesized cause of an ordinary failure.*

Definition 62 (Ordinary Failure) *An ordinary failure is a failure that occurs if the output of a program deviates from its standard specification.*

Definition 63 (Robustness Fault) *A robustness fault is an internal fault that enables an external fault to cause a failure of the program.*

Definition 64 (Robustness Failure) *A robustness failure is a failure that occurs if the output of a program deviates from its exception specification.*

Both truly-valid and truly-invalid inputs may be failure-causing inputs. To distinguish the causes of ordinary and robustness failures, failure-causing truly-valid inputs and failure-causing truly-invalid inputs are defined. A failure-causing truly-valid input causes ordinary failures and a failure-causing truly-invalid input causes robustness failures.

Definition 65 (Failure-Causing Truly-Valid Input) *A truly-valid input $\tau \in SI$ is a failure-causing truly-valid input if and only if the output of program $P(\tau)$ does not match the expected output of the standard specification, i.e. $(\tau, P(\tau)) \notin STS$.*

Definition 66 (Failure-Causing Truly-Invalid Input) *A truly-invalid input $\tau \in EI$ is a failure-causing truly-invalid input if and only if the output of program $P(\tau)$ does not match the expected output of the exception specification, i.e. $(\tau, P(\tau)) \notin ES$.*

Depending on the robustness fault, different types of robustness failures can be distinguished. Robustness faults can lead to security breaches when an attacker gains privileges by exploiting, for instance, a buffer overflow [BE11]. However, robustness faults are defined in a broad sense. They are not limited to internal faults that cause security breaches. Exploiting robustness faults can also cause programs to be unavailable because programs themselves or entire operating systems shut down [Koo+97]. Furthermore, robustness faults can remain silent when – despite the presence of an external fault – the provided response is incorrect but meets some characteristics of a correct response that might pass a superficial inspection. This type of failure can result in financial losses when, for instance, business rules or even laws are violated. As an example, a robustness fault can cause a program to accept a customer unintentionally reserving a car for the 29th of February in a non-leap year or a customer booking a flight with flipped begin and end dates. When these external faults remain unnoticed, downstream business problems are caused when the business is asked to fulfill the reservation or booking.

To make a program robust, it can be developed defensively [BF04]. *Defensive programming* is a principle that is based on mistrusting all influences that are external to the program's boundary [Mey92]; [BE11]; [MSB11]. External influences that are invalid or even dangerous must be either neutralized or rejected [TBL17]. Therefore, external faults must be anticipated and the program must detect errors in its state that are caused by external faults [BF04]; [Avi+04]. Once an error is detected, the hardware must execute computer instructions to recover the state by eliminating the error. If the program fails to recover, the absence or incorrectness of necessary computer instructions is a robustness fault.

In this context, *fail fast* is considered a good principle that strives to detect external faults and to propagate their occurrence as early and fast as possible such that the error in the state cannot become a robustness failure. SHORE [Sho04] provides the following argumentation in favor of failing fast: "failing immediately and visibly sounds like it would make your program more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production."

List of Symbols

STS	standard specification
$SI \subseteq AI$	standard input domain
$\tau \in SI$	truly-valid input
ES	exception specification
$EI \subset AI$	exception input domain
$\tau \in EI$	truly-invalid input

3.2.2. Exception Handling

Overview of Exception Handling

Exceptions and EH are concepts commonly used in defensive programming to develop robust programs [Goo75b]; [Gar+01]; [Mel+13]. Exceptions allow representing errors in the program's state and exception handling is used to detect errors in the state and to recover from them [DG94]; [MT97]; [LS98]; [Gar+01].

Both concepts are introduced to emphasize the input masking effect that is caused by EH and affects the fault detection ability of CT. The fundamental activities of EH are discussed in more detail to allow a finer-grained discussion of potential robustness fault. Further on, issues related to EH are presented to motivate the necessity of testing EH.

To realize the EH of a program, additional means are provided by the programming language, run-time environment, or the operating system which are summarized as exception handling mechanisms (EHMs). They are typically realized with features that are either inherent and embedded into programming languages, or they are added through libraries and system calls provided by the operating system [DG94]; [LS98]. The *return code* pattern [LS98]; [BDT06]; [BM00] requires each invoked operation to return a value to the invoking operation. Different values are either results in the context of standard behavior or codes that indicate exception occurrences. Together with `if-else` statements to separate computer instructions for EH, the return code pattern forms a simple EHM that solely relies on general purpose programming language features. Another example of a simple EHM is the *status flag* pattern where the invoking and invoked operation share a variable to indicate exception occurrences [BM00]. While made popular by the C programming language and UNIX operating system, the return code and status flag patterns are considered to have many major drawbacks [LS98]; [BM00].

EHM as inherent and embedded features can be traced back to programming languages like PL/I [CC16] and LISP 1.5 [JG93] in the 1960s or ML [MTH90] in the 1970s [ECS15]. GOOD-ENOUGH [Goo75b]; [Goo75c] first proposed a notation for EH in 1975. His work is commonly acknowledged as being the foundation of most research on EH [Knu87]; [RM03]; [CM07].

GOODENOUGH [Goo75b]; [Goo75c] provided a broad definition of *exceptions* as "those [conditions] brought to the attention of the operation's invoker". In addition to the recovery from errors caused by external faults, the definition also includes cases where exceptions are used for valid results and to monitor the progress of operations. Therefore, exceptions can be considered as a more general concept that is not limited to recovery of external faults. However, follow-up work argued against using exceptions for valid results and monitoring purposes [Cri82]. It is also argued not to use exceptions to recover from internal faults because they should be removed prior to execution [MR77]. We follow this argumentation and understand exceptions and related concepts as a means to deal with external faults.

Definition 67 (Exception) *An exception is a characterization of abnormal or unexpected conditions that can appear in a program's state and may require extraordinary computation [YB85]; [Knu87]; [BG18].*

Definition 68 (Exception Occurrence) *An exception occurrence denotes the presence of an exception in a program's state [Cri89].*

Definition 69 (Exception Condition) *An exception condition is a predicate and its satisfaction implies the occurrence of an exception in a program's state [YB85].*

The invocation of programs with truly-invalid inputs from the exception input domain EI implies external faults and the occurrence of one or more exceptions. However, the occurrence of an exception does not imply that the exception is detected and that a program's state is recovered. To be consistent with its exception specification, a program must detect exception occurrences and it must execute appropriate computer instructions to recover the state. Otherwise, it might fail to deliver outputs as expected by its standard and exception specifications. The absence of correct computer instructions for detection and recovery can be a robustness fault that can, in turn, cause robustness failures.

Definition 70 (Exception Handling) *EH is an umbrella term that encompasses all activities of detecting exception occurrences and providing appropriate responses for recovery [Bai99].*

When developing EH, two general strategies to deal with exception occurrences can be distinguished [Li+18]. The *internal exception removal strategy* attempts to recover from exception occurrences locally. For instance, invalid inputs can be neutralized, e.g. fixing the format of a date or floating-point number, requests to third-party programs that did not respond can be retried, or the occurrence can be logged and ignored if appropriate. Afterwards, the program is recovered and the normal mode of operation may continue.

If the program cannot recover from exception occurrences locally, the exception occurrences must be propagated to the invoker. We denote this as the *external exception propagation strategy* which often follows the fail fast principle. When the occurrence of an exception is detected and propagated, the normal mode of operation cannot continue and the invoker has to provide an appropriate reaction. Depending on the specific exception, the invoker may try to execute the program again, execute an alternative, or propagate the exception occurrence even further and back to its invoker.

From the very beginning, the objective of EH and EHMs is to improve the organization of computer instructions. In addition to the notation of EH, GOODENOUGH [Goo75a] also discussed issues and objectives related to the design of EHMs. According to GOODENOUGH, EHMs should be designed such that robustness faults are "avoided, discouraged, or detected at compile-time" [Goo75a]. Moreover, he emphasizes the importance of comprehensibility of EH designs which "is enhanced by reducing the complexity" and by fostering a "clear control-flow structure". He considers reusability as another aspect to support comprehensibility and distinguishes between the reuse of computer instructions to recover from the same type of exception at different locations in the source code and the reuse of computer instructions to recover from different types of exceptions at the same location in the source code. Finally, efficiency is mentioned as another important aspect to "avoid unacceptable run-time inefficiencies".

Since then, EH and EHM are actively researched and address the points made by GOODENOUGH [Goo75a]. Advances are, for instance, published in dedicated issues of the IEEE Transactions on Software Engineering (TSE) journal [PRT00b]; [PRT00a] and the Springer Lecture Notes in Computer Science (LNCS) [Rom+01]; [Don+06] book series. Most research has been devoted to the design of EHMs for experimental and stable programming languages that follow

the paradigms of imperative, object-oriented, or functional programming [BAZ00]. There are even proposals for alternative EHM designs for existing programming languages like C [Lee83]; [Geh92] or Java [GH05]; [Sil13]; [KSS17]. Furthermore, a variety of additional research on alternative EHM designs exists. For instance, aspect-orientation is researched as an alternative to improve reusability [LL00]; [Fil+06]; [Fil+09]; [Alm+14].

Language features of EHMs should be designed such that they avoid or discourage robustness faults or such that they at least allow the detection of robustness faults at compile-time. The language features should be understandable and reduce the complexity of EH. To improve the organization of computer instructions, EHMs follow the *separation of concerns* design principle and separate the computer instructions dedicated to the standard behavior of a program from those dedicated to the exception behavior [Fil+09]; [Ber+08]. However, EHMs are criticized for bringing only limited advantages if not bringing disadvantages [Bla82]; [MT97]; [LL00]; [Cac+08]. For instance, MILLER & TRIPATHI [MT97] argue "that EH may have different requirements than object-orientation" and "that the differences are more like conflicts because EH can contradict the conventional object-oriented paradigm". They present *unintended handler action* as a very common and harmful issue of EHMs where not the intended but another unintended exception handling is invoked as a response to a detected exception.

Further on, empirical studies show that EHMs are often not adequately applied and itself a significant cause of robustness failures (cf. [SAW12]; [BGB14]; [Cac+14]; [SCK16]; [Oli+16]; [Bar+16]; [Mon+18]). Investigations to better understand the causes of robustness failures are still subject of research. Developers tend to focus on the normal behavior of programs while oversimplifying [CM07]; [Jak+15]; [BG18] or even neglecting the exception behavior [SGH10]; [Ber+11]. For instance, SHAH et al. [SGH10] interviewed novice and expert software developers about their experience with EH. Novice software developers often adopt an *ignore-for-now* approach when confronted with EH until they are forced to address EH because, for instance, a robustness failure occurred. In contrast, expert software developers seem to acknowledge the importance of EH and have trained themselves to incorporate EH into the software development. The main usage pattern of EH is to propagate exceptions to the user in order to report an external fault, e.g. invalid input and to ask for valid input. Although, developing the standard and exception behaviors of a program simultaneously are perceived as conflicting goals because EH is "not the use case that was identified" and "[EH] tends to get complex because of the number of things that can happen all at once".

BARBOSA & GARCIA [BG18] describe EH as the "least understood part of software projects" and SAWADPONG & ALLEN [SA16] emphasize the inherent complexity of EH: "Intuitively, humans tend to make mistakes when dealing with complex tasks. This applies to programming tasks; mistakes are more likely to occur where the code is difficult to comprehend as is often the case with EH code. Exceptions may be handled locally within a method or [propagated] to [an exception handler] elsewhere in the call chain, further adding to the complexity of the [program]." In fact, EH is often not locally limited to sub-programs. Instead, it globally affects entire programs [Fil+09]. Due to dedicated language features of EHMs, an *implicit* control flow for exceptions is established which is detached from the standard control flow and complicates EH even more [RM00]; [Cac+08]; [WN08].

To support developers, static analysis tools to analyze and visualize the implicit flow of excep-

Figure 3.5.: Fundamental Activities of Exception Handling

tions from sources of exception propagation to target exception handlers are developed. Static analysis allows identifying paths from exception occurrences and to potential responses for recovery. Based on those paths, it is searched for exceptions that have no corresponding recovery instructions or exceptions that might lead to unintended handler actions (cf. [RM99]; [RM03]; [Jo+04]; [BDT06]; [FR07]; [WN08]; [Coe+08a]; [Cac+09]; [SLM12]; [MCK12]; [Mel+13]; [Jin+17]; [Oli+18]). However, due to the nature of static analysis, usually many false positives are reported [Ber+11]; [CC16]. Therefore, we focus on testing EH to identify robustness faults.

Subsequently, the fundamental activities of EH are discussed. They are a prerequisite for a discussion of issues that are related to EH and current EHMs.

Fundamental Activities of Exception Handling

Again, an exception characterizes abnormal or unexpected conditions that can appear in a program's state. Some abnormal or unexpected conditions are predefined by the hardware, the operating system, or by the run-time environment of a program [LS98]. Examples are attempts to divide by zero or to access array elements with indices that are out of bounds. Besides these *predefined exceptions*, another class of *program-defined exceptions* exists [LS98]. Program-defined exceptions are defined on the program-level and are specific for the program.

To better describe robustness faults and failures, we further structure EH into five fundamental activities. The fundamental activities and their ordering are depicted in Figure 3.5.

Exception Detection   Exception detection is the first activity of EH to detect the occurrence of an exception [DG94]. For predefined exceptions, the hardware, operating system, or run-time environment implicitly performs reliability checks to detect exception occurrences [Gar+01]. For program-defined exceptions, the developers must explicitly define conditions and computer instructions to detect exception occurrences.

After an exception is detected, appropriate computer instructions for the recovery must be executed. Sometimes, the part of a program in which the exception occurrence is detected can recover itself from an exception occurrence. For instance, some invalid inputs can be neutralized by removing illegal characters from it. This is the aforementioned *internal exception removal strategy* and exception detection is directly followed by the exception recovery as indicated by a dashed line in Figure 3.5. However, the significance of an exception and appropriate computer instructions for recovery are often only known outside the part of a program that detected the exception occurrence [Goo75b]. This is the aforementioned *external error propagation strategy*.

Exception Propagation   Exception propagation is the second activity of EH where the occurrence of an exception is propagated to request the recovery of the program's state [Bai99]. Occurrences of predefined exceptions are implicitly propagated from the hardware, operating system, or run-time environment [DG94]. For program-defined exceptions, the developers must explicitly define computer instructions to propagate the exception occurrences.

The exception propagation requires an explicit *exception representation* that allows attaching further information to distinguish different types of exceptions and to provide additional details that can be utilized during exception recovery [DG94]; [LS98]; [Bai99]; [Gar+01]. Depending on the programming language and run-time environment, an exception occurrence can be represented by a symbol, a data object, or a full object [Gar+01] with a singular, hierarchical or object structure [LS98].

Following the aforementioned return code pattern or the status flag pattern, where the invoking and invoked operation share a variable to indicate exception occurrences [BM00], general-purpose value types like integers or strings are used to represent exception occurrences. More sophisticated EHMs rely on dedicated types for exception representation. Object-oriented EHMs typically have exception class hierarchies and new exception classes can be defined by derivation from existing exception classes [BM00].

When exception occurrences are detected and all further information are collected, the exception occurrence is either implicitly propagated for predefined exceptions or explicitly propagated for program-defined exceptions. In the latter case, programming language features like `return`, `throw`, or `raise` can be used to propagate an exception occurrence [BM00].

Exception Handler Selection   Exception handler selection is the third activity of EH to search and select one appropriate exception handler for a propagated exception occurrence [BM00]. As a consequence of exception propagation, a different part of a program becomes responsible for the recovery activities. Therefore, two central elements can be distinguished: exception handler and guarded region [DG94]. An *exception handler* is a sequence of computer instructions that is devoted to recover from exception occurrences. A *guarded region* is a sequence of computer instructions that is 'protected" by exception handlers such that at least one exception handler is executed whenever an exception occurs within the protected region. The granularity of a guarded region can range from complete programs to procedures, statements, and expressions [DG94]; [LS98].

In order to determine an appropriate exception handler, guarded regions and exceptions are associated with exception handlers. The association is either static or dynamic [LS98]; [BM00]. The static exception handler association is a permanent binding for an exception within a protected region that is fixed at compile-time where the same exception handler is used for every propagated occurrence of the exception [LS98]. The dynamic exception handler association is not permanent and cannot be determined at compile-time because it depends on run-time conditions such as the path that the execution has taken [DG94]. The appropriate exception handler is determined at run-time by searching for exception handlers along the execution path [BM00]. Conceptually, a *default* exception handler exists which is selected as a last choice if no other appropriate exception handler is found [Bai99]; [BM00]. The dynamic exception handler association is adopted by most EHMs [LS98]; [BM00].

While the dynamic exception handler association offers a great flexibility, it can make it difficult to understand which exception handler will be selected for recovery [BM00]. Therefore, some programming languages strive for simplification and introduce *exception lists* where methods, functions, or similar source code regions must explicitly list all the exceptions that may be propagated.

Exception Recovery   Exception recovery is the fourth activity of EH where the program's state is recovered from a propagated exception occurrence once an appropriate exception handler has been determined. Therefore, the exception representation of the propagated exception occurrence and its attached information are passed to the selected exception handler and the exception handler is executed.

When the exception handler is successfully executed, i.e. if no exception occurrence is propagated, the exception occurrence is removed and the program's state is considered to be recovered. This case is followed by the next activity of exception continuation.

But an exception handler may also propagate exception occurrences [BM00]. Then, another exception handler must be selected as indicated by the dashed line from exception recovery to exception propagation in Figure 3.5.

Another propagation may be necessary because, for instance, another exception occurred during the recovery or the selected exception handler cannot recover from the original exception occurrence.

Exception Continuation   Exception continuation is the last activity of EH. After an exception handler has been determined and successfully executed, the normal execution should continue [Gar+01]. In general, *resumption* and *termination* are the two main models for continuation [DG94]. Although, further models can be identified [BM00]. The termination model is the most popular model among current programming languages [BM00].

Following the resumption model, the normal execution resumes in the protected region in which the exception occurred and after the computer instruction that propagated the exception [Gar+01]. Following the termination model, the protected region in which the exception occurrence was detected, propagated, and recovered by the successful execution of an exception handler is terminated and cannot be resumed [Gar+01].

Garcia et al. [Gar+01] distinguish three variations of the termination model: return, retry, and strict termination. The *return* variation terminates the protected region and the normal control-flow continues after the protected region and after the successfully executed exception handler. The *retry* variation can be understood as a special case of the return variation. Once the normal activities can continue, the protected region that was terminated is executed again. The *strict termination* variation does not only terminate the protected region but terminates the whole program and the control is returned to the run-time environment or operating system.

Summary of Key Concepts   To summarize the concepts and fundamental activities of EHMs, Figure 3.6 depicts an example of EH in the Java programming language. The guarded region is defined by the keyword `try` and different exception handlers are defined by the keyword `catch`. The immediate succession of `try` and `catch` associates the exception handlers

Figure 3.6.: Exception Handling in Java

with the guarded region and the exception types within parentheses bind an exception handler to specific exceptions.

The `readInput` method is a simplification to receive user inputs and convert them into integers. When zero is entered as the divisor, an exception occurrence takes place after the second execution of the `readInput` method finished. In this case, the exception condition is explicitly defined by `divisior == 0` and `ArithmeticException` is used as the corresponding exception representation.

The first activity of exception detection is realized using an `if` statement and the `throw` keyword is used to initiate exception propagation. The third activity of exception handler selection is done implicitly by the Java run-time environment. After the exception handler that is bound to the `ArithmeticException` has been selected, the exception recovery is initiated by passing the exception representation and executing the exception handler. Afterwards, the exception continuation resumes the normal activities after the `catch` statements. Further on, the exception list of the `readInput` method defines three different exceptions that the method may propagate. Although, this list is not necessarily exhaustive because Java only requires to

list *checked* exceptions. For instance, `IndexOutOfBoundsException` is not required to be listed. When no appropriate exception handler is found, the default exception handler is implicitly selected and terminates program.

3.2.3. Combinatorial Robustness Testing

In the previous subsections, concepts related to correct and robust programs are defined and details of EH are presented. Since these concepts are not considered by the CT test method, they are now transferred and the CRT test method is introduced. Afterwards, a summary of the key concepts is provided that points out the relationship between the ground truth concepts relative to the specification and the concepts of CRT.

Foundations of Combinatorial Robustness Testing

The invocation of a program with truly-invalid inputs from the exception input domain EI implies the occurrence of one or more external faults. The occurrence does not imply that the external fault is detected and that the program's state is recovered. But the occurrence implies that a program functions as defined by the exception specification. To ensure that both the standard and exception behaviors are checked, testing for correctness and robustness must be distinguished.

Definition 71 (Functional Testing) *Functional testing is a particular type of testing that checks whether the SUT is consistent with its standard specification STS [IEE90b].*

Definition 72 (Robustness Testing) *Robustness testing is a particular type of testing that checks whether the SUT is consistent with its exception specification ES [IEE90b]; [Mic+12].*

To conduct functional and robustness testing, not only truly-relevant inputs must be distinguished from truly-irrelevant inputs. But truly-valid inputs must also be distinguished from truly-invalid inputs. Therefore, the concepts are transferred to truly-valid and truly-invalid schemata. These concepts are refinements of Definition 57 (Truly-Valid Input) and Definition 58 (Truly-Invalid Input) that incorporate the structure of parameters and values. A truly-valid input is also a truly-valid schema and a truly-invalid input is also a truly-invalid schema.

Definition 73 (Truly-Valid Schema) *A truly-relevant schema s is truly-valid if and only if at least one truly-valid input $\tau \in SI$ exist that covers $s \subseteq \tau$.*

Definition 74 (Truly-Invalid Schema) *A truly-relevant schema s is truly-invalid if and only if no truly-valid input exists $\tau \notin SI$ exist that covers $s \subseteq \tau$, i.e. $\nexists \tau \in SI \land s \subseteq \tau$. Consequently, at least one truly-invalid test input exists that covers a truly-invalid schema s, i.e. $\exists \tau \in EI \land s \subseteq \tau$.*

Two special cases of truly-invalid schemata play a primary role work.

Definition 75 (Truly-Minimal Invalid Schema) *A truly-invalid schema s is a truly-minimal invalid schema if and only if no proper subset schema $s' \subset s$ exists that is itself a truly-invalid schema. In other words, each proper subset schema $s' \subset s$ is a truly-valid schema and a truly-valid input $\tau \in SI$ exists that covers s'.*

Definition 76 (Truly-Strong Invalid Schema) *A truly-invalid schema τ is a truly-strong invalid schema if and only if it covers exactly one truly-minimal invalid schema s_a, i.e. for all pairs of two truly-minimal invalid schemata (s_a, s_b) with $s_a \neq s_b$, $s_a \subseteq \tau \Rightarrow s_b \not\subseteq \tau$.*

By definition, a truly-minimal invalid schema is always a truly-strong invalid schema because it only covers itself and no other invalid schema. But, a truly-strong invalid schema is not necessarily a truly-minimal invalid schema since it can be expanded to a test input with a degree of $d = n$ that still covers exactly one minimal truly-invalid schema.

> **Running Example** For the running example, the schemata `[Packages:1]` or `[Country:UK, Phone:+44]` are truly-valid.
> In contrast, the schemata `[Country:UK, Phone:123]` and `[Country:UK, Phone:+1]` are truly-invalid. Both schemata are strong truly-invalid schemata because each one covers exactly one truly-minimal invalid schema. Although, only the second schema is a truly-minimal invalid schema because the first schema covers the truly-minimal invalid schema `[Phone:123]`.

Analogously to truly-valid and truly-invalid inputs, truly-valid schemata may cause ordinary failures and truly-invalid may cause schemata robustness failures.

Definition 77 (Failure-Causing Truly-Valid Schema) *A truly-valid schema s is a failure-causing truly-valid schema if and only if each truly-valid input $\tau \in SI$ that covers $s \subseteq \tau$ is a failure-causing truly-valid input.*

Definition 78 (Failure-Causing Truly-Invalid Schema) *A truly-invalid schema is a failure-causing truly-invalid schema if and only if each truly-strong invalid input $\tau \in EI$ that covers $s \subseteq \tau$ is a failure-causing truly-invalid input.*

The concepts can be further refined to minimal failure-causing truly-valid schemata and minimal failure-causing truly-invalid schemata. The definitions are analogously to the definition of minimal failure-causing schemata.

Definition 79 (Minimal Failure-Causing Truly-Valid Schema) *A failure-causing truly-valid schema s is a minimal failure-causing truly-valid schema if and only if no proper subset schema $s' \subset s$ exists that is itself a failure-causing schema.*

Definition 80 (Minimal Failure-Causing Truly-Invalid Schema) *A failure-causing truly-invalid schema s is a minimal failure-causing truly-invalid schema if and only if no proper subset schema $s' \subset s$ exists that is itself a failure-causing schema.*

It is important to note that the property of minimality is related to the concept of causing failures. It should not be confused with the property of minimality that is related to invalidity. A minimal failure-causing truly-invalid schema is a truly-invalid schema but not necessarily a truly-strong invalid schema or a truly-minimal invalid schema. That distinction will be examined more closely in Section 5.4 (Page 105 and following).

Truly-irrelevant schemata are not the only cause of the input masking effect. The same phenomenon can be observed with truly-invalid inputs which is neglected by most research on CT. SUTs that are developed defensively follow the external exception propagation strategy and initiate EH when the first truly-invalid schema of an input is detected. The SUTs switch to exception behavior and the standard behavior is not resumed. As a consequence, all other schemata of the test input are masked since the standard behavior for them remains untested.

Within the research on CT, the invalid input masking effect is recognized (cf. [She94]; [Coh+94]; [Coh97]; [GOA05]; [Cze06]; [WT14]), but not yet conclusively researched. For further research, we denote CRT as an extension to CT.

Definition 81 (Combinatorial Robustness Testing) *Combinatorial robustness testing (CRT) is a test method that is based on extended IPMs with additional semantic information to distinguish valid from invalid schemata. CRT builds upon test adequacy criteria and test selection strategies that incorporate the additional semantic information. The objective of CRT is to improve the fault detection by avoiding the invalid input masking effect. Therefore, valid and invalid test inputs are separated such that all values and value combinations are tested as expected.*

Existing approaches and our new approach to define valid and invalid schemata with regards to the IPM are discussed in Chapter 4 on related work and Chapter 6 on the robustness input parameter model (RIPM). Hence, a definition cannot be provided in this section.

Summary of Key Concepts

An overview of key concepts of correct and robust programs is provided in Figure 3.7. It depicts a structure diagram that is again based on the notation of UML 2.5.1 [Coo+17]. It consists of named elements, aggregations, and generalization sets (cf. [Coo+17, p.119]). For reasons of clarity, names of generalization sets and cardinalities are omitted. Further on, each generalization set is considered disjoint. The concepts that are taken over from the taxonomy of dependable programs (Figure 3.3, Page 37) are indicated by a gray background.

The diagram is again divided into four quadrants which are pictured by dashed circles and correspond to four different views on the central concept of input $\tau \in I$ and its input domain I.

The upper left quadrant no. 1 depicts the concepts that are established by the authoritative specification. Truly-relevant inputs $\tau \in AI$ are further separated into truly-valid $\tau \in SI$ and truly-invalid inputs $\tau \in EI$ according to the distinct standard specification STS and exception specification ES. In addition, truly-strong invalid inputs denote a special type of truly-invalid inputs which cover exactly one truly-minimal strong invalid schema. Since the concept requires schemata for its definition, it is only listed here for the sake of completeness.

Analogously, failure-causing inputs are further separated into failure-causing truly-valid and failure-causing truly-invalid inputs.

In the upper right quadrant no. 2, the concepts are transferred to the structure of parameters and values of IPMs. Truly-relevant schemata are separated into truly-valid schemata s that are covered by at least one truly-relevant input, i.e. $\exists \tau \in AI \wedge s \subseteq \tau$, and truly-invalid schema s that are not covered by truly-valid but covered by truly-invalid inputs, i.e. $\nexists \tau \in SI \wedge s \subseteq \tau$ and $\exists \tau \in EI \wedge s \subseteq \tau$.

The finer-grained structure of parameters provides more details about truly-invalid schemata. Therefore, truly-minimal invalid schema and truly-strong invalid schema are introduced as refinements. A truly-minimal invalid schema s is a truly-invalid schema with the additional property that no proper subset schema $s' \subset s$ is a truly-invalid schema. A truly-strong invalid schema is a truly-invalid schema that covers exactly one truly-minimal invalid schema.

Furthermore, the failure-causing schemata are further separated into failure-causing truly-valid schemata and failure-causing truly-invalid schemata. The classification depends on whether the failure-causing schema is truly-valid or truly-invalid.

Analogously, minimal failure-causing truly-valid schemata and minimal failure-causing truly-invalid schemata are refined concepts of minimal failure-causing schemata that again utilize the finer-grained structure of parameters to provide more details about the cause of a failure.

Quadrant no 1 and 2 depict the concepts that are necessary to discuss testing in the concept of EH and the invalid input masking effect. The authoritative specification is used to define them. Although when testing, not the authoritative specification but an IPM is used to select test inputs. Therefore, counterparts of the concepts are necessary and they must be described by the structure of an IPM.

Therefore, quadrant no. 3 and no. 4 depict a partitioning of test inputs and schemata that is based on the structure of an extended IPM. Since the structure is developed throughout this work, both lower quadrants do not depict a summary of previously described concepts. Instead, they depict concepts that are defined in Chapter 6 (Page 111 and following).

The following is therefore a summary of Chapter 6. It is integrated in this comprehensive reference to support the understanding of the relationships. Since a test input is a special schema with $d = n$, only the lower right quadrant no. 4 is explained.

Besides the exclusion-constraints that provide a means to separate relevant from irrelevant schemata, error-constraints are introduced to separate valid from invalid schemata.

Based on the error-constraints, relevant schemata s, i.e. $\texttt{SAT}(C^{ex}, s) = true$, are separated into valid schemata s, i.e. $\texttt{SAT}(C^{ex} \cup C^{err}, s) = true$, and invalid schemata s, i.e. $\texttt{SAT}(C^{ex}) = true \wedge \texttt{SAT}(C^{err}, s) = false$.

Furthermore, strong invalid schemata and minimal invalid schemata are the counterparts of truly-strong schemata and truly-minimal schemata. An invalid schema s is strong invalid if exactly one error-constraint remains unsatisfied, i.e. $\exists!c \in C^{err}, \texttt{SAT}(C^{ex} \cup C^{err}) = false \wedge \texttt{SAT}(C^{ex} \cup C^{err} \backslash \{c\}) = true$. Further on, a strong invalid schema s is a minimal invalid schema if and only if no proper subset $s' \subset s$ is an invalid schema.

Accordingly, the counterparts of failure-causing truly-valid schemata and failure-causing truly-invalid schemata are defined as follows. A failure-causing valid schema is a valid schema that is failure-causing and a failure-causing invalid schema is an invalid schema that is failure-causing. Further on, a failure-causing valid schema or a failure-causing valid schema is minimal if no proper subset is a failure-causing schema.

In the following, we review existing approaches to CRT, argue why additional research for CRT is necessary and highlight their shortcomings (Chapter 4, Page 55 and following). Afterwards, we present our CRT test method including the extended IPM structure.

Figure 3.7.: Key Concepts of Robust Programs and its Relations

Chapter 4.

Related Work

Contents

Based on the previous introduction and problem description, we can distinguish CT as a general black box test method and CRT as an extension that focuses on special treatment of invalid schemata. But CRT is not the only test method that can be used to test for robustness. Therefore, we distinguish combinatorial and non-combinatorial robustness testing. Although the focus is on CT, we also discuss non-combinatorial robustness testing afterwards.

4.1. Combinatorial Robustness Testing

CT can be viewed as an adaptation of design of experiments (DOE) which can be dated back to the 1920's where it was used to conduct experiments in agricultural science [KKL13, p.237]; [Kuh+15]; [Tzo19]. In the 1980's, methods of DOE were first used as test selection strategies for testing programs [Tzo19]. Since then, significant advances have been made to original methods and techniques that result in a plethora of different approaches to input parameter modeling, test adequacy criteria, and test selection strategies. To get an overview of CT and different research streams, we refer to existing surveys, e.g. [GOA05]; [NL11a]; [KL14]; [Ahm+17]; [Wu+19b]; [Wu+19a], summaries, e.g. [Yil+14b]; [Kuh+15]; [MPZ15]; [Tzo19], and books, e.g. [KKL13]; [ZZM14]; [TQ17].

In this section, we do not discuss all research streams of CT. First, we discuss general contributions to CT that are related to our research. Afterwards, we discuss CT test methods that are related to our research but that are not concerned with robustness testing. Finally, we discuss CRT test methods that are also concerned with robustness testing. The CT and CRT test methods are ordered chronologically.

4.1.1. General Contributions

Surveys and Case Studies   GRINDAL et al. [Gri+03]; [GOA05]; [Gri+06] surveyed and evaluated test selection strategies and related test adequacy criteria. Further on, the GRINDAL et al. considered the exclusion of irrelevant schemata and special treatment of invalid values. Therefore, they described the *t-wise valid coverage* which "requires every possible combination of valid values of t parameters be included in some test [input], and the rest of the values are valid". To satisfy t-wise valid coverage, all irrelevant schemata and all invalid schemata must be excluded, for instance, via exclusion-constraints. Thereby, the main drawback of t-wise coverage, i.e. irrelevant and invalid input masking, is mitigated.

As pointed out earlier, testing only valid schemata is not sufficient. Therefore, GRINDAL et al. also described *single error coverage* that is satisfied "if each [invalid] value of every parameter is included in some test [input] in which the rest of the values are valid". Since single error coverage requires exactly one invalid value per test input, the risk of invalid input masking between two or more invalid values is mitigated.

One case study conduced by WOJCIAK & TZOREF-BRILL [WT14] reports on applying CT that also includes invalid test inputs. They report that single error coverage was not sufficient because exception handling depended on interactions between invalid and valid values. In particular, "the same [exception] would often be handled differently depending on the firmware in control [...] or depending on the configuration of the system".

A further remark by WOJCIAK & TZOREF-BRILL is concerned with the ratio of valid versus invalid test inputs: "Since a lot of attention was given to [robustness] testing [...] where full recovery in the presence of [exceptions] was expected, the [test suite] contained a ratio of up to 2:1" invalid test inputs vs. valid test inputs. An evaluation on efficiency and early fault detection of CT conducted by PETKE et al. [Pet+13] came to the same conclusion when comparing the sizes of test suites that include invalid test inputs with test suites that do not.

In general, the aforementioned publications do often not clearly distinguish between relevance and validness which results in blurred lines between what should be excluded entirely and what should be tested separately.

We also did not find any discussions about the relationship between hypothesized robustness faults and the single error coverage or the t-wise interaction between invalid values and valid values. Hence, it is unclear if these test adequacy criteria are appropriate.

Further on, the invalid input masking effect is frequently mentioned in all aforementioned publications as an argument for separating valid and invalid test inputs. But only GRINDAL et al. [Gri+06] provide a deeper analysis and discussion. In their comparison of t-wise with base-choice (another CT test method that is explained subsequently), they experienced differences in the failure detection capability depending on ratio of valid to invalid values per parameter: "The pair-wise coverage [...] gives best results for test subjects where there are many valid values of each parameter, which is when [base-choice] is least effective. For test problems with many invalid values for each parameter, when [base-choice] is most effective, the pair-wise strategies may mask faults, that is, when tests cover several parameter value pairs with multiple invalid values, the effects of some parameters may be masked by others" [Gri+06].

While this example demonstrates the impact of the invalid input masking effect, a deeper analysis and discussion is missing. In fact, it is generally unknown to what degree t-wise coverage is

suitable in the presence of EH. For those reasons, we conduct an experiment to assess the fault detection effectiveness of CT in the presence of invalid values in Section 5.1.

Testing of Logical Expressions   Testing of logical expressions is an important topic and a lot of different test adequacy criteria such as decision or condition coverage exist [FG10]; [Gar11]. According to FRASER & GARGANTINI [FG10], the focus of new test methods is often "to detect specific classes of realistic faults and to produce as small as possible test suites".

Since CT is a generally-applicable test method, it can also be applied to the testing of logical expressions. BALLANCE et al. [BVJ12], VILKOMIR et al. [VSB13], VILKOMIR & ANDERSON [VA15] conduct experiments to investigate pair-wise and t-wise coverage. Their investigations reveal that CT can be effectively applied to test logical expressions. An overview of related work is provided by VILKOMIR [Vil16].

Moreover, WANG & QI [WQ15] analyze MFCSs to further understand why CT performs well in testing logical expressions and present a model to compute the lower boundary of fault detection probability for CT. Later, YU et al. [Yu+18] revisit the model to compute the lower boundary of fault detection probability and publish a counterexample to illustrate a fault in the model. In a follow-up study, XU et al. [Xu+16] analyze MFCSs of programs known as the Siemens suite and report an observation that corresponds to an observation that we made when investigating the fault detection effectiveness (FDE) in the presence of invalid input masking (Page 89): "The high strengths of MFCSs makes it difficult to cover MFCSs by a lower-strength combinatorial test suite. But the big number of MFCSs for each fault makes that it is easy to cover at least one MFCS."

Our CRT test method also tests logical expressions, i.e. exception conditions that should initiate correct EH. Although we consider the invalid input masking effect since a schema that causes an exception condition to be satisfied cannot satisfy another subsequent exception condition.

Repairing Inconsistent Test Models   ARCAINI et al. [AGV14] research the detection of inconsistencies in IPMs. Therefore, they introduce desired properties of IPMs (consistency, completeness, minimality) and present algorithms to check whether IPMs adhere to these desired properties. Our approach for detection and repair of over-constrained RIPMs (Chapter 9, Page 159 and following) is a related idea as we also check for desired properties in RIPMs. Although, the work of ARCAINI et al. [AGV14] focuses on exclusion-constraints in IPMs. The desired properties of RIPMs are slightly different due to the different semantics of exclusion- and error-constraints. Further on, our approach reuses existing conflict diagnosis techniques and can also repair inconsistencies semi-automatically.

To the best of our knowledge, it is only the related work by PILL & WOTAWA [PW17]; [PW18]; [PW19] where conflict diagnosis techniques are also applied in the context of CT. Although their focus is substantially different. They combine CT and conflict diagnosis techniques to derive knowledge-bases that can be used for abductive diagnosis [PW17]; [PW18]. Further on, they use conflict diagnosis techniques to compute which components of a SUT explain the failure of test inputs [PW19].

GARGANTINI et al. [Gar+16] also research the detection of inconsistencies in IPMs. They focus on over-constrained exclusion-constraints that might prevent a truly-relevant schema from

being selected because the IPM classifies it as a irrelevant. Our approach for detection and repair of over-constrained RIPMs (Chapter 9, Page 159 and following) is related since we focus on over-restrictive error-constraints that prevent truly-invalid schemata from being selected. Further on, our and their approach assume that the parameters and values are correctly captured and focus on inconsistencies of constraints.

Although, their approach is not based on conflict diagnosis techniques but rather relies on test execution to check if exclusion-constraints of an IPM conform to the SUT. Therefore, six policies are proposed that cause a test selection strategy to purposely select test inputs that violate some exclusion-constraints. If a test input that purposely violates some exclusion-constraints yields acceptable behavior of the SUT, a conformance fault is identified where the IPM is considered over-constrained. In addition, if a test input that does not violate any exclusion-constraints does not yield acceptable behavior of the SUT, a conformance fault is identified where the IPM is considered under-constrained [GPR17].

GARGANTINI et al. [GPR17] extend their work on conformance faults and propose a repair process to automatically identify conformance faults and to automatically repair the IPM by either relaxing or strengthening exclusion-constraints. To automatically identify conformance faults and to repair the IPM, FC and fault characterization algorithms (FCAs) are utilized to compute a MFCS that represents the conformance fault and the MFCS is used to relax or strengthen some exclusion-constraints.

As already stated before, we rely on conflict diagnosis techniques to detect and repair inconsistencies in RIPMs that do not require any test execution. But we adopt and extend the idea of finding conformance faults based on test execution and aligning the RIPM with the program using FCAs to identify invalid schemata and to align RIPMs with the EH of programs (Chapter 10, Page 187 and following).

4.1.2. CT Test Methods

Base-Choice   Base-choice was originally published by AMMANN & OFFUTT [AO94] as an alternative to each-choice. Base-choice does not consider exclusion-constraints but values are annotated with additional semantic information to define a *base* test input with default values for each parameter. Test inputs are created by replacing one parameter value of the base test input at a time.

Although not mentioned in the initial publication [AO94], GRINDAL et al. show that base-choice can subsume single error coverage, i.e. a test suite that satisfies base-choice also satisfies single error coverage [Gri+03]; [GOA05]. As long as the base test input is valid, the value domains of some parameters may contain invalid values and the base-choice test selection strategy selects invalid test inputs according to single error coverage.

Even though it was not discussed in the initial publication nor in the survey or evaluation by GRINDAL et al. [Gri+03]; [GOA05]; [Gri+06], base-choice also has limited support for invalid value combinations: Since one parameter value of the base test input is replaced at a time, an invalid value combination can be created as a combination of a replaced value and other remaining values of the base test input.

The failure detection capability of base-choice highly depends on the quality of the base test input [Gri+03]; [GOA05]; [Gri+06]. In contrast, failure detection capability of t-wise coverage

only depends on the chosen testing strength [Tzo19]. Therefore, we focus on t-wise coverage and advancements to mitigate the various types of input masking effect.

Shielding Parameters CHEN et al. [CYZ10] propose another extension to CT. They introduce so-called shielding parameters to target another kind of input masking. In general, parameters in CT are assumed to be *effective* all the time. But some specific parameter values may *disable* and therefore mask potential effects of other parameter values. For instance, when the help option -h of a command-line program is used, all other options are disabled.

When a parameter value $v_1 \in p_1$ is defined to disable another parameter p_2, a *vain* value # is implicitly defined and always selected for p_2. Thereby, no schema that covers an ineffective interaction of p_2 and some other parameters is selected.

SEGALL et al. [STZ12] describe shielding parameters as a common pattern "conditionally-excluded values" and discuss how to model it using *traditional* CT test methods. Therefore, vain values must be explicitly modeled in the IPM and exclusion-constraints like $p_1 = v_1 \Rightarrow p_2 = \#$ model the shielding. Further interactions between explicit vain values and other relevant values can be avoided via additional exclusion-constraints.

This extension can also be used for invalid values because if an invalid values is assigned to parameter p_1, parameter values that are evaluated after p_1 are disabled. However, this requires knowledge about the specific order of exception detection which we circumvent in our approach by relying on valid values for all other parameter values. Shielding parameters are also limited to single values and do not consider the shielding of value combinations. Further on, interactions between invalid schemata and other valid schemata can also not be tested because of the vain values.

Variable Strength Variable strength is a generalization of t-wise coverage that allows defining different testing strengths for subsets of parameters [Coh+03]; [YCP06]. Thereby, a *main* testing strength is defined as a lower boundary for all parameter interactions. For interactions between subsets of parameters, higher testing strengths can be defined.

When defining $t = 0$ as the main testing strength, variable strength coverage is very flexible and each interaction between parameters can be defined individually. Consequently, the parameter interactions that are defined by our combinatorial test adequacy criteria for strong invalid test inputs (Chapter 7, Page 119 and following) can also be defined in terms of variable strengths.

Of course, the computed interactions and schemata could then be transferred to a variable strength test selection strategy. A test selection strategy that supports variable strength coverage with $t = 0$ as the main testing strength can be used to select strong invalid test inputs. However, our family of test selection strategies allows including optimizations that cannot be realized in a generic variable strength test selection strategy. Variable strength coverage should also not be understood as a generalization that incorporates our work. The main concern of our test adequacy criteria and test selection strategies is the computation of parameter interactions and corresponding schemata that must be covered.

Existential-Constraints HALLÉ et al. [HCG15] propose a generalization of t-wise coverage and introduce *existential-constraints* as a particular type of constraints to express conditions

that have to be satisfied at least once. In contrast, exclusion-constraints are defined as "universal-constraints" that must be satisfied by each test input. Existential-constraints allow to express requirements such as "at least one test input should ..." or "the test suite should include a test input that ...". In their approach, one test input may satisfy more than one existential-constraint which allows to compose test inputs that satisfy several existential-constraint at once.

Our CRT test method follows a related idea that ensures for each error-constraint (another type of constraints) at least one test input to exist for which the error-constraint remains unsatisfied. The number of test inputs that exist for one error-constraint can be further adjusted by choosing different test adequacy criteria.

Existential-constraints can be converted to error-constraints by negating them. But to ensure that the test inputs are strong invalid, exactly one error-constraint remains unsatisfied for one test input. Therefore, more test inputs would be required because the satisfaction of several existential-constraints cannot be combined into one test input. Although the idea of HALLÉ et al. [HCG15] is related, it is unsuitable for selecting strong invalid test inputs because test inputs that satisfy several existential-constraints are not strong invalid.

4.1.3. CRT Test Methods

Constrained Array Test System (CATS) To the best of our knowledge, SHERWOOD [She94] first mentioned invalid values in the context of CATS which is a test selection strategy and tool for CT. SHERWOOD describes the effect of invalid input masking that is caused by exception handling and the external exception propagation strategy. But the discussion is restricted to invalid values and does not consider invalid value combinations. SHERWOOD further proposes two approaches with manual intervention to adjust the test selection in order to cope with invalid values. When the tester wants to ensure that each invalid value is covered by a test input, SHERWOOD suggests executing the test selection strategy with only valid values and to modify the test suite afterwards by replacing valid values with invalid values. When the tester also wants to test interactions between invalid values and valid values of other parameters, Sherwood suggests modifying the IPM such that exactly one parameter contains invalid values at a time and to execute the test selection strategy once for each modified IPM.

The two approaches with manual intervention may be ok from the perspective at the time given limited computer resources and less advanced conflict handling strategies. However, from today's perspective with more advanced computer resources and conflict handling strategies, the two approaches do not represent the state-of-the-art and repetitive human intervention should be avoided. But the IPM of CATS only consists of parameters and values, i.e. $IPM = \langle \tilde{P}, V \rangle$, and does not contain necessary semantic information for further automation.

The two approaches also resemble the *replace* and *sub-models* strategies for conflict handling as presented by GRINDAL & OFFUTT [GOM07]. The strategies are originally intended to remove conflicts from an IPM. But they can be used to add invalid values just as well.

Similarly, the third strategy *abstract parameter* could be used to cope with invalid value combinations. Following the original strategy, two or more original parameters of the IPM are merged into a new abstract parameter and the value domain of the abstract parameter is created by computing the cartesian product for the value domains of all merged parameters and by removing all composed values that are irrelevant or invalid. The abstract parameter could also

represent invalid value combinations when all irrelevant and valid composed values are removed instead. However, the strategy is not ideal since it also requires repetitive manual intervention.

Avoid as the fourth strategy for conflict handling allows to reduce repetitive manual intervention since the test selection strategies should automatically avoid the occurrence of unwanted schemata. Since it promises the highest degree of automation, we follow this approach. However, it requires semantic information such that the test selection strategy can distinguish relevant from irrelevant and valid from invalid schemata.

Automatic Efficient Test Generator (AETG)   COHEN et al. [Coh+94]; [Coh+97] published another test selection strategy and tool AETG for CT. They also acknowledge the necessity to separate valid and invalid test inputs and to ensure that one invalid test input only contains one invalid value. The IPM contains semantic information to distinguish relevant from irrelevant schemata and to distinguish valid from invalid values. Besides a set of parameters $\tilde{P}$ and a set of value domains V, the IPM, i.e. $IPM = \langle \tilde{P}, V, R \rangle$, contains a nonempty set of relations R to further specify relationships between subsets of parameters and their value domains. Each relation $r \in R$ is a tuple $r = \langle \tilde{P}^r, V^r, U \rangle$ that defines a set of value domains V^r for some parameters $\tilde{P}^r \subseteq \tilde{P}$ such that for each parameter $\forall p_i \in \tilde{P}^r$, a value sub-domain $V_i^r \subseteq V_i$ denotes the values that should be considered in the context of relation r. Further on, each relation has a *degree* which corresponds to the testing strength t. In addition, U is a set of "unallowed tests", i.e. predefined forbidden schemata, that must not be covered by any schema selected for that relation. To distinguish relevant from irrelevant schemata, either the set U can be used or multiple relations can be used to resemble the *sub-models* strategy of conflict handling. To distinguish valid from invalid values, the value domain V_i of each parameter $p_i \in \tilde{P}$ is separated into a valid value domain V_i^{valid} and an invalid value domain $V_i^{invalid}$. The two value domains are complete and disjoint subsets of the value domain, i.e. $V_i = V_i^{valid} \cup V_i^{invalid}$ and $V_i^{valid} \cap V_i^{invalid} = \emptyset$.

AETG selects schemata for one relation $r \in R$ at a time and merges the schemata afterwards so that their degree is $k = n$ and that they cover a value for each parameter $p \in \tilde{P}$. In order to select valid test inputs for one relation $r \in R$, all invalid values are excluded from the test selection strategy, i.e. $\forall V_i^r \in R, V_i^r \setminus V_i^{invalid}$. Invalid test inputs are also selected for one relation $r \in R$ at a time. For each invalid value described by relation $r \in R$, i.e. $\forall V_i^r \in r, V_i^r \cap V_i^{invalid}$, a valid test input of that relations is taken and a valid value is replaced by an invalid value.

In direct comparison to CATS, AETG also resembles the sub-model and replace strategies but automates their application using the semantic information of the IPM. However, AETG does not account for interactions between invalid and valid values. Although it can be modeled using relations, it would require the same manual interventions as with CATS.

Pairwise Independent Combinatorial Testing (PICT)   CZERWONKA [Cze06] published another test selection strategy and tool PICT. It supports propositional logic to describe exclusion-constraints and the IPM consists of parameters, values, and exclusion-constraints, i.e. $IPM = \langle \tilde{P}, V, C^{ex} \rangle$. Similar to AETG, the value domain V_i of each parameter $p_i \in \tilde{P}$ is separated into a valid value domain V_i^{valid} and an invalid value domain $V_i^{invalid}$ which are complete and disjoint subsets of V_i.

The test selection strategy is divided into two actual strategies. First, only the valid value do-mains are considered and `PICT` selects test inputs according to t-wise coverage while satisfying all exclusion-constraints. Afterwards, the invalid value domains are considered and additional exclusion-constraints are generated to ensure that no two invalid values are covered by the same test input. To account for interactions between invalid values and other valid values, `PICT` allows t-wise combining the invalid values with all valid values of other parameters.

Automated Combinatorial Testing for Software (ACTS)   YU et al. [Yu+13b] published yet another tool `ACTS` which implements the input parameter order (IPO) family of test selection strategies (cf. [LT98]; [Lei+07]; [Yu+13a]; [Yu+15]; [KS18]). The scientific publication about `ACTS` [Yu+13b] does not contain ideas that are concerned with invalid schemata. Their usage of the terms "invalid combination" and "constraint handling" refers to irrelevant schemata and exclusion-constraints. Although, the authors state that they "plan to support negative testing, where each test contains only one 'invalid' value. Negative testing is critical to ensure robustness of a system."

`ACTS` in version 3.0 contains the functionality and its user guide describes it accordingly. It is basically the same functionality as offered by `PICT`, i.e. a separation of valid and invalid values and the possibility of t-wise interaction among valid and invalid values. `ACTS` just provides a graphical user interface and different test selection strategies. Besides IPO-based test selection strategies, it also implements base-choice.

`CATS`, `AETG`, and `ACTS` do not even mention the selection of invalid test inputs that cover invalid value combinations. `PICT` also has no direct support to model invalid value combina-tions and to select invalid test inputs that cover those invalid value combinations. But the author of `PICT` [Cze06] mentions that the concept that integrates invalid values "can be extended from disallowing two values to disallowing any two or more combinations to coexist in one test case" and that "appropriately crafted [exclusion-]constraints" can be used to achieve the same results.

Although the author provides no additional information on how to extend the approach or how to craft the exclusion-constraints, invalid value combinations can in fact be integrated with the help of a workaround that relies on propositional logic. The workaround can be applied using `PICT`, `ACTS`, and all other tools that support the invalid values and exclusion-constraints based on propositional logic. It requires a new artificial parameter p_{err} to be added to the IPM. The artificial parameter has no actual counterpart in the SUT and its only purpose is to influence the test selection strategy. The actual values of the artificial parameter are ignored during the execution.

To model invalid value combinations, let the variables $\text{inv}_1, \text{inv}_2, \ldots$ represent logical con-ditions expressed in propositional logic that describe the invalid value combinations. The valid value domain $V_{err}^{valid} = \{\text{ok}\}$ consists of exactly one artificial valid value which will be covered by each valid test input. The invalid value domain $V_{err}^{invalid} = \{\text{err}_1, \text{err}_2, \ldots\}$ contains one artificial invalid value err_i for each invalid value combination inv_i that is to be modeled.

In addition, a *dedicated* exclusion-constraint c_{err_i} that follows the pattern $(p_{err} = \text{err}_i) \Leftrightarrow \text{inv}_i$ is added to C^{ex} for each invalid value combination inv_i. Since $p_{err} = \text{ok}$ is selected for all valid test inputs, the left-handed sides of these dedicated exclusion-constraints evaluate to false. Consequently, the right-handed side of all dedicated exclusion-constraints must evaluate

to false as well when the modeled invalid value combination is not covered. When $p_{err} = \text{err}_i$ is selected for an invalid test input, the right-handed side of exclusion-constraint c_{err_i} must evaluate to true while the right-handed side of all other dedicated exclusion-constraints must not evaluate to true.

> **Running Example** The value domains of all parameters are separated into two separate domains for valid and invalid values.
>
> $$\text{Packages: } \{1, 3\}^{valid} \cup \{123\}^{invalid}$$
> $$\text{Country: } \{\text{UK, USA}\}^{valid} \cup \{123\}^{invalid}$$
> $$\text{Phone: } \{+44, +1\}^{valid} \cup \{123\}^{invalid}$$
> $$\text{Shipping: } \{\text{Standard, Express}\}^{valid}$$
>
> In addition, an artificial parameter `Error` is introduced with an invalid value for each invalid value combination that is to be modeled.
>
> $$\text{Error: } \{\text{ok}\}^{valid} \cup \{\text{err}_1, \text{err}_2\}^{invalid}$$
>
> Two dedicated exclusion-constraints are added to model the invalid value combinations between `Country` and `Phone`.
>
> $$c_1 : \text{Error} = \text{err}_1 \Leftrightarrow (\text{Country} = \text{UK} \wedge \text{Phone} = +1)$$
> $$c_2 : \text{Error} = \text{err}_2 \Leftrightarrow (\text{Country} = \text{USA} \wedge \text{Phone} = +44)$$
>
> During the selection of valid test inputs, all invalid value domains are ignored and the `Error` is fixed to the `ok` value. During the selection of invalid test inputs, err_1 is selected for `Error` and as a consequence the invalid value combination is selected such that (`Country = UK` $\wedge$ `Phone = +1`) evaluates to true.

In general, the workaround is suitable to describe and select invalid value combinations. But the workaround increases the complexity of the IPM and makes mistakes more likely. It reduces the clarity and expressiveness since additional values must be used and constraints are required to match a certain pattern. *Real* exclusion-constraints are formulated such that schema τ is relevant when it satisfies the exclusion-constraints C, i.e. $\text{SAT}(C, \tau) = \text{true}$. When applying the workaround, however, the pattern $(p_{err} = \text{err}_i) \Leftrightarrow \text{inv}_i$ requires the right-handed side inv_i to be modeled the other way around. Hence, to define the characteristics of a valid τ, $\overline{\text{inv}_i}$ (the negation of inv_i) must evaluate to true. In practice, it becomes even more complex because $a \Leftrightarrow b$ is not supported by all user interfaces. Instead, the equivalent expression $a \Rightarrow b \wedge b \Rightarrow a$ must be used.

Besides the complexity of nested and negated conditions, the IPM with exclusion-constraints, separated value domains, and the workaround also has a conceptual flaw and weakness. From our point of view, there is no reason why invalid value combinations should be a second choice that is not considered by the algorithm designers and tool developers. Compared with invalid values, invalid value combinations are equally important.

Although invalid value combinations are not mentioned, one could argue that the workaround establishes a transitive relation between the artificial invalid values $V_{err}^{invalid} = \{\texttt{err}_1, \texttt{err}_2, ...\}$ and the invalid value combinations $\texttt{inv}_1, \texttt{inv}_2, ...$ that are modeled by the workaround. When each invalid value is covered by at least one invalid test input, all invalid value combinations are covered as well. Nevertheless, when a test method explicitly includes the concept invalid values, we believe it should also explicitly include the concept of invalid value combinations.

Further on, an explicit representation of invalid value combinations is necessary to define test adequacy criteria independently from any workarounds. Single error coverage requires all invalid values to appear at least once. When using it in conjunction with the workaround, it required all invalid value combinations to appear at least once. In Chapter 7 (Page 119 and following), we introduce test adequacy criteria that also allow to define combinatorics among the parameters of invalid value combinations. Then, not all invalid value combinations but all a-sized sub-schemata must appear at least once. Such test adequacy criteria require an explicit representation of invalid value combinations.

Beyond that, the missing logical separation of *real* exclusion-constraints and exclusion-constraints used to work around invalid value combinations complicates further techniques to analyze invalid value combinations and to support their modeling.

In this aspect, the aforementioned approaches differ from our solution that we present in this work. Our solution provides a generalization and provides an equal representation of invalid values and invalid value combinations. Further on, our solution provides a logical separation of exclusion-constraints and invalid value combinations to allow further analysis and support.

TestCover `TestCover`[1] is developed by SHERWOOD and attempts to increase the expressiveness of constraints by using general-purpose programming rather than propositional logic to specify them [She14]; [She15]; [She16]; [She17]. Therefore, `TestCover` uses *combination functions* [She15] to model functional dependencies among input parameters. In contrast to logic-based constraints which partition already defined values of parameters, combination functions are used to define values of a parameter in accordance to one or more other parameters. According to SHERWOOD [She15], a combination function represents values of a parameter p_i in a "functionally dependent form" that is converted into a value set V_i before a test selection strategy is applied. Although the details of the test selection strategy are not published, it seems that parameters, values, and combination functions are converted into conflict-free sub-models.

While combination functions are designed to exclude irrelevant schemata, a workaround to define invalid schemata exists that originates from an e-mail conversation with SHERWOOD.

Therefore, all relevant schemata are modeled as discussed before and for each parameter or combination of parameters that encompass invalid schemata, an auxiliary parameter is defined with a combination function that either takes `valid` or `invalid` as a value. In addition, another auxiliary parameter with a combination function is defined to ensure the composition of only strong invalid test inputs by counting the number of `invalid` values that are assigned to auxiliary parameters.

---

[1]See https://www.testcover.com (Last access on 1st December 2020).

Running Example The actual parameters and values are modeled as shown in Figure 2.1. In addition, the auxiliary parameter `Packages_Aux` is defined to model an invalid value. The corresponding combination function is depicted below. It is written in the PHP programming language which is also used by the `TestCover` tool.

```php
Packages_Fun($Packages) {
  if(strcmp($Packages, "123") == 0) return "invalid";
  else return "valid";
}
```

The `Packages_Fun` combination function is defined in accordance to the `Packages` input parameter. The dependency is explicated by the function parameter `$Packages` and all values of the parameter are passed to the combination function in order to compute the values. The function `strcmp()` compares two strings and returns 0 if they are equal. The string `invalid` is returned for invalid values. For valid values, the string `valid` is returned.

Further on, the auxiliary parameter `PhoneCountry_Aux` is defined to model individual invalid values for `Phone` and `Country` as well as to model combinations of them that are invalid.

```php
PhoneCountry_Fun($Country, $Phone) {
  $Errors = 0;

  if(strcmp($Country, "123") == 0) $Errors++;
  if(strcmp($Phone, "123") == 0) $Errors++;

  if(strcmp($Country, "UK")
    && strcmp($Phone, "+1") == 0) $Errors++;

  if(strcmp($Country, "USA")
    && strcmp($Phone, "+44") == 0) $Errors++;

  if($Errors == 0) return "valid";
  if($Errors == 1) return "invalid";
  else return;
}
```

The `Errors` variable is used to count the invalid schemata that are covered by each combination of `Country` and `Phone`. For each combination that does not cover any invalid schemata, the string `valid` is returned. For each combination that covers exactly one invalid schemata, the string `invalid` is returned. For each combination that covers two invalid schemata, e.g. the combination `[Country:123, Phone:123]`, no value is returned to mark the combination as irrelevant.

Further on, another auxiliary parameter `TestInput_Aux` and associated combination function is necessary to ensure only strong invalid test inputs.

```php
TestInput_Fun($Packages_Aux, $PhoneCountry_Aux) {
  $Errors = 0;
```

```
        if(strcmp($Packages_Aux, "invalid") == 0) $Errors++;
        if(strcmp($PhoneCountry_Aux, "invalid") == 0) $Errors++;

        if($Errors == 0) return "valid";
        else if($Errors == 1) return "invalid";
        else return;
    }
```

The combination function encompasses all auxiliary parameters and marks a test input as valid if the values all auxiliary parameters are valid. If a value of exactly one auxiliary parameter is invalid, the test input is marked as invalid. If more than one value is invalid, the test input is marked as irrelevant.

The t-wise coverage guarantees the inclusion of each schema with degree $d \leq t$. Therefore, each invalid value and each invalid value combination of size $d \leq t$ appears in at least one test input as well.

Since the auxiliary parameters are considered as well, at least one test input takes on the value `invalid` for each auxiliary parameter p^{aux}. If there is only one invalid value combination modeled by p^{aux}, then its inclusion is indirectly guaranteed by the one-to-one relationship – even for invalid schemata with degrees $d > t$. But, the guarantee does not hold if there is more than one invalid schema modeled by p^{aux} because there is no one-to-one relationship.

Running Example There is at least one test input that covers [Packages_-Aux:invalid]. Since only one invalid value is modeled by Packages_Fun, the test inputs that cover [Packages_Aux:invalid] always cover [Packages:123] as well.

There is also at least one test input that covers [PhoneCountry_-Aux:invalid]. Since three invalid schemata are modeled by PhoneCountry_Fun, the test inputs that cover [PhoneCountry_Aux:invalid] also cover [Country:123], [Phone:123], [Country:UK, Phone:+1], or [Country:USA, Phone:+44]. For t-wise coverage with $t = 1$, the coverage of [Country:123] and [Phone:123] is guaranteed and consequently [PhoneCountry_Aux:invalid] is also covered.

To ensure the coverage of larger invalid schemata, either the testing strength must be increased to $t = d$ or additional auxiliary parameters must be introduced such that each auxiliary parameter and combination function only model one invalid schemata.

The one-to-one relationship between an auxiliary parameter, a combination function, and one invalid schema also allows accounting for interactions between invalid schemata and other valid schemata. For t-wise coverage, the value `invalid` of each auxiliary parameter is combined with valid schemata for all parameter combinations of size $t - 1$.

However, the introduction of numerous auxiliary parameters makes the IPM more complex and confusing, might slow down the execution of test selection strategies, and the involved breakdown into numerous sub-models might result in test suites that are larger than necessary.

Further on, the combination functions ensure that invalid schemata only appear in strong invalid test inputs. But test adequacy criteria for valid and invalid test inputs are not separated. The invalid input masking effect still applies because a schema that is covered by an invalid test input is not guaranteed to appear in another valid test input. To separate valid and invalid test suites, the combination function `TestInput_Fun` must be adjusted such that all invalid schemata are marked as irrelevant for the selection of valid test suites and that all valid schemata are marked as irrelevant for the selection of invalid test suites.

Because all these concepts belong to a workaround, an automatic switch between test selection for valid and invalid test inputs does not exist. The mixed usage of combination functions to exclude irrelevant schemata and to distinguish valid from invalid schemata does not only complicate the IPMs but makes further automated support and analysis techniques impossible.

JCUnit   The `JCUnit`[2] tool is developed by UKAI & QU [UQ17]. `JCUnit` integrates the modeling of IPMs, the application of test selection strategies and the test execution into one Java-based framework that is build on top of the `JUnit`[3] framework for automated test execution. The IPMs are modeled as Java classes with constraints being modeled as ordinary Java methods that return boolean values. Further on, parameterized test methods encode the computer instructions to interact with the SUT. The parameterization allows using the same test methods for many different test inputs. One or more parameterized test methods can be defined and conditions can be attached to decide which test inputs are executed by which test method.

By default, the test selection strategy selects test inputs such that all constraints are satisfied. Thus, the selected test inputs are relevant and valid. In addition, "negative testing" can be enabled where invalid test inputs are selected such that they do violate constraints. Although the selection of invalid test inputs is not discussed in the publication [UQ17], further information is available in the online documentation[4][5].

To enable "negative testing", parameterized test methods are defined with conditions that require at least one constraint to remain unsatisfied. If exactly one constraint must remain unsatisfied, strong invalid test inputs for that constraint are executed. But the approach is more flexible than the aforementioned approaches and also allows combining several unsatisfied constraints. Moreover, it can also deal with invalid value combinations when they are modeled by constraints.

Based on the limited information available, it is unclear how many test inputs are selected from an unsatisfied constraint and if all t-sized schemata appear in at least one valid test input to prevent input masking. Further on, the constraints do not have an internal representation such as propositional logic or forbidden schemata, the Java methods that represent constraints are executed many times during the test input selection. To improve the efficiency, the IPM is divided into sub-models for which test inputs are selected individually and merged afterwards. However, sub-models lead to larger test suites and the constraint handling is still not as efficient

---

[2]See https://github.com/dakusui/jcunit (Last access on 1st December 2020).

[3]See https://junit.org (Last access on 1st December 2020).

[4]See https://github.com/dakusui/jcunit/issues/49 (Last access on 1st December 2020).

[5]See https://github.com/dakusui/jcunit/wiki/0.7.x-manual-07.Automatic-negative-test-generation (Last access on 1st December 2020).

as CSP solving or minimal irrelevant schemata. In addition, the missing internal representation not only results in an inefficient constraint handling technique. It also impedes further reasoning to develop automated support and analysis techniques.

Other CRT Test Methods   A survey conducted by KHALSA & LABICHE [KL14] focused on test selection strategies and tools for CT and identified two more tools that support CRT: (1) `PictMaster` which integrates `PICT` into Microsoft Excel and (2) `FoCuS` which is a CT tool developed by IBM. According to SEGALL et al. [STF11], `FoCuS` supports all features that are also present in `PICT` which includes "testing with negative values that indicate error conditions". Beyond that, we do not have an indication that invalid value combinations are explicitly considered. Although, KHALSA & LABICHE [KL14] state in their survey that "constraints can also be used to test the robustness of the system, e.g., behavior of the system if the user selects invalid options". But, not further information or references are provided.

4.2. Non-Combinatorial Robustness Testing

SHAHROKNI & FELDT [SF13] provide a systematic review of software robustness and as such they also discuss verification and validation of robustness. Since these tools are only remotely related to our work, we discuss them only briefly. For more information, please refer to systematic reviews by SHAHROKNI & FELDT [SF13] and MICSKEI et al. [Mic+12].

Fault injection is the primary technique used for robustness testing [SF13]. Given our transformational view of programs $P : I \rightarrow O$, fault injection can be achieved by stimulating a program P with non-valid inputs $\tau \notin SI$. In practice, fault injection not only encompasses passing inputs to a program but also injecting faults into or simulating faults in physical systems [Mic+12].

Related black-box test methods are the *robust* variants of equivalence class testing (ECT) and boundary value testing (BVT) that incorporate invalid values to create invalid test inputs [Jor14]. For robust weak ECT and robustness BVT, one invalid value is combined with all other parameter values being valid. Robust strong ECT and robust worst-case BVT describe the cross-product of all valid and invalid values. They comprise single error coverage and N-wise coverage for invalid values but do not mention invalid value combinations. They also do not support t-combinatorics in between single error coverage and N-wise coverage.

Robustness testing is often automated and random testing is one of the first approaches to robustness testing where random inputs are generated and used to stimulate a program [Mic+12]. It can be efficient and effective when combined with a simple test oracle such "program may never crash or hang".

This is the main objective of these robustness testing approaches whereas the objective of our CRT is to check whether business rules are implemented correctly. For more detailed test oracles that do not check for crashes but for the correct implementation of business rules like "the start date must not be after the end date of a booking", irrelevant test inputs with no defined behavior may be selected by random testing. In addition, invalid test inputs that cover more than one truly-minimal invalid schema may be generated and suffer from invalid input masking.

Grammar-based and type-specific tools like `Ballista` (cf. [KJS98]; [KD00]) are more advanced approaches that rely on a model that describes the legal structure of inputs. Thereby, the amount of irrelevant test inputs and invalid test inputs that cover more than one truly-minimal invalid schema can be reduced [Mic+12]. However, the models describe legal structure of inputs on a syntactic level and not on a semantic level. Thereby, business rules are not adequately captured.

In a review on exception analysis by CHANG & CHOI [CC16], 15 publications on white-box test methods are identified that incorporate the structure of exceptional control-flows. While these approaches can be effective in revealing robustness faults, they only test existing EH and cannot reveal robustness faults caused by the absence of EH.

Part II.

A Combinatorial Robustness Test Method

Chapter 5.

A Combinatorial Robustness Fault Model

Contents

A CRT test method requires a fault model which functions as the basis of test adequacy criteria and test selection strategies. A fault model is a collection of hypothesized faults. Test adequacy criteria and test selection strategies are then constructed to detect these hypothesized faults.

Since CT already has a fault model – the t-factor fault model – we first evaluate if CT and its t-factor fault model are appropriate to describe and to detect robustness faults.

Therefore, we first conduct an experiment (Section 5.1) to assess the appropriateness of CT in the presence of invalid input masking. Since our findings indicate that extensions like CRT are useful, we derive a classification of robustness fault characteristics (Section 5.2). Then, we conduct a case study on MFCSs in order to investigate robustness failures and to draw conclusions regarding the classified robustness fault characteristics (Section 5.3). Afterwards, we integrate the findings into a refined t-factor fault model for robustness faults that can be used to guide the development of test adequacy criteria and test selection strategies (Section 5.4).

5.1. Fault Detection Effectiveness in the Presence of Invalid Input Masking

5.1.1. Motivation

Although intended to make a SUT more robust, EH is itself a source of robustness faults and failures [MT97]; [Mar11]; [SAW12]. In the context of CT, the invalid input masking effect may mask a fault and prevent it from being detected by a test input. As discussed in the chapter on related work (Chapter 4, Page 55 and following), the invalid input masking effect is used to justify CRT extensions because the effectiveness of CT can become worse due to the masking. But beyond qualitative arguments and examples, it is unclear to what extent the invalid input masking effect can impact the effectiveness of CT. It is thus unclear to what degree CT is appropriate and if extensions like CRT are needed. Therefore, we investigate the following research objective: How effective is CT and what are its shortcomings in detecting faults when invalid input masking is present?

An important aspect when assessing the effectiveness of CT is **collateral coverage**: The testing strength t determines which MFCSs are guaranteed to be detected. When testing with a test suite that satisfies t-wise coverage, all schemata with degree $d = t$ appear in some test input and testing should fail for each test input that covers a MFCS with degree $d = t$. In addition, all smaller sub-schemata with degrees $d < t$ also appear in some test input and testing consequently fails for MFCSs with degrees $d < t$ as well. Testing with t-wise coverage cannot only detect faults of factor $d \leq t$. Even some larger super-schemata of degrees $d = (t + k)$ with $k > 0$ and $d \leq n$ are covered by a test suite that satisfies t-wise coverage [CZ11]. This so-called collateral coverage can help detect additional faults with MFCSs of degrees $d > t$ [Pet+15]. While t-wise coverage provides a guarantee that each MFCS with a degree $d \leq t$ is covered by at least one test input, it cannot be guaranteed that a MFCS with a degree $d > t$ is covered. However, there is a probability that these larger MFCSs are also covered due to the collateral coverage.

For each-choice ($t = 1$), the fault detection probability of a minimal test suite can be calculated for a given fault and a given IPM. When increasing the testing strength to $t > 1$, the probability of fault detection also increases. But, finding a minimal set of test inputs that satisfies t-wise coverage for $t > 1$ is in general NP hard [LT98]. Heuristics are used as test selection strategies which produce small but not always minimal test suites and the probability of detecting a fault based on collateral coverage cannot be calculated easily anymore.

In this context, CHEN & ZHANG [CZ11] introduce the tuple density metric which "is the sum of t and the percentage of the covered [schemata with degree $d = t + 1$] out of all possible [schemata with degree $d = t + 1$]". It allows distinguish test suites even if they have the same size and satisfy t-wise coverage for the same testing strength t. A test suite that covers more [schemata with degree $d = t + 1$] has a higher tuple density and the probability that MFCSs are detected due to collateral coverage is also higher.

As the test suites and tuple density values differ for each IPM and each test selection strategy, it becomes difficult to calculate the probability. Particularly noteworthy are test selection strategies of CHEN & ZHANG [CZ11] or CHOI et al. [Cho+16], which specifically try to increase the tuple density for test suites that satisfy t-coverage.

In this respect, WANG & QI [WQ15] present a model to compute the lower boundary of fault detection probability for CT. Later, YU et al. [Yu+18] revisit the model to compute the lower boundary of fault detection probability and publish a counterexample to illustrate a fault in the model. This is also discussed in the related work on testing of logical expressions (See Page 57).

> **Running Example** Suppose another fault in the Ordering Web-Service implementation (Listing 2.1, Page 14) where the function `isInvalidCountry(country)` is incorrect and does not detect invalid country codes which are modeled as `[Country:123]`.
>   The fault is illustrated below.
>
> ```
> if(isInvalidNumberOfPackages(packages)) return "INV_PACKAGES";
> if(isInvalidCountry(country)) /* fault, do nothing */;
> if(isInvalidPhoneNumber(phone)) return "INV_PHONE";
> if(isInvalidCountryCode(country, phone)) return "INV_CODE";
> ```
>
> To satisfy t-wise coverage with $t = 1$, a minimal set of three test inputs can be selected with exactly one test input that covers `[Country:123]`. In total, 18 test inputs that cover `[Country:123]` can be combined and one of them must be selected. Of the 18 test inputs, six cover `[Packages:123]` that cause invalid input masking. Therefore, the probability of selecting a test input that activates the fault is $\frac{12}{18} = 67\%$.
>   In this example, t-wise coverage with $t = 2$ guarantees the selection of a MFCS like `[Packages:1, Country:123]` where the probability increases to 100%. In a more complicated example where three particular values are required to activate a fault, the probability of detecting a fault by a test suite that satisfies t-wise coverage with $t = 2$ cannot be calculated simply.

Because of the difficulties in calculating the probability, we instead conduct a controlled experiment to investigate the research objective.

First, we apply the t-factor fault model to describe robustness faults. Then, we derive factors that influence the detectability of robustness faults. Afterwards, we conduct a controlled experiment to measure the effectiveness of CT.

Please note, the assessment is consciously limited to robustness faults that are activated by invalid values. Invalid value combinations are excluded since they complicate the description of influencing factors. But the assessment still demonstrates an impact of invalid input masking on the effectiveness: If invalid values and their masking have an impact on the effectiveness, then invalid value combinations have an impact as well because their masking and their MFCSs encompass more parameters.

5.1.2. Applying the t-Factor Fault Model

Overview

In general, a fault model is a description of hypothesized faults and how to activate them [Bin00]. To observe a failure, the source code region or regions that contain the fault must be reached (*reachability*), the state must be incorrect after executing the regions (*infection*) and the infection must propagate to a failure (*propagation*) [LO17].

The idea of CT and t-wise coverage is based on the corresponding t-factor fault model which is formally introduced by DALAL & MALLOWS [DM98]. The t-factor fault model assumes that faults are caused by the interaction of t parameters [DM98]. These faults are called t-factor faults and t refers to the number of involved parameters.

Faults are static defects in the source code that manifest as incorrect computer instructions in the SUT [Avi+04]. In contrast, MFCSs are black-box concepts that are defined in terms of a specification and parameters and values of an IPM.

A t-factor fault is an abstraction of an actual fault. It is associated with a condition that is defined relative to the SUT such that each input that satisfies the condition activates the fault and causes the behavior of the SUT to deviate from its specification.

Definition 82 (t-Factor Fault) *A t-factor fault is an abstraction that describes an actual fault in the SUT by a condition over t parameters of the SUT which must be satisfied by an input to the SUT in order to activate the fault.*

To detect a t-factor fault, i.e. to satisfy reachability and infection, it is necessary and sufficient for a test input to satisfy the condition of the t-factor fault (short: fault condition). If the fault then propagates to a failure, it can be revealed by a test oracle [LO17]. To satisfy the fault condition, the test input covers a MFCS with the parameter value pairs that correspond to the parameters and values of the fault condition.

MFCSs are described relative to an IPM and t-factor faults are described relative to a SUT model. Both models are input-oriented abstractions that describe an actual program as a set of parameters. Since t-factor faults and MFCSs are used in the same context, both models typically use the same set of parameters to ensure that at least one schemata selected to satisfy t-wise coverage activates the t-factor fault. Although strictly speaking, the SUT model for t-factor faults and the IPM for MFCSs could use different sets of parameters. Then, a mapping function would be necessary to *align* the parameters of both models.

When assuming the same set of parameters, the input domain of a SUT I can be structured by n parameters with each parameter p_i having a domain D_i consisting of a potentially infinite number of values, i.e. $I = D_1 \times ... \times D_n$. In contrast, the n parameters of an IPM have associated nonempty sets of discrete values $V = \{V_1, ..., V_n\}$ where each value domain V_i contains all values of the SUT $V_i \subseteq D_i$ a tester is interested in. The cartesian product of all value domains $E = V_1 \times ... \times V_n$ denotes the exhaustive test suite which contains all test inputs that are modeled by the IPM. It is a discrete subset of the SUT's input domain $E \subseteq I$ and a superset $T \subseteq E$ of each test suite T that is selected by a test selection strategy.

> **Running Example**   To describe the aforementioned fault in the `isInvalidCoun-`
> `try(country)` function as a t-factor fault, EH must be taken into account. The fault
> cannot be described as a 1-factor fault because not every input that satisfies `isIn-`
> `validCountry(country)` activates the fault. Consequently, `[Country:123]` is
> not a MFCS with regards to the given IPM (Figure 2.1, Page 14).
>
> Detecting the fault does not only require an invalid country code. A valid number of
> packages is necessary as well to prevent invalid input masking. Therefore, the fault can

be modeled as a 2-factor fault using a conjunction over all four parameters.

```
¬isInvalidNumberOfPackages(packages)
∧isInvalidCountry(country)
```

For the given IPM, the following schemata are minimal failure-causing.

```
[Packages:1, Country:123],
[Packages:3, Country:123]
```

Oftentimes, one MFCS exists for one t-factor fault and the degree of the MFCS is of the same size as the t-factor fault, i.e. $d = t$. Therefore, a t-factor fault is often considered and used as a synonymous term for MFCS. But there are also differences since a t-factor fault depends on the implementation in which it is located while a MFCS depends on the parameters and values of its IPM.

First, there can be zero, one, or multiple MFCSs that activate the same t-factor fault. Zero MFCSs exist when the value domains of an IPM do not contain the values that are necessary to satisfy the fault condition. In contrast, when each value domain of all parameters that are involved in the fault condition contains a value to satisfy the fault condition, one MFCS exists. When a value domain of an involved parameter contains multiple values that can be used to satisfy the fault condition, multiple MFCSs exist for the same t-factor fault.

Second, the number of parameters involved in the t-factor fault condition predetermines an upper boundary of the degree of MFCSs. Typically, the number of involved parameters and the degree of MFCSs are the same, i.e. $d = t$.

There is also an exception which occurs in the context of incomplete IPMs that result in parts of the SUTs not being tested. MFCSs do not cover values of a parameter when all values of that parameter are sufficient to satisfy the fault condition. For instance, when no invalid value is modeled for a particular parameter, related EH is skipped and there cannot be any invalid input masking for that parameter. But the MFCSs cover fewer parameter-value pairs than the t-factor fault, i.e. $d < t$. Since this work is about EH, our objective is to test and not to skip EH. Therefore, we focus on complete IPMs without skipped invalid schemata to test EH extensively.

Another case is the occurrence of irrelevant schemata. The exclusion of irrelevant values by removing them from the value domains and of irrelevant value combinations by modeling them as exclusion-constraints does not result in untested parts of the SUT. While the existence of irrelevant schemata may increase the number of parameters involved in a fault condition, the exclusion of irrelevant schemata decreases the degree of MFCSs.

Running Example The aforementioned fault in the `isInvalidCountry(country)` function allows illustrating this exception. The corresponding t-factor fault involves two parameters because of the EH related to invalid numbers of packages. If there were no invalid number of packages modeled in the IPM, the EH would be skipped and invalid input masking could not happen.

```
Packages: { 1, 3 }
```

> As a result, every combination of a number of packages and `[Country:123]` would be failure-causing which makes it a MFCS with a degree of $d = 1$.

When applying the t-factor fault model, the number of parameters involved in a fault condition and the number of MFCSs that activate the fault are important characteristics that influence the effectiveness of CT. In the following, they are discussed in more detail.

Characteristics affecting Size of t-Factor Faults

Number of Parameters involved in Exception Handling   In the presence of irrelevant and invalid input masking, the condition to detect a t-factor fault can be formulated as a conjunction of two sub-conditions. First, the location of the fault must be reached by ensuring that the input is executable and that no prior EH propagates and terminates the SUT. We denote this as the **effective prevention sub-condition**. Second, a MFCS must activate a fault so that it can cause an incorrect program state that can propagate to a failure. We denote this as the **activation sub-condition**.

> **Running Example**   The   fault   condition   can   be   divided   as   follows: `¬isInvalidNumberOfPackages(packages)` is used for effective prevention and `isInvalidCountry(country)` is used for activation.

The distinction of effective prevention and activation sub-conditions also allows to further explain the aforementioned exception in the context of incomplete IPM. When no irrelevant value combination or no invalid value is modeled by an IPM, the effective prevention sub-condition is always satisfied and it is not necessary to specify failure-causing values for those parameters.

The size of a t-factor fault increases with the number of parameters involved in effective prevention and activation sub-conditions. To guarantee that a t-factor fault is detected, the test input set must satisfy t-wise coverage of the same size.

Priority of Exception Handling   Each EH with a position in the control-flow prior to the fault has the potential to propagate and terminate the SUT before the fault is reached. Therefore, each prior EH extends the effective prevention sub-condition.

> **Running Example**   The effective prevention sub-condition for the fault in the `isInvalidCountry(country)` function is already illustrated beforehand. For a fault in the first EH of the example (Line 5 of Listing 2.1, Page 14), the effective prevention sub-condition would be empty. In contrast, a fault in the third EH would be modeled by a effective prevention sub-condition that includes the prior two EHs, e.g. `¬(isInvalidNumberOfPackages(packages) ∨ isInvalidCountry(country))`.

While testing with t-wise coverage is appropriate to detect a fault that can be modeled as a t'-factor fault with $t' \leq t$, the location of the fault is required to be known beforehand to determine the appropriate testing strength t. But, the specification does typically not impose a specific

order of EH. To determine testing strength t independent from the specific location of a fault within the control-flow, all EHs in every possible order must be taken into account. Then, the testing strength t would grow with the number of parameters checked by EH.

Deriving the testing strength from all parameters that are involved in any EH is an idealized approach that allows testing for incorrect EH without knowing the location of the fault. It is assured that each EH is reached and potential faults are detected. While this testing strength denotes the lower limit to ensure that potential faults are detected independently from the ordering of EHs, testing is still conducted for a specific implementation with a specific ordering. Therefore, we distinguish the **effective prevention sub-condition** which is sufficient for a specific implementation from the **general prevention sub-condition** which is sufficient for all possible orderings. On average, the effective prevention sub-condition considers fewer EHs which improves the likelihood of detecting a fault when using a testing strength that is not sufficient for the general prevention sub-condition.

> **Running Example**   The general prevention sub-condition for the aforementioned fault in the `isInvalidCountry(country)` function would be as follows.
>
> $$\neg(\texttt{isInvalidNumberOfPackages(packages)}$$
> $$\lor \texttt{isInvalidPhoneNumber(phone)}$$
> $$\lor \texttt{isInvalidCountryCode(country, phone))}$$
>
> Together with the activation sub-condition `isInvalidCountry(country)` it constitutes a 3-factor fault since three parameters are involved in the fault condition.

Characteristics affecting the Number of Minimal Failure-causing Schemata

Besides the number of parameters involved in the fault condition, the number of MFCSs is also an important factor that affects the fault detection effectiveness. Depending on the number of parameters involved in any EH and the location of a fault, the required testing strength to guarantee the detection can easily exceed $t > 6$ and collateral coverage becomes more important.

A test suite that satisfies t-wise coverage contains all MFCSs with degrees of $d \leq t$ at least once. In contrast, MFCSs with degrees of $d > t$ have a probability of less than 100% to be covered and the detection cannot be guaranteed. Although, the probability of detecting a t'-factor fault with $t' > t$ increases with the number of MFCSs that exist in the exhaustive test suite because it becomes more likely that at least one of them is covered due to collateral coverage.

Therefore, it is important to discuss characteristics that increase or decrease the number of MFCSs for a t'-factor fault. To illustrate the effects, the following discussion is based on a given t'-factor fault with $t' > 1$, a given IPM, and a testing strength of $t = 1$. The impact of adding another value to the IPM is assessed.

In general, the impact of valid and invalid values as well as the involvement of a value's parameter in the fault condition can be distinguished.

Number of Valid Values   A valid value of a parameter that is involved in the activation sub-condition has no effect on the number of MFCSs because it cannot contribute to satisfy the activation sub-condition. The valid value and a value that is necessary to satisfy the activation sub-condition are mutually exclusive because the activation sub-condition of a robustness fault requires an invalid value.

A valid value of a parameter that is involved in the effective prevention sub-condition can contribute to satisfy the effective prevention sub-condition. As a result, multiple additional MFCSs exist that cover the valid value.

A valid value of a parameter that is neither involved in the activation nor in the effective prevention sub-condition has no immediate effect on the number of MFCSs and the probability of selecting failure-causing test inputs. Although, each value contributes to the set of t-sized schemata that must be covered by some test input to satisfy t-wise coverage. As a result, additional values may affect the overall selection of test inputs and may cause a test selection strategy to introduce some redundant selections that affect the fault triggering probability. But, these effects are specific to particular test selection strategies and beyond the scope of this discussion.

> **Running Example**   For the t-factor fault with the fault condition `¬isInvalidNumberOfPackages(packages)` ∧ `isInvalidCountry(country)`, adding another valid number of packages [Packages:5] is an example of adding an valid value that does satisfy the effective prevention sub-condition. It increases the probability of selecting a test input that covers [Country:123] and that does not cause invalid input masking from $\frac{12}{18} = 67\%$ to $\frac{18}{24} = 75\%$.
>
> In contrast, adding a valid [PhoneNumber:+49] that is not involved in the fault condition does not change the probability because the number of MFCSs remains the same. Without the additional value $\frac{12}{18} = 67\%$ of the test inputs that cover [Country:123] can detect the fault. With the additional value $\frac{16}{24} = 67\%$ of the test inputs that cover [Country:123] can detect the fault.

Number of Invalid Values   An invalid value of a parameter that is involved in the activation sub-condition results in multiple MFCSs if the invalid value satisfies the activation sub-condition. If the invalid value does not satisfy the activation sub-condition, it has no effect on the number of MFCSs because a satisfying value and this invalid value are mutually exclusive selections.

An invalid value of a parameter that is involved in the effective prevention sub-condition increases the risk of invalid input masking because it results in multiple schemata that do not satisfy the effective prevention sub-condition.

An invalid value of a parameter that is neither involved in the activation nor in the effective prevention sub-condition has no immediate effect on the number of MFCSs and the probability of selecting failure-causing test inputs. The invalid values do not affect the MFCSs because the EH would happen at a location after the fault.

Running Example For the aforementioned t-factor fault with the fault condition `¬isInvalidNumberOfPackages(packages) ∧ isInvalidCountry(country)`, adding another invalid country `[Country:456]` that satisfies the activation sub-condition increases the probability because more MFCSs exist. The probability of selecting a test input with `[Country:123]` that does not cause invalid input masking remains the same $\frac{12}{18} = 67\%$. However, a second test input with `[Country:456]` that has the same probability must be selected as well.

In contrast, adding another invalid number of packages `[Packages:456]` is an example of adding an invalid value that does not satisfy the effective prevention sub-condition. The number of MFCSs remains the same. Moreover, it decreases the probability of selecting a test input with `[Country:123]` that does not cause invalid input masking from $\frac{12}{18} = 67\%$ to $\frac{12}{24} = 50\%$.

5.1.3. Experiment Design

Test Scenarios

The objective of our experiment is to evaluate the effectiveness of CT when selecting test inputs in the presence of invalid input masking due to EH. In particular, we want to determine in which cases CT is appropriate such that CRT and its additional effort can be avoided. Therefore, we generated test inputs with different characteristics and executed them in test scenarios.

For an experiment, it is important to control the factors that can influence the results. Therefore, we designed artificial test scenarios and changed different characteristics in a controlled and traceable way. Each test scenario contains exactly one fault in EH that always propagates to a failure when activated and the failure is always revealed by the test oracle. Thus, the selection of test inputs is the only factor that influences the results of test execution.

Based on the application of the t-factor fault model, we describe each **test scenario** SC in terms of (1) the number of parameters, (2) the number of parameters that are involved in EH, (3) the number of valid values per parameter, (4) the number of invalid values per parameter, and (5) the location l of the incorrect EH that contains a fault.

The implementation of a test scenario is illustrated in Listing 5.1. A test scenario accepts input values for a number of parameters and includes a sequence of EHs with one EH for one parameter implemented with `if` statements. The result of a test scenario execution is `INV_-INPUT` if an invalid value is correctly identified and if the execution is correctly terminated. If an invalid value is identified by an incorrectly implemented EH, `null` is returned instead; `VAL_INPUT` is returned if all EHs are passed because all values are valid.

```
1  String checkInput (Object a, b, c){
2    if(isInvalid(a)) return "INV_INPUT";
3    else if(isInvalid(b)) return null;
4    else if(isInvalid(c)) return "INV_INPUT";
5    else return "VAL_INPUT";
6  }
```

Listing 5.1: Illustration of a Test Scenario Implementation

The illustrated test scenario uses 3 parameters where the second EH (location $l = 2$) is incorrect. The number of valid and invalid values per parameter is implicitly encoded by the IPM that is used to generate test inputs.

By varying the index of the incorrect EH, three different test scenarios for our example can be created, where either the first, second or third EH is incorrect. A set of test scenarios that shares the same parameters and values but differs in the index of the incorrect EH is called a **test scenario family** SC^*.

As a notation, we use `P-V-I-E` where `P` refers to the total number of parameters, `V` refers to the number of valid values per parameter, `I` refers to the number of invalid values per parameter, and `E` refers to the number of parameters that are involved in EH.

The experiment starts with a *root* test scenario family `6-1-1-6`. It is chosen to represent a simple application of CT. With fewer parameters, we believe that CT would not be beneficial. With fewer values, EH and strong invalid test inputs would not be possible. The root test scenario family is then extended as follows. The number of parameters is increased by six up to 30 parameters. The number of EHs in a test scenario family either remains at six or is equal to the number of parameters. The total number of values per test scenario family is extended up to six with one to five valid and invalid values.

Test Input Generation

The IPMs used in this experiment share the same set of parameters with the test scenario and define the number of valid and invalid values per parameter. To avoid any bias, we use test suites from the NIST Covering Array Tables[1]. They are publicly available and contain many of the smallest known test suites [For+08]. Table 5.1 depicts the sizes of the test suites. Column *P* refers to the number of parameters and column *V* refers to the total number of values per parameter. The test suites are reused for different ratios of valid and invalid values.

The order of parameters and values in an IPM has no impact on whether a generated test suite satisfies t-wise coverage. However, it has an impact on which schemata with degree t are combined in a single test input. To reduce the effect of accidental fault detection that is caused by ordering, the parameters and values of a test suite are randomly reordered and 100 different variants of each test suite are generated. The set of all test suite variants is called a **test suite family** T^*.

The testing strengths used in the experiment range from $t = 1$ to $t = 5$ because most failures are induced by this range.

Evaluation Metrics

A common metric to evaluate test selection strategies is called **fault detection effectiveness (FDE)** [GOA05]; [Pet15]. A test suite T is denoted as *failing* for a test scenario SC if at least one of the test inputs $\tau \in T$ detects the t-factor fault f in SC and the test suite consequently fails.

$$\texttt{failing}(T, SC) = \begin{cases} 1 \text{ if } \exists \tau \in T \text{ that fails for } SC \\ 0 \text{ otherwise} \end{cases}$$

---

[1] https://math.nist.gov/coveringarrays/ (Last access on 1st May 2020)

P	V	t=1	t=2	t=3	t=4	t=5
6	2	2	6	14	26	42
6	3	3	15	49	140	318
6	4	4	24	109	442	1377
6	5	5	38	209	1059	4195
6	6	6	52	361	2165	10407
12	2	2	8	19	47	105
12	3	3	20	71	262	885
12	4	4	31	170	835	3854
12	5	5	48	329	2068	11889
12	6	6	67	564	4295	29645
18	2	2	10	24	60	144
18	3	3	20	88	336	1241
18	4	4	36	202	1098	5458
18	5	5	54	401	2725	16843
18	6	6	75	689	5650	42102
24	2	2	10	26	71	175
24	3	3	21	98	396	1513
24	4	4	40	232	1298	6629
24	5	5	59	454	3203	20482
24	6	6	83	786	6662	51287
30	2	2	11	28	80	200
30	3	3	23	106	446	1715
30	4	4	41	255	1448	7555
30	5	5	63	495	3589	23369
30	6	6	86	859	7473	58468

Table 5.1.: Test Suite Sizes used for the Experiment

Using the `failing` function, FDE is defined as the ratio between the number of test suites $T \in T^*$ of a family that fail for a test scenario SC and the number of all test suites in a family T^* that is used to test SC.

$$\text{FDE}(T^*, SC) = \frac{\sum_{T \in T^*} \texttt{failing}(T, SC)}{|T^*|}$$

In other words, the FDE is based on randomized variants of a test suite that all satisfy the same testing strength. They all are used to test the same test scenario SC which has an incorrect EH with a fixed location.

The FDE metric can be used to identify and isolate characteristics that influence the FDE metric values. However, that information cannot be used in practice because one must know which EH is incorrect beforehand. Therefore, we introduce the **average fault detection effectiveness (AFDE)** which is the average FDE over a family of test scenarios SC^*. Thus, AFDE represents the effectiveness of a test scenario family when knowing that an EH is incorrect but without

knowing its location.

$$\text{AFDE}(T^*, SC^*) = \frac{\sum_{SC \in SC^*} \text{FDE}(T^*, SC)}{|SC^*|}$$

List of Symbols

SC test scenario

SC^* test scenario family

T^* test suite family

l location of an incorrect EH

5.1.4. Results and Discussion

Overview

The overall results of the experiment are consistent with the indications of applying the t-factor fault model to robustness faults. Whenever an EH at location l is incorrect, the effective prevention sub-condition includes the parameters checked by all EHs with earlier locations. Since one parameter is involved in the activation sub-condition, $l - 1$ parameters belong to the effective prevention sub-condition.

When the first EH is incorrect ($l = 1$), the effective prevention sub-condition is empty and one parameter is involved in the activation sub-condition. The resulting 1-factor fault is detected in each test scenario and by all test suites that satisfy the testing strength $t = 1$. For all considered testing strengths $1 \leq t \leq 5$, all l-factor faults are detected by all test suites that satisfy the corresponding testing strength. Beyond that, collateral coverage causes higher $l + k$-factor faults to be repeatedly detected by test suites that satisfy lower testing strengths.

Fault Detection Effectiveness

Table 5.2 depicts an excerpt[2] of the FDEs computed for test scenario families with six to 30 parameters consisting of two valid values and one invalid value. Each test scenario family contains six EHs. The `l` column denotes the location of the incorrect EH, `t` denotes the testing strength that is satisfied by the family of test suites and the remaining columns denote the computed FDE values.

According to the t-factor fault model, a testing strength of $t = 2$ is required to guarantee that incorrect EH at location $l = 2$ is detected.

According to the FDE values of this experiment, the same fault is triggered by all test suites that satisfy a testing strength of $t = 2$. Moreover, the same fault is on average also triggered by 67.6% of all test suites that only satisfy $t = 1$. The exact FDE values for the five test scenarios

---

[2]The complete data of this experiment can be found in Appendix A (Page 279 and following).

l	t	6-2-1-6	12-2-1-6	18-2-1-6	24-2-1-6	30-2-1-6
1	1	100	100	100	100	100
2	1	63	74	65	66	70
2	2	100	100	100	100	100
3	1	43	34	38	35	52
3	2	98	99	100	99	100
3	3	100	100	100	100	100
4	1	30	35	33	29	31
4	2	90	96	95	95	99
4	3	100	100	100	100	100
5	1	22	22	20	19	27
5	2	72	82	84	85	91
5	3	100	100	100	100	100
6	1	12	14	11	13	18
6	2	61	61	69	63	69
6	3	95	99	100	100	99
6	4	100	100	100	100	100

Table 5.2.: FDE of Test Scenario Families with Six to 30 Parameters and Six EHs (Excerpt)

(`6-2-1-6`, `12-2-1-6`, `18-2-1-6`, `24-2-1-6`, and `30-2-1-6` with $l = 2$ and $t = 1$) are depicted in the second row of Table 5.2 (Page 85).

Testing with higher testing strengths is even more effective. On average, test suite families that satisfy a testing strength of $t = 2$ detect incorrect EHs at location $l = 3$ in 99.2% of all cases. As expected, a family of test suites that satisfies $t = 3$ always detects the incorrect EH at location $l = 3$. However, incorrect EHs at locations $l = 4$ and $l = 5$ are always detected as well. The incorrect EH at location $l = 6$ is detected by 98.6% of all test suite families.

When increasing the number of parameters while the number of EHs remains at six, the data indicates no general trend for the FDE. In many cases, FDE improves slightly. Although, there are cases where the FDE deteriorates. For instance, the FDE for detecting an incorrect EH at location $l = 2$ with a test suite family that satisfies only testing strength $t = 1$ improves from 64% for the test scenario family with six parameters to 74% for the test scenario family with 12 parameters. But, it deteriorates to 65% for the test scenario family with 18 parameters.

Table 5.3 (Page 86) depicts the FDE for test scenario families with six to 30 parameters and an equal number of EHs. Increasing the number of EHs such that one EH exists for each parameter has no direct impact on the FDE. Over all indices, testing strengths and parameter sizes, the difference between the FDE for six EHs and the FDE for $|P|$ EHs is 1.25 percentage points on average. For instance, the difference for $l = 2$, $t = 1$ and 12 parameters is 11 percentage points with an FDE of 74% for 6 EHs (Table 5.2, Page 85) and a FDE of 63% for 12 EHs (Table 5.3, Page 86).

The biggest deviations are noticed for test scenario families with 30 parameters. For $l = 2$ and $t = 1$, the difference between the FDE for six EHs (57%) and the FDE for 30 EHs (42%) is 15 percentage points. For $l = 6$ and $t = 2$, the difference between the FDE for six EHs (40%)

l	t	6-2-1-6	12-2-1-12	18-2-1-18	24-2-1-24	30-2-1-30
1	1	100	100	100	100	100
2	1	63	63	66	77	66
2	2	100	100	100	100	100
3	1	43	48	55	39	49
...	...	...	...	...	...	...
8	3		78	86	91	87
...	...	...	...	...	...	...
10	3		52	52	61	55
10	4		95	96	98	99
...	...	...	...	...	...	...
12	3		34	29	30	34
12	4		67	78	88	83
12	5		99	100	100	100
...	...	...	...	...	...	...
18	3			2	2	2
18	4			13	16	16
18	5			38	36	44
...	...	...	...	...	...	...
24	4				2	2
24	5				3	5

Table 5.3.: FDE of Test Scenario Families with Six to 30 Parameters and an Equal Number of EH (Excerpt)

and the FDE for 30 EHs (60%) is -20 percentage points.

Analyzing later locations of incorrect EHs ($l > 6$) emphasizes that the required testing strength to detect a fault increases as well. Although, the testing strength grows slower compared to the growth of locations ($l \leq 6$). For instance, $t = 3$ is sufficient to detect incorrect EHs at location $l = 5$ in 100% of all cases and it is almost sufficient (98.6% on average) to detect all incorrect EHs at location $l = 6$. For $l = 8$, the average FDE decreases to 85.5%. For $l = 18$, only 2% of all incorrect EHs are detected. In comparison, $t = 4$ is sufficient to detect an incorrect EH at location $l = 8$ in 100% of all cases. $t = 5$ is sufficient to detect a fault at location $l = 11$ in 100% of all cases and even a fault at location $l = 12$ is detected in 99.75% of all cases. Afterwards, the FDE for $t = 5$ decreases to an average of 39.3% for location $l = 18$ and to 4% for $l = 24$.

Overall, the findings are consistent with the t-factor fault model. Changing the number of parameters of a test scenario family has no clear effect on the FDE as well as changing the total number of EHs. In contrast, increasing or decreasing the specific location of the incorrect EH has the biggest effect on the FDE.

According to the t-factor fault model, a testing strength t guarantees to detect incorrect EH up to location $l = t$. Beyond that, collateral coverage of CT causes a reliable detection of incorrect EH at later locations. Although for index $l = 12$ and higher, even $t = 5$ is not sufficient to detect

incorrect EH for the test scenario families depicted in Table 5.3 (Page 86).

To use the knowledge acquired from the FDE metric, knowledge about the location of an incorrect EH is required which makes it hard to use the information in practice. In contrast, the AFDE allows drawing conclusions regarding the effectiveness of CT assuming that one EH is incorrect when only the number of checked parameters as well as the number of valid and invalid values is known. Therefore, all further analyzes are based on the AFDE metric.

Average Fault Detection Effectiveness

P	E	t	1-1	1-2	1-3	1-4	1-5	2-2	3-3	2-1	3-1	4-1	5-1
6	6	1	36	33.17	34.83	34.17	33.83	47.5	53.33	45	53	63.33	65.83
6	6	2	55.33	53.83	53	53.33	52.33	79.5	94.67	86.83	97	99.5	99.83
6	6	3	71.83	71.5	73.33	70.83	70.67	98	99.83	99.17	100	100	100
6	6	4	86.67	86.67	86.5	85.33	86.33	100	100	100	100	100	100
6	6	5	95	96	96.33	97.33	98.33	100	100	100	100	100	100
6	6	*Avg.*	68.97	68.23	68.8	68.2	68.3	85	89.57	86.2	90	92.57	93.13
12	6	1	33.33	32.17	31.5	33	34	46.5	57.5	46.5	55.83	64	67.5
12	6	2	60	60.67	57.33	54.67	56	84.5	92.83	89.67	99	99.67	100
12	6	3	76.67	74.83	74.33	75	71.33	99.33	100	99.83	100	100	100
12	6	4	94.17	92.5	90.5	91	88.83	100	100	100	100	100	100
12	6	5	97.83	99.33	99.83	99.33	100	100	100	100	100	100	100
12	6	*Avg.*	72.4	71.9	70.7	70.6	70.03	86.07	90.07	87.2	90.97	92.73	93.5
18	6	1	33.33	32.83	33.33	32.83	34	43.83	52.17	44.5	57	58.33	65.83
18	6	2	62.33	58.83	57	56.67	53.5	86	94.5	91.33	97.5	100	100
18	6	3	80.83	79.67	77.33	74	73.83	99.5	100	100	100	100	100
18	6	4	94	94.17	94.17	92.17	90.5	100	100	100	100	100	100
18	6	5	99.67	100	100	100	100	100	100	100	100	100	100
18	6	*Avg.*	74.03	73.1	72.37	71.13	70.37	85.87	89.33	87.17	90.9	91.67	93.17
24	6	1	31.33	33.33	33	33.5	32.67	46.17	52.33	43.67	53.67	60.33	66.5
24	6	2	63.17	59.83	58.17	57.33	55.5	85.83	94.17	90.33	98.33	100	100
24	6	3	80.67	77.67	77.5	75.67	75	99.67	100	100	100	100	100
24	6	4	94.83	94.67	94.33	91.5	91.83	100	100	100	100	100	100
24	6	5	100	99.83	100	100	100	100	100	100	100	100	100
24	6	*Avg.*	74	73.07	72.6	71.6	71	86.33	89.3	86.8	90.4	92.07	93.3
30	6	1	35.33	34.5	33.17	33.83	32.33	45.83	51.67	49.67	54.17	60.83	69.17
30	6	2	65.17	61.67	57.17	57.17	56.67	84.83	95	93.17	98.83	100	100
30	6	3	84.33	83.5	79	77	72.5	100	100	99.83	100	100	100
30	6	4	95.83	96.67	93.5	91.33	93	100	100	100	100	100	100
30	6	5	99.83	99.83	100	100	100	100	100	100	100	100	100
30	6	*Avg.*	76.1	75.23	72.57	71.87	70.9	86.13	89.33	88.53	90.6	92.17	93.83

Table 5.4.: AFDE of all Test Scenario Families with Six EHs

Table 5.4 (Page 87) and Table 5.5 (Page 88) list the AFDE values for all test suite families and the testing strengths from $t = 1$ to $t = 5$. Column `P` denotes the number of parameters, column `E` denotes the number of EHs, and column `t` denotes the testing strength. `Avg.` depicts the average AFDE among all testing strengths. The remaining columns follow the pattern `V-I` with

P	E	t	1-1	1-2	1-3	1-4	1-5	2-2	3-3	2-1	3-1	4-1	5-1
6	6	1	36	33.17	34.83	34.17	33.83	47.5	53.33	45	53	63.33	65.83
6	6	2	55.33	53.83	53	53.33	52.33	79.5	94.67	86.83	97	99.5	99.83
6	6	3	71.83	71.5	73.33	70.83	70.67	98	99.83	99.17	100	100	100
6	6	4	86.67	86.67	86.5	85.33	86.33	100	100	100	100	100	100
6	6	5	95	96	96.33	97.33	98.33	100	100	100	100	100	100
6	6	*Avg.*	68.97	68.23	68.8	68.2	68.3	85	89.57	86.2	90	92.57	93.13
12	12	1	17	16	16.58	16.5	16.67	23.08	26.33	25.42	32.08	38.75	44.5
12	12	2	29.5	30	27.83	26.83	26.92	46.58	54.17	54.08	74.75	89.58	95.83
12	12	3	41.92	39.5	38.08	36.75	36.08	64.92	78.33	79.67	98.25	100	100
12	12	4	52	49.5	46.83	45.67	45.17	81.42	96.75	95.58	100	100	100
12	12	5	61.33	57.83	55.33	55.08	54.75	94.67	100	99.92	100	100	100
12	12	*Avg.*	40.35	38.57	36.93	36.17	35.92	62.13	71.12	70.93	81.02	85.67	88.07
18	18	1	11	11.17	11.28	11	10.72	15.78	18.61	17.11	23.22	27.94	32.78
18	18	2	22.33	19.72	19.28	18.5	17.5	31.83	36.67	39.22	58.94	72.67	84.72
18	18	3	28.61	27.72	26	24.83	24.22	44.78	54.78	58.11	84.56	98.78	99.83
18	18	4	36.39	33.83	32.44	31	30.83	58.56	70.94	74.33	98.5	100	100
18	18	5	42.67	40.11	38.72	37.56	36.56	70.28	86.5	90.17	100	100	100
18	18	*Avg.*	28.2	26.51	25.54	24.58	23.97	44.24	53.5	55.79	73.04	79.88	83.47
24	24	1	8.25	8.38	8.46	8.04	8.63	11.79	13.38	11.75	16.71	21.08	32.06
24	24	2	16.46	14.96	14.08	14.58	13.75	23.29	29.33	31.04	47.88	63.25	87.5
24	24	3	22.08	19.67	19.88	19.29	19.13	34.46	41.33	49	80.75	98.08	99.94
24	24	4	27.83	26.54	24.71	24.38	23.67	45.04	54.71	72.79	99.33	100	100
24	24	5	33.92	31.21	29.21	28.79	28.58	54.75	69.29	92.58	100	100	100
24	24	*Avg.*	21.71	20.15	19.27	19.02	18.73	33.72	40.79	43.37	64.3	73.78	79.32
30	30	1	6.3	6.93	6.3	6.47	6.83	9.07	10.8	10.3	13.5	17	19.73
30	30	2	13.57	12.1	11.8	11.47	10.93	19.53	22.53	23.57	35.73	48.87	60.5
30	30	3	18.03	16.67	15.6	15.73	15.07	27.8	34.17	36.17	57.87	77.53	92.93
30	30	4	23.1	20.77	19.77	19.3	19.03	35.77	44	47.47	75.2	96.6	100
30	30	5	27.13	24.97	23.67	23.07	22.8	44.3	53.7	58.57	91.87	100	100
30	30	*Avg.*	17.63	16.29	15.43	15.21	14.93	27.29	33.04	35.21	54.83	68	74.63

Table 5.5.: AFDE of all Test Scenario Families with Increasing No. of EHs

V describing the number of valid values and I describing the number of invalid values. Table 5.4 (Page 87) contains the AFDE values for test scenario families with six EHs. In Table 5.5 (Page 88), the number of EHs equals the number of parameters.

As discussed in the prior subsection, increasing only the number of parameters has no clear effect on the FDE. The AFDE metric reflects this as depicted by Table 5.4 (Page 87).

Increasing the total number of EHs had no impact on the FDE of existing EH indices. But the FDE for the additional locations is worse because more parameters belong to the effective prevention sub-condition. Since AFDE represents the average FDE among all locations of incorrect EHs, the worse FDE of additional EHs decreases the AFDE value. This is depicted in Table 5.5 (Page 88).

Changing the number of values per parameter has a great effect on the AFDE. Depending on whether the additional values are valid or invalid, the AFDE increases or decreases.

The AFDE always improves when adding only valid values. For the largest test scenario

$P = 30$ and $E = 30$ with four and five valid values, the testing strengths $t = 4$ and $t = 5$ are sufficient to detect an incorrect EH almost always. Although, testing with $t = 5$ can be perceived as impractical as it would require the execution of 23369 (4-1) and 58468 (5-1) test cases (See Table 5.1, Page 83).

This result is also consistent with the characteristics of valid values when applying the t-factor fault model. The FDE is improved because more MFCSs are created by the additional valid values.

In another context that is not concerned with CRT, the same general observation has been made by XU et al. [Xu+16] when analyzing MFCSs of the Siemens tool suite: "The high strengths of MFCSs makes it difficult to cover MFCSs by a lower-strength combinatorial test suite. But the big number of MFCSs for each fault makes that it is easy to cover at least one MFCS."

Besides the two favorable cases with (4-1) and (5-1), $t = 5$ is not sufficient to detect all EHs reliably. For 2 valid values (2-1) and $E = 18$, the AFDE is 90.17%. A higher testing strength would be necessary which further increases the time for test input generation and the time for test execution due to the larger test suite size.

There is also a trend indicating that the AFDE decreases when adding only invalid values. For instance, the average AFDE for $P = 30$ and $E = 6$ decreases from 76.1% for one valid and one invalid value (1-1) to 70.9% for five invalid values (1-5). However, this trend is not as clear as for valid values. One exception exists for $P = 6$ and $E = 6$ where the average AFDE improves by 0.57 (1-3) and 0.1 (1-5) percentage points.

But the trend is also consistent with the characteristics of invalid values when applying the t-factor fault model. The parameter of one invalid value belongs to the activation sub-condition and improves the probability of detecting the fault. But all other invalid values either deteriorate the probability because their respective parameter belongs to the effective prevention sub-condition or the invalid value has no effect.

When adding valid and invalid values equally (2-2 and 2-2), the AFDE always increases compared to 1-1. The AFDE is also always higher when comparing it to test scenario families with more invalid than valid values. But the AFDE is always lower compared to test scenario families with more valid than invalid values.

This finding is still consistent with the t-factor fault model. Although, it cannot be directly derived from it. The numbers indicate that the additional MFCSs introduced by additional valid values have a stronger effect on the AFDE than the effective prevention sub-conditions increased by additional invalid values.

To summarize the findings, CT can detect all t-factor faults as guaranteed by t-wise coverage. Beyond that, even more faults are detected via collateral coverage. Test suites that satisfy higher testing strengths $t \geq 3$ trigger many faults in EH at later locations.

Adding valid values to the test suite increases the AFDE. But the required test suite size increases as well. Conversely, adding invalid values decreases the AFDE. Adding new parameters with EH has no impact on the FDE for existing EH. But the faults in the additional EH are harder to detect. Hence, AFDE deteriorates with an increasing number of parameters involved in EH.

The experiment shows that CT can be an effective approach to detect robustness faults. Although, not all robustness faults can be detected reliably. Depending on the number of EH and

the distribution of valid and invalid values, a high testing strength is necessary which requires the execution of large test suites.

To the best of our knowledge, this experiment is the first that assess the effect of invalid input masking on combinatorial test suites. Although, only invalid values are considered. When considering invalid value combinations, the number of parameters involved in fault conditions and MFCSs likely increases and even higher testing strengths would be required. To improve the reliability of CT and to improve the efficiency of CT, extensions like CRT are useful additions worth investigating.

5.1.5. Threats to Validity

The effectiveness of CT in the presence of invalid input masking is evaluated. Therefore, test inputs are generated and executed on test scenarios. Publicly available test suites are used to avoid bias in the test input selection.

The used test scenarios are artificial and do not necessarily represent realistic scenarios. In addition, it is possible that we unconsciously designed the test scenarios in a way that their pre-established characteristics are supported. However, the considered characteristics are explicit and all data is made available in our Appendix A (Page 279 and following) so that it is comprehensible and repeatable. In addition, the AFDE metric is designed to derive knowledge and to apply it to real-world scenarios.

To prevent falsified results due to accidental fault detection, the parameters and values of each test suite are randomized and 100 variants of each test suite are combined to a test suite family. All presented numbers are average numbers.

5.2. A Classification of Robustness Fault Characteristics

The previous section has demonstrated that extensions like CRT are useful additions to CT. In order to construct a CRT test method, we derive a classification of robustness fault characteristics as a next step. It encompasses the hypothesized fault types that our CRT test method focuses on.

Therefore, we link the activities of EH to faults that are hypothesized and observed by related work. Based on the previous discussion of EH and EHMs (Subsection 3.2.2, Page 43 and following) and the related work, we group robustness faults into two categories *configuration-independent* and *configuration-dependent robustness faults*.

The related work that we use consists of studies concerned with issues related to EHMs. Moreover, it is concerned with categorizing and detecting EH-related robustness faults [MT97]; [BDT06]; [Coe+08b]; [MCK12]; [Mel+13]; [BGB14]; [ECS15]; [SCK16]; [PS17a]; [Li+18]; [Che+19]. We do not claim that the selection of related work nor that the distinction of configuration-dependent and -independent robustness faults are complete. However, this classification and the selected related work show that robustness faults of both types are prevalent and should be detected.

5.2.1. Configuration-independent Robustness Faults

A configuration is part of the input abstraction (Definition 13, Page 21) that allows to adapt a program and to customize a program for different needs. Hence, configuration-independent robustness faults are present in each configuration of a program. They are activated by invalid schemata that initiate the corresponding incorrect EH. Configuration-independent robustness faults may occur in all activities of EH as illustrated in Figure 3.6 (Page 49). In the following, faults of each activity are discussed in more detail.

Fault in Exception Detection   The first activity of EH is to detect the occurrence of an exception. Faults in this activity are limited to the detection of program-defined exceptions because predefined exceptions are automatically detected by hardware, operating system, or the run-time environment of the program.

For program-defined exceptions, the developer must define exception conditions and computer instructions to detect exception occurrences. Thereby, mistakes may happen and faults may be introduced. The exception condition that is used for the detection may be too strong, too weak, or completely missing. Virtually all types of faults associated with logic expressions may appear in exception conditions. Examples of fault types are variable reference faults where a variable is replaced by another variable, incorrect relational operators, or incorrect parentheses (cf. [VTP94]; [WGS94]; [Kuh99]).

In the case of too strong exception conditions, testing with truly-invalid inputs is not necessary because truly-valid inputs would incorrectly satisfy the exception condition. In the literature it is classified as *excessive throwing condition* [BGB14] and *exception that should not have been thrown* [ECS15]. This is an issue of correctness and not robustness since the incorrect exception condition prevents the program from fulfilling the functions as defined by the standard specification.

In case of too weak exception conditions or completely missing exception conditions (which can be understood as a special case of too weak exception conditions), testing with truly-invalid inputs that *should* satisfy the exception condition is necessary. In the literature it is classified as *missing throwing condition* [BGB14] and *error in the exception assertion* [ECS15].

Fault in Exception Propagation   The second activity of EH is to propagate a detected exception occurrence. Faults in this activity are again limited to program-defined exceptions because predefined exceptions are automatically propagated.

For program-defined exceptions, the developer must define an exception representation that describes the exception occurrence and propagate it. Mistakes may introduce faults and either the exception representation may contain incorrect information or the propagation of the exception representation may be missing. In the literature, incorrect information in the exception representation is classified as *error in the definition of exception class* [ECS15], *information swallowed* [BGB14], *uninformative or wrong error message* [BGB14]; [Che+19], and *wrong encapsulation of exception cause* [ECS15]. Missing propagation is classified as *exception not thrown* [ECS15].

More sophisticated EHMs rely on dedicated types for exception representations. Then, incorrect information of exception representations can mean that a wrong types is chosen. For instance, a `NullPointerException` is propagated in a Java program when a `IllegalArgumentException` would be correct. As a result, not the intended but another or no exception handler will be selected and executed. In the literature it is classified as *throwing wrong type* [BDT06]; [BGB14], *wrong exception thrown* [ECS15], and *wrong exception class* [Che+19].

When choosing a type that is too generic, the corresponding exception handler must also be generic and may not be able to recover adequately. In the literate it is classified as *uninformative generic type thrown* [BGB14] and *throws generic* [PS17a].

Another type of fault can occur when exceptions that may be propagated are nor documented and the client does not expect them. In the literature it is classified as *inconsistency between source code and application programming interface (API) documentation* [ECS15] and *uncaught run-time exceptions undocumented* [SCK16].

Fault in Exception Handler Selection   The third activity of EH is the selection of an exception handler after an exception occurrence has been detected and propagated. Therefore, an exception handler is associated with a guarded region that it "protects" and exceptions for which it provides recovery instructions.

The selection itself is typically determined by the EHM and can be considered correct. However, the selection is based on the exception handlers and their associations. A fault in the exception handler selection is caused by incorrect associations.

One cause of an incorrect association can be a guarded region that is defined too narrow such that an exception handler is not selected when it should be selected. In the literature it is classified as *fragile catch bug pattern* [Coe+08b] and *unreachable handler* [PS17a]. We understand the entire absence of an exception handler as a special case of a too narrow association. It is one of the most common failures of EH [Coe+08b] and in the literature it is classified as *uncaught*

exception [Coe+08b]; [BGB14], unhandled exceptions [PS17a], *throw without catch bug pattern* [Coe+08b], and *lack of a handler that should exist* [ECS15].

One cause of an incorrect association can be a guarded region that is defined too wide such that it is selected for a propagated exception when it should not be selected. In literature it is classified as *overly protective try-block fault* [BGB14].

It is closely related to another incorrect association where an exception handler is associated with too many exceptions or with a too generic exception type. In the literature it is classified as *exception stealer* [Coe+08b], *overly-generic catch-block* [BGB14], *exception caught at the wrong level* [ECS15], and *over-catch* [PS17a].

An exception handler with an association that is too generic is selected for too many propagated exceptions. The too generic association causes the exception handler to be selected when it should not be selected [Coe+08b]; [BGB14]. It catches the propagated exception before the intended exception handler could be selected. This problem is also known as the *unintended handler action* [MT97] and faults are considered as being very difficult to diagnose [Coe+08b] and harmful [SCK16].

Fault in Exception Recovery   The fourth activity of EH is exception recovery which is invoked after the correct exception handler has been selected.

The unintended handler action is not the only problem of exception handlers that are associated with too many or too generic exceptions. As a consequence of too many or too generic exceptions, an exception handler can only provide generic responses and may not be able to recover adequately. In the literature it is classified as *general catch block* [ECS15], *catch generic* [PS17a], *catching generic exceptions* [Li+18], *centralized handler* [Bez+19], *overly-general reaction* [Che+19].

Since an exception handler consists of computer instructions, it may be incorrect and contain faults. In the literature it is classified as *error in the handler* [ECS15], *incomplete implementation* [PS17b], and *incorrect reaction logic* [Che+19].

Besides incorrect computer instructions, another general issue may be the absence of computer instructions. In this case, an exception handler is correctly selected and invoked but the exception handler does not contain any computer instructions for recovery. In addition, the exception occurrence is ignored. In the literature it is classified as *swallowed exception* [BGB14], *empty catch block* [ECS15], *ignoring exceptions* [SCK16]; [Li+18], *catch and do nothing* [PS17b], and *missing reaction* [Che+19].

Another issue is related to exception handlers that return NULL or propagate another exception representation. Thereby, the original exception occurrence is obfuscated and the diagnosis is complicated. In the literature it is classified as *catch and return null* [PS17b], *destructive wrapping* [PS17b], [SCK16], and *exception remapping* [BGB14], [SCK16]. A special case is classified as *suppressed exception* [BGB14] and *throw within finally* [PS17b] where the original exception occurrence is suppressed by another exception occurrence.

Another special case is the *premature termination* [BGB14] where the return of NULL or the repeated propagation is premature and results in the termination of the program when retrying the original operation would be a better option.

On the contrary, *missing termination action* [BGB14] refers to an issue where a (strict) termination is expected but the computer instruction is missing.

Another issue is related to exception handlers that also do not recover but log the occurrence of an exception. In the literature it is classified as a *dummy handler* [PS17b]. A special case of this issue is classified as *log and throw* [PS17b], *logging and returning null* [Li+18], and *log and return null* [PS17b], [SCK16].

A closely related issue refers to exception handlers that are correctly selected and invoked but that logged exception occurrences insufficiently. In the literature it is classified as *missing log* [BGB14] [BDT06] and *multi-line log* [PS17b].

Fault in Exception Continuation   The last activity of EH is exception continuation. It is concerned with performing the standard behavior once an exception handler has been executed successfully and once the program's state has been recovered. The continuation is typically done by the EHM and in most cases the successful recovery is synonymous with a strict termination and an error message. Therefore, no severe faults related to this activity are expected. But two issues exist that can be related to exception continuation.

First, exception continuation may execute computer instructions that are not intended to be executed right after a successful exception recovery. In the literature it is classified as *wrong location of execution resumption* [BGB14]

Second, computer instructions to clean-up and deallocate resources are important to prevent resource and memory leaks. Often, these computer instructions are written once and should be executed with and without the occurrence of an exception. For that purpose, special language features exist in many languages which may be used inappropriately. In the literature it is classified as *resource leak fault* [BGB14], *error in the clean-up action* [ECS15], *lack of a finally block that should exist* [ECS15], and *catch block where only a finally would be appropriate* [ECS15].

5.2.2. Configuration-dependent Robustness Faults

The aforementioned robustness faults are related to specific activities of EH. Furthermore, they are present in every configuration of a program. Besides them, robustness faults may exist that are not present in every configuration of a program. In the following, we discuss configuration-dependent robustness faults and again link them to faults that are hypothesized and observed by related work.

Configuration-dependent robustness faults must also be activated by truly-invalid schemata that initiate the corresponding incorrect EH. But in addition, certain configurations of the program are required. Many modern programs like JHipster, Apache, or GCC are highly configurable [CDS08]; [Hal+19]. Features can be enabled or disabled and different implementations of the same feature can be exchanged via configuration parameters at compile-time or at run-time. Typically, these programs can be divided into a common part which is relevant to all configurations and variable features which are relevant only for certain configurations. Therefore, different implementation techniques exist [AG01]. Examples are *conditional compilation* where source code regions are enabled or disabled at compile-time, *dynamic class loading* where classes are chosen and loaded at run-time, *polymorphism* and *design patterns* where implemen-

```
class CurrentAccount implements Account {
  void withdraw(int amount) {
    if( (balance - amount) < limit )
      throw new AccountNotCoveredException();
    ...
  }
}
```
Robustness fault in exception detection of variable feature
```
class SavingAccount implements Account {
  void withdraw(int amount) {
    if( (balance + amount) < 0 )
      throw new AccountNotCoveredException();
    ...
  }
}
class CoreBankingSystem {
  void withdraw(int accountId, int amount) {
    try {
      Account acc = findAccount(accoundId);
      acc.withdraw(amount);
    } catch(AccountNotCoveredExc. ex) { ... }
  }
}
```

Figure 5.6.: Illustration of Configuration-dependent Robustness Fault

tations are exchanged at run-time, or *aspect-orientation* where meta-programming is used to modify and inject additional instructions at run-time.

In general, the configuration-independent robustness faults may appear in variable features as well. Then, not only the truly-invalid schema but also a specific configuration that enables the variable feature is required. In addition, configuration-dependent robustness faults may be caused by interactions of configuration parameters [CDS08].

Research on EH in configurable programs shows that the activities of EH are scattered across the common part and variable features [MCK12]; [Mel+13]. Variable features are involved in propagating exception occurrences, handling propagated exceptions or they are intermediaries between exception detection and exception recovery. For most propagated exceptions, exception handlers are selected that reside in the same variable feature or in the common part of the program.

But a significant number of exceptions propagated in variable features are not recovered appropriately [Mel+13]. For exceptions propagated by one variable feature, MELO et al. [Mel+13] observed the selection of no exception handler or exception handlers of other distinct variable features with no association between the two variable features. In the latter case, the exception handler was mistakenly selected had not enough information to properly recover from the exception occurrence. This is another instance of the unintended handler action (cf. [MT97]) where the propagated exception is catched before the intended exception handler could be selected.

For further illustration of configuration-independent and configuration-dependent robustness faults, Figure 5.6 depicts an example. A core banking system has a common part which provides a service to withdraw money from an account. Different implementations of the withdraw functionality exist which are considered variable features. Depending on the account type, different implementations are chosen at run-time.

First, assume `findAccount` propagates a `AccountNotFoundException` for non-existent accounts, the core banking system would crash because no appropriate exception handler exists. This is caused by a configuration-independent robustness fault because it is present in every configuration of the program and can be activated with each non-existent account.

Second, assume a robustness fault in the `SavingAccount` feature where the exception condition is incorrect, i.e. plus is used instead of minus. This is caused by a configuration-dependent robustness fault because it can be activated with the combination of a certain truly-invalid schema, i.e. the withdraw amount exceeds the account limit, and a certain configuration that enables the `SavingAccount` feature.

5.3. A Case Study on Minimal Failure-causing Schemata

The previous section provides a classification of robustness fault characteristics. In order to develop a CRT test method that detects robustness faults with these characteristics, we now conduct a case study to gain insights from existing ordinary failures and robustness failures in order to draw conclusions regarding the degrees of minimal failure-causing valid and invalid schemata.

For the sake of simplicity, we again assume that the value domains of a hypothetical IPM contain all values necessary for the propagation of analyzed failures.

5.3.1. Motivation

CT can help detect failures that are triggered by the interaction of two or more parameter values. For instance, a bug report analyzed by WALLACE & KUHN [WK01] describes that "the ventilator could fail when the altitude adjustment feature was set on zero meters and the total flow volume was set at a delivery rate of less than 2.2 liter per minute". The failure is triggered by the interaction of `altitude` $= 0$ and `delivery-rate` < 2.2. Because of the interaction of two parameter values, a MFCS of degree $d = 2$ is required for its detection.

Testing each value only once is not sufficient to detect MFCSs with a degree $d > 1$ and exhaustively testing all interactions among all input parameters is almost never feasible in practice. Therefore, CT pursues an alternative approach.

If the highest degree of parameters involved in MFCSs was known before testing, i.e. if all failures of a SUT are triggered by an interaction of d or fewer parameter values, then testing all d-wise parameter value combinations should be as effective as exhaustive testing [KWG04].

However, d cannot be determined for a SUT a-priori because the faults are not known before testing. Hence, the motivation of related empirical studies is to derive common knowledge about the MFCS degrees and its frequency of occurrence to guide future test activities.

Existing research on the effectiveness of CT derives the distribution and maximum MFCS degree among different types of programs based on bug reports. WALLACE & KUHN [WK01] review 15 years of recall data from medical devices, i.e. software written for embedded systems. KUHN & REILLY [KR02] analyze bug reports from two large open-source software projects; namely the Apache web server and the Mozilla web browser. KUHN & WALLACE [KWG04] report findings from analyzing 329 bug reports of a large distributed data management system developed at NASA Goddard Space Flight Center. BELL & VOUK [BV05] analyze the effectiveness of pairwise testing a network-centric program. They derive their characteristics from a public database of security flaws and create simulations based on that data. COTRONEO et al. [Cot+16] again analyze bug reports from Apache and the MySQL database system and RATLIFF et al. [Rat+16] report the MFCS degrees of 242 bug reports from the MySQL database system.

As concluded by KUHN et al. [KKL16], the studies show that most failures in the investigated domains are triggered by single parameter values and parameter value pairs. Progressively fewer failures are triggered by an interaction of three or more parameter values. In addition to the distribution of MFCS degrees, a maximum interaction of four to six parameter values is identified. No reported failure required an interaction of more than six parameter values to be triggered.

Thus, pairwise testing should trigger most failures and 4- to 6-wise testing should trigger all failures of a SUT.

So far, research focuses on *positive* test scenarios with valid test inputs. To the best of our knowledge, it is only PAN et al. [Pan99]; [PKS99] who characterize data of failures from testing with invalid test inputs. Although in contrast to the aforementioned studies, their results are not obtained from analyzing bug reports. Instead, the results are obtained from testing the POSIX APIs of 15 operating systems with the Ballista robustness testing tool. The results indicate that most (81.75%) robustness failures are caused by single invalid values, i.e. MFCSs of degree $d = 1$. Unfortunately, there is no more information on robustness failures that are caused by MFCSs with higher degrees. According to a later publication [KD00], robustness failures of type *silent* or *hindering* are not detected and considered by the study.

Also, as KUHN et al. [KWG04] state, more empirical studies are required to confirm or reject the distribution and upper limit of MFCSs for other types of programs. Therefore, we conduct this case study.

5.3.2. Case Study Design

Research Method

To gather more information on robustness failures triggered by invalid value combinations, we conduct a case study to analyze bug reports of a newly developed distributed enterprise application for financial services. Therefore, we follow the guidelines for conducting and reporting case study research in software engineering as suggested by RUNESON & HÖST [RH09]. As they state, a case study "investigates a contemporary phenomenon within its real life context, especially when the boundaries between phenomenon and context are not clearly evident". Case study research is typically used for exploratory purposes, i.e. seeking new insights and generating hypotheses for new research, and explanatory purposes, i.e. seeking explanations of situations or problems which involves testing of existing theories [RH09].

The guidelines suggest conducting a case study in five steps. First, the objectives are defined and the case study is planned. As a second step, the data collection is prepared before the data is collected in a third step. Afterwards, the collected data is analyzed and finally, the results of the analysis are reported.

Research Objective

The overall objective of this case study is to gather information on MFCSs to compare the results with the ones of other published case studies and to identify MFCSs that trigger robustness failures in order to provide further information for the development a combinatorial robustness fault model.

The work by PAN et al. [Pan99]; [PKS99] indicates that most robustness failures in operating system APIs are triggered by single values rather than by value combinations, i.e. a MFCS degree of one, Hence, our aim is to either confirm or reject this indication for enterprise applications. By conducting this case study, the indication cannot only be confirmed or rejected but the indication can also be enriched with additional information.

Therefore, the objective of this case study is to extract MFCSs from the bug reports and to identify their degrees, their maximum degree, and their distribution.

Further on, the objective is to classify MFCSs as either positive or negative depending on whether or not the failure is a robustness failure.

If the MFCS is classified as negative, a second objective is to identify the invalid schema that is covered by the MFCS and to derive their degrees, maximum degree and distribution as well.

Case and Unit of Analysis

The case is a software development project from an IT service provider for an insurance company. A new system is developed to manage the life-cycle of life insurances. It is based on an off-the-shelf framework which is customized and extended to meet the company's requirements. In total, the new system consists of 2.5 MLOC and an estimated workload of 5000 person days. The core is an inventory sub-system with a central database to store information on customer's life insurance contracts. In addition, complex financial calculation engines and business processes like capturing and creating new customer insurances are implemented. The business processes also integrate with a variety of already existing systems which are, for instance, responsible for managing information about the contract partners, about claims and damages, and to the support insurance agents.

Since life insurance contracts have decade-long lifespans and rely on complex financial models, the correctness of the system is business critical. Mistakes can have severe effects which can even amplify over the long-lasting lifespans and cause enormous damage to the company. Therefore, thorough testing is important.

Even though the business processes are managed by the new system, they rely on other systems of which each system again relies on other systems. This makes it hard to test the system or its parts in isolation. It is also difficult to control the state of the systems and to observe the complete behavior which makes testing even more complicated.

Therefore, most testing is conducted on a system level within an integrated test environment in which all required systems are deployed. The test design is often based on experience and error-guessing. Tests are executed mostly manually because of the low controllability and observability.

Data Collection Procedure

To answer the research objectives, the case study relies on archival data from the aforementioned software development project. A project-wide issue management system contains all bug reports from the project start in 2015 to the productive deployment at the beginning of 2018. In general, a bug report is a specifically categorized issue which coexists with other project management- and development-related issues.

For our case study, we analyzed the issue's title, its category, its initial description, additional information in the comment section and its status. Further on, some bug reports are also connected to a central source code management system. If a bug report does not contain sufficient information, the corresponding source code modifications can be analyzed as well.

The issues are filtered to restrict the analysis to only reasonable bug reports. Therefore, issues created automatically by static analysis tools are excluded. Further on, only issues categorized as bug reports whose status is set to *complete* are considered because we expect only them to contain a correct description on how to systematically reproduce the failure.

Data Analysis Procedure

Once the bug reports are exported from the issue management system, each bug report is analyzed one at a time. First, it is checked if the bug report describes a failure in the sense that an incorrect behavior is observable by the user. Otherwise, the bug report is rejected.

Afterwards, the trigger type of the reported failure is determined. A trigger is a set of conditions that exposes a failure when all conditions are satisfied and different types of triggers exist [Cot+16]. In the context of CRT, we focus on variations and different constellations of values for the same parameters which are in particular stimulus, configuration, and state spaces of a program which we abstract as inputs (Definition 13). Trigger that do not depend on inputs but rather on sequences of operations, their ordering, or timing of operations are excluded from further analysis. The bug report is also not further analyzed if no systematically reproducible trigger is found.

If a specific value or value combination is identified to trigger the failure, the degree of the MFCS is determined in the next step.

Then, the bug report is classified as either *positive* or *negative* depending on whether any invalid schema is covered. If it is classified as negative, the degree of the invalid schema is identified as well.

5.3.3. Results and Discussion

Analyzed Data

In total, 683 bug reports are analyzed. All reported bugs are revealed and fixed during the development phase of the system. Even though filters are applied to export the bug reports, 249 bug reports are classified as *unrelated*, because the issue management system is also used as a communication and task management tool. For instance, problems with configurations of test environments, refactorings or build problems are categorized as bug reports as well.

The remaining 434 bug reports describe failures, they are classified as follows. Eight bug reports do not provide enough information for further analysis and classification. 38 reported bugs require specific timing and ordering of sequences to be triggered. For instance, one sequence to trigger a failure is to search for a customer, open its details, edit the birthday, press cancel and edit the birthday again. Three reported bugs are related to robustness testing. They are triggered by other systems that timeout and do not response to requests. All these bug reports are excluded from further analysis because CRT is about varying test inputs rather than varying sequences and timing.

The remaining 388 bug reports describe failures triggered by some test inputs. A subset of 176 bug reports describes integration failures with other systems where values are not correctly mapped from one data structure to another. They can be triggered by *any* test input. The use

d	All	Positive	Negative
1	162	121	41
2	40	31	9
3	6	5	1
4	4	4	-
5	-	-	-
6	-	-	-

Table 5.7.: Observed MFCS Degrees

d	Previous Studies								Our Study		
	(A)	(B)	(C)	(D)	(E)	(F)	(G)	Avg.	All	Pos.	Neg.
1	66	28	41	67	18	9	49	39.7	76	75	80
2	97	76	70	93	62	47	86	75.9	96	94	98
3	99	95	89	98	87	75	97	91.4	98	98	100
4	100	97	96	100	97	97	99	98.0	100	100	
5		99	96		100	100	100	99.0			
6		100	100					100.0			

Table 5.8.: Cumulative Percentage of MFCS Degrees

of different date time formats is another cause. There are so many reported integration issues (45% out of 434 bug reports) because the system consists of several independently developed components which are early and often integrated with the other components and other systems using one of the test environments.

Finally, 212 bug reports are considered to be suitable for CT, which is 49% of all 434 bug reports that describe failures which is 55% of all 388 bug reports that are triggered by some test input To reproduce one of the bugs reported, the test input requires at least one specific value.

Observed MFCS Degrees

The observed MFCS degrees for the 212 bug reports are depicted in Table 5.7. Overall, the numbers confirm the conclusion made by Kuhn et al. [KKL16]. Most failures are triggered by single parameter values and parameter value pairs and progressively fewer failures are triggered by 3-wise and 4-wise interactions. In our case, no reported bug requires an interaction of more than 4 parameters in order to trigger the failure.

Table 5.8 presents the cumulative percentage of the MFCS degrees. The last three columns refer to our case study and show that 76% of all reported failures require 1-wise (each choice) coverage to be reliably triggered. It adds up to 96% when testing with pairwise coverage and 100% are covered when all 4-wise parameter value combinations are used for testing.

To compare our results, the first columns of the table show the results of previous case studies, briefly introduced in the related work section. The numbers and also the average percentage values are taken from KUHN et al. [KKL16]. The first case (A) refers to WALLACE & KUHN [WK01] analyzing bug reports of medical devices. The second case (B) and the third case (C)

refer to KUHN & REILY [KR02] analyzing bug reports of the Apache web server and Mozilla web browser, respectively. Bug reports of a large distributed data management system are analyzed in the fourth case (D) [KWG04]. The fifth case (E) analyzes security flaws from a public database [BV05]. Case six (F) again analyzed Apache web server and the MySQL database system [Cot+16]. Though, the numbers cannot directly be derived from the case study. The seventh case (G) analyzes bug reports of the MySQL database system [Rat+16]. The distribution of MFCS degrees obtained in our case study is not in contradiction to the other cases. However, the distribution is mostly similar to cases [WK01] and [KWG04]. While there are no obvious similarities with embedded systems for medical devices [WK01], the large data management system [KWG04] is probably quite similar to our case in terms of requirements and used technologies. Similar to our case, the bug reports are also from a development project whereas the other studies analyze fielded products [KKL16]. For all three cases, most failures are triggered by single parameter values and almost all failures are triggered by the combination of single parameter values and pairwise parameter value combinations. All failures should be triggered by 4-wise parameter value combinations.

So far, only the degree of failure-inducing interactions is considered but differences between positive and negative scenarios are not discussed. Table 5.7 depicts the observed MFCS degrees and their distribution divided into positive and negative test scenarios. As can be seen, the maximum degree of MFCSs that trigger robustness failures is three. Compared to positive test scenarios, fewer negative scenarios are discovered and the MFCS dimensions are also lower. For single parameter values and parameter value pairs, the ratio is 3:1 of valid vs invalid test inputs and no invalid test inputs are identified for higher degrees.

While these numbers indicate that most failures are triggered by valid test inputs, we emphasize that the test design is based on experience and error-guessing, robustness testing was not in the focus. Hence, the ratio can also result from a general bias towards testing of positive scenarios which is identified in research [Lev+93]; [Tea+94]; [Cau+13].

Nevertheless, these findings underpin the results of Pan et al. [Pan99]; [PKS99] who observe that most robustness failures in operating systems APIs are triggered by single invalid values. In their study, 81.75% of robustness failures are triggered by single invalid values. We observe the same ratio in our case.

Besides the quantitative assessment, qualitative information can be extracted as well. All in all, 51 robustness failures are identified which are classified as follows. 22 failures are caused by *incorrect exception detection* where conditions to detect exception occurrences are either too weak or missing. Consequently, these occurrences are not discovered. For instance, bank transfer is accepted as a payment option even though incorrect or no bank account information is provided.

In 19 cases, reported failures are caused by *incorrect exception propagation* where an exception occurrence is correctly detected but incorrectly propagated. As a consequence, the correct exception handler cannot be selected. For instance, a misspelled first name is detected by a user registration service but the error message complaints a misspelled last name.

In seven cases, failures are triggered by the system's run-time environment. For instance, a `NullPointerException` is propagated when the run-time environment detects attempts to access `NULL` values. Since developers mistakenly did not anticipate `NULL` values, no respective

exception handlers are implemented and, as a result, the default handler is incorrectly selected and terminates the process. These failures denote incorrect *exception handler selections*.

For three reported failures, the exception occurrence is correctly detected and propagated. However, the system performs *incorrect exception recovery* because the instructions to recover from the exception occurrence contain faults. For instance, the user is asked to correct wrong input, e.g. a misspelled first name. After the input is corrected, the system does not recover and the corrected input cannot be processed.

Regarding the *exception continuation* activity, no faults are found. On the one hand, this may be due to the fact that the Java run-time environment takes care of the exception continuation and automatically follows the termination model without additional effort from developers. On the other hand, this may be due to the fact that the analyzed program follows the fail fast principle and terminates with an error message returned to the user.

The bug reports also demonstrate the importance of strong invalid test inputs, i.e. test inputs with exactly one invalid value or exactly one invalid value combination. For instance, the component that manages contracting parties ensure data quality by checking that, e.g. the title of a person matches the gender of the first name and that the first name and family name are correct and not confused with each other. However, when using an unknown invalid title, the system responds with a wrong error message saying that the family name was wrong. If an invalid family name was combined with the unknown title, the failure would not have been discovered.

The 10 invalid test inputs with a MFCS degree greater than one are further analyzed. For two reported bugs, the invalid schema is not identical but rather a subset of the MFCS. Even though a combination of two and three specific parameter values is required to trigger the robustness failures, the invalid schema only has a degree of one and two, respectively.

The two reported bugs describe failures with *robustness interactions* that require an interaction of invalid value (combinations) and a valid value of another parameter.

One reported bug is related to the communication between two systems. The response of the second system contains one parameter value with error information to indicate whether the requested operation succeeded or failed. Another value provides details about the internal processing of the request and a certain value indicates an internal resolution of the error. In that case, the calling system is expected to handle the error in a different way.

The other reported bug belongs to the storage of details on contracting parties where one contracting party must be responsible for paying the insurance premiums. If direct debit is chosen as the payment method but an invalid bank account, i.e. an invalid IBAN number, is provided, the resulting error message remains even after the invalid IBAN is replaced by a valid IBAN. While the combination `[payment-method:direct-debit, account-number:invalid]` is required as a invalid combination, the bug report states that this phenomenon could only be observed for `[role:contributor]`.

Beyond the two reported failures caused by configuration-dependent robustness faults, no further configuration-dependent robustness faults are identified.

Implications for Combinatorial Testing

To summarize the findings of this case study, the observed MFCS degrees of 212 failures could be extracted from bug reports from which 161 bug reports describe *ordinary* failures and 51

describe robustness failures.

In general, the MFCS degrees confirm to the findings of previous empirical studies. But additional observations can be made regarding the MFCS degrees of robustness failures.

The importance of avoiding the invalid input masking effect is again highlighted for valid test inputs and strong invalid test inputs that must not contain more than one invalid schema.

Further on, 49 out of 51 robustness failures are caused by configuration-independent robustness faults that are located in four out of five fundamental EH activities.

For the fifth fundamental activity, which is exception continuation, we did not encounter any robustness fault. We hypothesize that exception continuation in the analyzed case is rather simple because (1) Java has built-in continuation that automatically applies the termination model, and (2) invalid requests are terminated and fail fast with an error message returned to the user.

10 robustness failures are revealed by MFCSs with degrees $d > 1$ which once more demonstrates that the fundamental hypothesis of single error coverage and base-choice coverage, i.e. robustness failures are caused by invalid values, is not sufficient and invalid value combinations should also be taken into account.

The majority of analyzed robustness failures does not indicate any interaction between valid values and invalid values or invalid value combinations. However, two robustness failures could be observed where invalid values or invalid value combinations interact with valid values and valid value combinations. This is in line with aforementioned existing research that most faults are caused by low interactions among parameters. In an analogous manner, we can expect (1) that most robustness faults are activated by single values and are independent from any further configuration and (2) that fewer robustness faults require invalid value combinations and interactions between valid and invalid schemata.

5.3.4. Threats to Validity

A general threat to validity is that case studies are difficult to generalize from [RH09]. Especially, because only one particular type of programs of one company is analyzed. The archival data of the case study is only a snapshot and the ground truth, i.e. the set of all failures that can be triggered, is unknown. Hence, the data set can be biased, for instance, towards positive scenarios which has been observed in research [Lev+93]; [Tea+94]; [Cau+13]. Since the bug reports result from tests based on experience an error-guessing, it may apply here as well.

Further on, the data set can be biased towards *simple* faults and MFCSs with lower degrees as they are easier to trigger. Relevant and reasonable bug reports may be excluded by our filtering because the bug reports are incorrectly categorized. Maybe not all triggered failures are reported. For instance, a developer who finds a fault might just fix it without creating a bug report.

The bug reports can further be incomplete resulting in a wrong classification. For instance, default values of some parameters may be hidden. When those default values are required to trigger a failure, the extracted MFCS does not contain all parameter-value pairs. As a consequence, the MFCS degree is too low and interactions between valid and invalid schemata may remain unnoticed.

All in all, the findings of the case study must be grasped as examples of an incomplete data set. But in combination with the previous sections, two different types of robustness faults can be distilled.

5.4. A Refined t-Factor Fault Model for Robustness Faults

After the deficiencies of CT in the presence of invalid input masking are shown in Section 5.1 (Page 74 and following), robustness fault characteristics are developed (Section 5.2, Page 91 and following) and a case study to analyze MFCSs is conducted (Section 5.3, Page 97 and following). Now, the previous findings are combined into a refined t-factor fault model which then functions as a basis for the development of test adequacy criteria and test selection strategies.

The basic assumption of the t-factor fault model is that faults are caused by the interaction of t parameters [DM98]. The t-factor faults are detectable by certain schemata that have a degree of $d \leq t$ and that satisfy the fault condition of the t-factor fault (MFCSs). To detect t-factor faults, the corresponding t-wise coverage is a test adequacy criterion that ensures to cover each schema of size $d \leq t$ by at least one test input. Thereby, it is assured that the failure-causing schemata with $d \leq t$ are also tested.

Since the subset of failure-causing schemata is not known in advance, it is important to test all schemata with degrees of $d \leq t$ since all of them are potentially failure-causing. The degrees of MFCSs are also not known in advance. Therefore, empirical studies like [WK01]; [KR02]; [KWG04]; [BV05]; [Cot+16]; [Rat+16] and our case study in Section 5.3 (Page 97 and following) help to estimate what is an appropriate testing strength t for a specific SUT.

The controlled experiment in Section 5.1 that was based on the t-factor fault model to describe robustness faults shows that the t-factor fault model also applies to SUTs with EH, invalid input masking, and robustness faults. Although, depending on the number of parameters that are involved in EH and the distribution of truly-valid and truly-invalid schemata, high testing strengths and large test suite may be required with CT and t-wise coverage. Even with high testing strengths, not all robustness faults can be detected reliably. Because of irrelevant and invalid input masking, the number of parameters involved in the effective prevention sub-condition increases with the number of irrelevant combinations and EH. Irrelevant schemata can be excluded from testing to reduce the degrees of corresponding MFCSs. But truly-invalid schemata cannot simply be excluded because it is important to test EH as well. With CT, a higher testing strength and a larger test suite are required resulting in a potentially lower FDE when the number of parameters involved in EH increases. Thus, an alternative approach to reduce the degrees of MFCSs is necessary.

The CRT extension is based on the incorporation of additional semantic information beyond the number of parameters that are involved in the fault condition. To develop alternative test adequacy criteria and test selection strategies, we define a refined t-factor fault model for ordinary faults and robustness faults. It utilizes another perspective that is based on a distinction of truly-valid and truly-strong invalid schemata.

For ordinary faults regarding the standard behavior as well as for robustness faults regarding the exception behavior, faults can only be detected when they are reached, i.e. when the effective prevention sub-condition is satisfied, and when the fault is activated, i.e. when the activation sub-condition is satisfied [LO17]. By refining the fault model to focus on truly-valid and truly-strong invalid schemata, the invalid input masking effect is avoided and the effective and also general prevention sub-conditions are always satisfied.

This is another perspective on the same faults and distinguishes ordinary faults and robust-

ness faults. The perspective imposes additional requirements on test adequacy criteria and test selection strategies. To detect a fault, test inputs that satisfy the activation sub-condition must be truly-valid or truly-strong invalid. Since the invalid input masking effect is excluded, the focus is on the activation sub-condition which determines the number of involved parameters.

To detect an ordinary fault, the fault condition of the corresponding t-factor ordinary fault must be satisfied. A t-factor ordinary fault is an abstraction of an ordinary fault and a truly-minimal failure-causing and truly-valid schema is required to detect it. Since the effective and general prevention sub-conditions are always satisfied by a truly-valid schema, the fault condition and the activation sub-condition are equivalent.

Definition 83 (t-Factor Ordinary Fault) *A t-factor ordinary fault is a particular t-factor fault that defines a fault condition over t parameters of the SUT which must be satisfied by a truly-valid input in order to activate the fault.*

To activate robustness faults, minimal failure-causing schemata are required that are also truly-strong invalid schemata. Two cases can be distinguished.

First, to detect a configuration-independent robustness fault, the minimal failure-causing and truly-strong invalid schema is also a truly-minimal invalid schema. The MFCS is congruent with the truly-minimal invalid schema. Besides the truly-minimal invalid schema, no other schema is necessary to activate the robustness fault. The assignments of all other parameters is of no relevance as long as they do not cause any input masking.

Second, to detect a configuration-dependent robustness fault, the minimal failure-causing and truly-strong invalid schema is not a truly-minimal invalid schema. This is because additional assignments of other parameters are required such that the combination activates the fault. In this case, the MFCS is composed of a truly-minimal invalid schema and another truly-valid schema and only the combination of both is minimal failure-causing.

Both, configuration-independent and -dependent robustness faults can be uniformly described by a (a, b)-factor robustness fault.

Definition 84 ((a,b)-Factor Robustness Fault) *A (a, b)-factor robustness fault is a particular t-factor fault that defines a condition over $t = a + b$ parameters of the SUT that must be satisfied by a strong invalid input in order to activate the fault.*

The fault condition and the activation sub-condition are again equivalent. Because of the interaction between a truly-minimal invalid schema and another truly-valid schema, the fault condition over t parameters is divided into two conditions over disjoint parameters with a and b parameters, respectively. First, the **exception sub-condition** over a parameters constitutes what a truly-strong invalid test input must satisfy to ensure the occurrence of an exception. Second, the **configuration sub-condition** over b parameters constitutes what a truly-strong invalid test input must additionally satisfy to ensure that the fault is reached and the state is infected. The conjunction of the exception sub-condition and the configuration sub-condition constitutes the fault condition over $t = a + b$ parameters.

> **Running Example** Let's again suppose a fault in the ordering web-service implementa-
> tion (Listing 2.1, Page 14). First, let's suppose an ordinary fault in the `processDomes-`
> `ticStandardOrder(packages, phone)` function. The fault can be modeled
> as a t-factor ordinary fault with `processDomesticStandardOrder(packages,`
> `phone)` as its activation sub-condition.
>
> Next, a configuration-independent fault where the function `isInvalidCoun-`
> `try(country)` is incorrect and does not properly handle an exception occurrence.
> The configuration-independent fault can be modeled as a (a, b)-factor robustness fault
> with $a = 1$ and $b = 0$ since only one parameter is involved in the `isInvalidCoun-`
> `try(country)` exception sub-condition and the configuration sub-condition is empty.
>
> In contrast, another configuration-independent fault in the `isInvalidCoun-`
> `tryCode(country, phone)` function would be modeled as a (a, b)-factor robust-
> ness fault with $a = 2$ and $b = 0$ because two parameters are involved in the exception
> sub-condition.
>
> A configuration-dependent fault in the `isInvalidCountryCode(country,`
> `phone)` function that is only activated for express shippings would be modeled as a
> (a, b)-factor robustness fault with $a = 2$ and $b = 1$ because the configuration sub-condi-
> tion `¬isStandardShipping(shipping)` would comprise one parameter.

For the detection of t-factor ordinary faults, a truly-valid test input that covers a corresponding
MFCSs of degree $d = t$ is required. Consequently, the MFCSs for t-factor ordinary faults are
truly-valid schemata as well. Therefore, the extended structure of an IPM for our CRT test
method must provide additional semantic information that allows defining and selecting valid
schemata that are potentially minimal failure-causing.

For the detection of (a, b)-factor robustness faults, a truly-strong invalid test input that covers
MFCSs of degree $d = a + b$ is required. The MFCSs are truly-strong invalid schemata as well
but not necessarily truly-minimal invalid schemata. Therefore, the extended structure of an IPM
for our CRT test method must provide additional semantic information that allows defining and
selecting strong invalid schemata that are potentially minimal failure-causing.

To reflect the exception and configuration sub-conditions, minimal failure-causing truly-in-
valid schemata have two associated properties. These properties are later transferred to the
concept of minimal failure-causing invalid schemata. The degree d of a minimal failure-causing
truly-invalid schema consists of a *robustness degree* a and a *robustness interaction degree* b with
$d = a + b$.

Definition 85 (Robustness Degree) *The robustness degree a of a minimal failure-causing truly-
invalid schema is defined by the number of parameter-value pairs that constitute the truly-mini-
mal invalid sub-schema s_{inv}, i.e. $a = |s_{inv}|$.*

In addition to the truly-minimal invalid sub-schema, the minimal failure-causing truly-invalid
schema may cover another disjoint truly-valid sub-schema with which the truly-minimal invalid
sub-schema *interacts*. Let τ denote the minimal failure-causing truly-invalid schema and let s_{inv}
denote the truly-invalid sub-schema, then $s_{valid} = \tau \smallsetminus s_{inv}$ denotes the truly-valid sub-schema.

Definition 86 (Robustness Interaction Degree) *The robustness interaction degree b of a minimal failure-causing truly-invalid schema is defined by the number of parameter-value pairs that do not constitute the truly-minimal invalid sub-schema. The degree of the truly-valid sub-schema s_{val} constitutes the robustness interaction degree, i.e. $b = |s_{val}|$.*

In the case of robustness failures triggered by single truly-invalid values ($d = 1$), the robustness fault is always configuration-independent. Further on, the minimal failure-causing truly-invalid schema and the truly-invalid schema are congruent, i.e. both contain the same parameter-value pairs. As a consequence, the robustness degree is $a = 1$ and the robustness interaction degree is $b = 0$. For robustness failures that are triggered by truly-invalid value combinations ($d \geq 2$), two cases of configuration-independent and configuration-dependent robustness failures can be distinguished. First, when the truly-invalid value combination is a truly-minimal invalid schema, the minimal failure-causing truly-invalid schema activates a configuration-independent fault, its robustness degree is $a = d$ and there is zero robustness interaction ($b = 0$). Second, when the truly-invalid value combination is not a truly-minimal invalid schema, the minimal failure-causing truly-invalid schema covers a truly-minimal invalid schema and a truly-valid schema, i.e. $b > 0$ and $a = d - b$.

> **Running Example**   For the given IPM (Figure 2.1, Page 14), one MFCS with $d = 2$ exists that can activate the t-factor fault if it is covered by a truly-valid test input: `[Country:UK, Shipping:Standard]`.
>
> One minimal failure-causing truly-invalid schema exists for the configuration-independent fault in the `isInvalidCountry(country)` function: `[Country:123]`. The degree and robustness degree are equal and there is zero interaction with valid schemata, i.e. $d = 1$, $a = 1$, and $b = 0$.
>
> Two minimal failure-causing truly-invalid schemata with $d = 2$, $a = 2$, and $b = 0$ exist for the configuration-independent fault in the `isInvalid-CountryCode(country, phone)` function: `[Country:UK, Phone:+1]` and `[Country:USA, Phone:+44]`.
>
> For the configuration-dependent fault in the `isInvalidCountryCode(country, phone)` function that interactions with express shipping, two MFCSs exist: `[Country:UK, Phone:+1, Shipping:Express]` and `[Country:USA, Phone:+44, Shipping:Express]`. In this case, the degree is $d = 3$ but the robustness degree is still $a = 2$ because there is an interaction with a valid schema $b = 1$.

To conclude, information about the context is required to distinguish t-factor ordinary faults and (a, b)-factor robustness faults. When the activation sub-condition of a fault can only be satisfied by truly-valid schemata, then the fault can be abstracted as a t-factor ordinary fault. When the activation sub-condition of a fault can only be satisfied by truly-strong invalid schemata, then the fault can be abstracted as a (a, b)-factor robustness fault. Furthermore, when the schemata that satisfy the activation sub-condition are truly-minimal invalid schemata, the fault is a configuration-independent robustness fault which can be abstracted as a $(a, 0)$-factor robustness fault. When the schemata that satisfy the activation sub-condition are not truly-minimal invalid

schemata, the fault is a configuration-dependent robustness fault which can be abstracted as a (a, b)-factor robustness fault with $b > 0$. The parameters involved in the exception sub-condition determine the robustness degree a and the parameters involved in the configuration sub-condition determine the robustness interaction degree b.

The example faults and example minimal failure-causing truly-invalid schemata also highlight the convertibility between the new perspective of the refined t-factor fault model and the t-factor fault model. The refined t-factor fault model always satisfies the effective and general prevention sub-conditions because of its focus on truly-valid and truly-strong invalid schemata. In order to convert faults described by the refined t-factor model, the general prevention sub-condition must be defined. Thereby, it is also ensured that truly-valid test inputs are used to activate ordinary faults. When the general prevention sub-condition is used, it is also ensured that truly-invalid test inputs that activate robustness faults are truly-strong invalid. When the effective prevention sub-condition is used, the test inputs that activate a robustness fault are not necessarily truly-strong invalid but there is at least no invalid input masking for that fault.

Because of the effective and general prevention sub-conditions, more parameters are involved in the fault condition and higher testing strengths are required to guarantee the detection of faults. In contrast, the fault conditions of the refined t-factor fault model are smaller. Because the distinction of truly-valid and truly-strong invalid schemata is utilized, smaller testing strengths are required. But, the test adequacy criteria and test selection strategies are required to strictly distinguish truly-valid and truly-strong invalid test inputs.

Therefore, the IPM is extended in the next chapter to capture semantic information that allows a distinction between valid and strong invalid schemata. Together with the refined t-factor fault model and its additional properties, it is then used to define test adequacy criteria for CRT.

List of Symbols

a	robustness degree
b	robustness interaction degree

Chapter 6.

A Combinatorial Robustness Input Parameter Model

In the previous chapters, concepts to describe correct and robust programs from the perspective of an authoritative specification are defined. Further on, the concepts are used to discuss issues of CT in the presence of invalid input masking and a refined t-factor fault model is developed. To develop a CRT test method with test adequacy criteria and test selection strategies that target the refined t-factor fault model, the concepts are required to be defined in terms of an IPM because an authoritative specification is not available for the test selection. Therefore, we extend the structure of IPMs and provide the required definitions in this chapter.

To capture semantic information that allows a distinction between valid and invalid schemata, we distinguish two different types of constraint. The previously discussed set of exclusion-constraint C^{ex} (Definition 45, Page 32) models schemata that should be excluded from testing because they are marked as irrelevant. In addition, a separate set of error-constraints is introduced to mark schemata as invalid.

Definition 87 (Error-Constraints) *The set of error-constraints C^{err} is a set of constraints to distinguish valid and invalid schemata.*

We define the robustness input parameter model (RIPM) as an extension of the IPM that includes the set of error-constraints. Formally, a RIPM is a 4-tuple $RIPM = \langle \tilde{P}, V, C^{ex}, C^{err} \rangle$ which includes the separate set of error-constraints denoted as C^{err}.

Definition 88 (Robustness Input Parameter Model) *A RIPM is a specification-based model of the input domain. It is an extension of the IPM structure that structures the input domain via parameters, associated values, exclusion-constraints, and error-constraints.*

In contrast to the related work by COHEN et al. [Coh+94]; [Coh+97], CZERWONKA [Cze06], and YU et al. [Yu+13b] (Page 61 and following), values are not modeled via disjoint sets V^{valid} and $V^{invalid}$. Only one set of values is used and unary error-constraints describe the validity of values. Taking the two types of constraints into account, exclusion-constraints partition values into relevant and irrelevant schemata and error-constraints further partition relevant schemata into valid and invalid schemata.

Since IPMs and RIPMs can deviate from the authoritative specification, the partition is with regards to the model. Valid and invalid schemata are the counterparts of the truly-valid and truly-invalid schemata. Valid and invalid test inputs are not defined separately since they are subsumed

by the schema-based concepts. Analogously to truly-irrelevant schemata, the irrelevant schemata are not further refined.

To define valid and invalid schemata, recall the $\mathtt{SAT} : C^* \times A \rightarrow Bool$ function (See Page 32) to evaluate whether a schema $s \in A$ satisfies a set of constraints $C \in C^*$. The function evaluates to true if schema s satisfies all constrains C. Otherwise, the function evaluates to false.

Definition 89 (Valid Schema) *A relevant schema s is a valid schema if and only if it satisfies all exclusion- and error-constraints, i.e.* $\mathtt{SAT}(C^{ex} \cup C^{err}, s) = true$.

Definition 90 (Invalid Schema) *A relevant schema s is an invalid schema if and only if all exclusion-constraints C^{ex} are satisfied but at least one error-constraint $c \in C^{err}$ is not satisfied, i.e.* $\mathtt{SAT}(C^{ex}, s) = true$ *and* $\mathtt{SAT}(C^{err}, s) = false$.

To improve readability, the following predicates are introduced.

$$\mathtt{IsRelevant}(s) = \mathtt{SAT}(C^{ex}, s)$$
$$\mathtt{IsValid}(s) = \mathtt{SAT}(C^{ex} \cup C^{err}, s)$$
$$\mathtt{IsInvalid}(s) = \mathtt{SAT}(C^{ex}, s) \wedge \neg\mathtt{SAT}(C^{err}, s)$$

Running Example   The example IPM in Figure 2.1 (Page 14) can also be interpreted as a RIPM with error-constraints $c_1, ..., c_4$. To further illustrate the difference between exclusion- and error-constraints, suppose a fault in the `processDomensticStandardOrder(packages, phone)` function that is activated by valid test inputs that cover the MFCS `[Country:  UK, Shipping:  Standard]`. Since this fault is known, it should be excluded from testing while valid and invalid test inputs should be distinguished. A RIPM that considers both aspects is depicted in Figure 6.1 with error-constraints $C^{err} = \{c_1, ..., c_4\}$ and exclusion-constraint $C^{ex} = \{c_5\}$.

`[Packages:1, Country:UK, Phone:+44, Shipping:Standard]` is a valid schema because it satisfies all exclusion- and error-constraints. In contrast, schema `[Packages:123, Country:UK, Phone:+44, Shipping:Standard]` is invalid because it satisfies all exclusion-constraints but error-constraint c_1 remains unsatisfied. `[Packages:1, Country:UK, Phone:+1, Shipping:Standard]` is another invalid schema with an invalid value combination that does not satisfy c_4. Further on, `[Packages:123, Country:UK, Phone:+1, Shipping: Standard]` is an invalid schema with two unsatisfied error-constraints c_1 and c_4.

The partition of valid and invalid schemata eliminates irrelevant and invalid input masking for valid test inputs. However, invalid schemata may still be subjected to invalid input masking when more than one minimal invalid schema is covered. Therefore, the concepts of minimal and strong invalid schemata are introduced.

According to Definition 75 (Page 50), a truly-invalid schema s is minimal if and only if no proper subset schema $s' \subset s$ exists that is itself truly-invalid. Further on, a truly-invalid schema is strong truly-invalid if and only if it covers exactly one minimal truly-invalid schema (Definition 76, Page 51). With regards to error-constraints, the concepts are transferred as follows.

```
Parameters:
  p₁ : Packages:          1, 3, 123
  p₂ : Country:           UK, USA, 123
  p₃ : Phone:             +44, +1, 123
  p₄ : Shipping:          Standard, Express
Error-Constraints:
  c₁ : Packages ≠ 123
  c₂ : Country ≠ 123
  c₃ : Phone ≠ 123
  c₄ : ¬(Country = UK ∧ Phone = +1) ∧ ¬(Country = USA ∧ Phone = +44)
Exclusion-Constraints:
  c₅ : Shipping = Standard ⇒ Country ≠ USA
```

Figure 6.1.: RIPM for the Ordering Web-Service Example

Definition 91 (Strong Invalid Schema) *An invalid schema s is a strong invalid schema if and only if exactly one error-constraint remains unsatisfied, i.e. $\exists! c \in C^{err}$ such that $\mathtt{SAT}(C^{ex} \cup C^{err} \backslash \{c\}, s) = true$ and $\mathtt{SAT}(C^{ex} \cup C^{err}, s) = false$.*

To reduce the number of evaluations, let $\overline{c_i}$ denote the negation of c_i. Then, the following predicates determine whether a schema s is strong invalid for error-constraint c or in general.

$$\mathtt{IsStrongInvalid}(s, c_i) = \mathtt{SAT}(C^{ex} \cup (C^{err} \backslash \{c_i\}) \cup \{\overline{c_i}\}, s)$$
$$\mathtt{IsStrongInvalid}(s) = \exists! c_i \in C^{err}, \mathtt{IsStrongInvalid}(s, c_i)$$

A strong invalid schema is an extension of a minimal invalid schema which may cover additional valid schemata besides the one minimal invalid schema. A minimal invalid schema is always a strong invalid schema while not every strong invalid schema is a minimal invalid schema.

Definition 92 (Minimal Invalid Schema) *A strong invalid schema s is a minimal invalid schema if and only if no proper subset schema $s' \subset s$ exists that is itself an invalid schema, i.e. $\nexists s' \subset s$ such that $\mathtt{IsInvalid}(s')$.*

The following predicate determines whether a schema s is minimal invalid.

$$\mathtt{IsMinimalInvalid}(s) = \mathtt{IsStrongInvalid}(s) \wedge \forall s' \subset s, \neg \mathtt{IsInvalid}(s')$$

Running Example The invalid schema [Packages:123, Country:UK, Phone:+44, Shipping:Standard] is a strong invalid schema because it satisfies all constraints except error-constraint c_1. It covers the minimal invalid schema [Packages:123]. [Packages:1, Country:UK, Phone:+1, Shipping:Standard] is another strong invalid schema that does not satisfy c_4. It covers the minimal invalid schema [Country:UK, Phone:+1]. Further on, [Pack-

> `ages:123, Country:UK, Phone:+1, Shipping:Standard]` is an invalid schema but not a strong invalid schema because two error-constraints c_1 and c_4 remain unsatisfied. Consequently, it covers two minimal invalid schemata `[Packages:123]` and `[Country:UK, Phone:+1]`.

Error-constraints have a dual role. First, they are supposed to exclude invalid schemata from valid test inputs and from strong invalid test inputs. Second, they are supposed to specify invalid schemata that should appear in some strong invalid test inputs depending on the test adequacy criteria. If it wasn't for the second role, there would be no difference between exclusion- and error-constraints. Because of the dual role, our design of error-constraints has some implications. To discuss them, another view on error-constraints is useful that highlights the relationship between error-constraints and invalid schemata.

An error-constraint specifies a set of strong invalid schemata I_{c_i}. While each strong invalid schema $s \in I_{c_i}$ must not be covered by any valid test input, some invalid schemata must be covered by strong invalid test inputs depending on the test adequacy criteria. A single error-constraint c_i can be transformed into a set of strong invalid schemata I_{c_i} for a non-empty set of d parameters by reasoning about the specified condition.

To transform $c_i \in C^{err}$ into the set of strong invalid schemata I_{c_i}, we introduce the following symbols and functions that are based on the concept of parameter indices.

Definition 93 (Parameter Index) *A parameter index $r \in [1, n]$ is a label that references one parameter $p_r \in \tilde{P}$ of $\tilde{P} = \{p_1, ..., p_n\}$.*

Let ϕ denote a set of parameter indices and let $d = |\phi|$ denote its degree.

$$\phi = \{r_1, ..., r_d\} \text{ with each } r \in [1, n] \text{ and } d = |\phi|$$

Let the function $\texttt{Indices}(c_i)$ compute the set of parameter indices ϕ for an error-constraint $c_i \in C^{err}$ and let the function $\texttt{Degree}(c_i)$ compute the number of parameters involved in the condition of an error-constraint, i.e. $\texttt{Degree}(c_i) = d$ for $\phi = \texttt{Indices}(c_i)$.

Let the function $\texttt{Pairs}(r)$ compute all m_r parameter-value pairs for all values in V_r of parameter p_r.

$$\texttt{Pairs}(r) = \{(p_r, v_j) | v_j \in V_r\} = \{\pi_1, ..., \pi_{m_r}\}$$

Further on, let X_ϕ denote the set of all d-sized schemata with $d = |\phi|$ that exist for the parameters as indexed by ϕ. It is computed by the Cartesian product of all parameter-value pairs for all parameters indices in ϕ.

$$X_\phi = \{\{\pi_{j_1}, ..., \pi_{j_d}\} | \pi_{j_1} \in \texttt{Pairs}(r_1), ..., \pi_{j_d} \in \texttt{Pairs}(r_d)\}$$

To compute the set of all strong invalid schemata I_{c_i} for error-constraint $c_i \in C^{err}$, the set of all d-sized schemata X_ϕ for $\phi = \texttt{Indices}(c_i)$ is computed first. Then, the set of all strong invalid schemata is a filtered proper subset $I_{c_i} \subset X_{\texttt{Indices}(c_i)}$ where each schema $s \in I_{c_i}$ does not satisfy error-constraint c_i but all exclusion-constraints and all other error-constraints.

$$I_{c_i} = \{s \in X_{\texttt{Indices}(c_i)} | \texttt{IsStrongInvalid}(s, c_i)\}$$

Please note if the set of strong invalid schemata I_{c_i} was a not proper subset of the Cartesian product, i.e. $I_{c_i} = X_{\mathtt{Indices}(c_i)}$, no valid test inputs and no strong invalid test inputs with other unsatisfied error-constraints could exist.

> **Running Example** Error-constraint c_1 as listed in Figure 6.1 (Page 113) specifies a strong invalid schema for parameter indices $\mathtt{Indices}(c_1) = \phi = \{1\}$. The corresponding Cartesian product is $X_{\mathtt{Indices}(c_1)} = \{[\texttt{Packages:1}], [\texttt{Packages:3}], [\texttt{Packages:123}]\}$ and the subset of strong invalid schemata is $I_{c_1} = \{[\texttt{Packages:123}]\}$.
>
> Error-constraint c_4 specifies strong invalid schemata for parameter indices $\mathtt{Indices}(c_4) = \phi = \{2,3\}$. The corresponding Cartesian product is $X_{\mathtt{Indices}(c_4)} = \{[\texttt{Country:UK, Phone:+44}], [\texttt{Country:UK, Phone:+1}], [\texttt{Country:UK, Phone:123}], ..., [\texttt{Country:123, Phone:+1}], [\texttt{Country:123, Phone:123}]\}$ and the subset of strong invalid schemata is $I_{c_4} = \{[\texttt{Country:UK, Phone:+1}], [\texttt{Country:USA, Phone:+44}]\}$.

The strong invalid schemata I_{c_i} are typically not only strong invalid but minimal invalid schemata. But there is an exception when error-constraints are defined in a non-minimal formulation. The non-minimal formulation has no implication for the first role of error-constraints to partition valid and invalid schemata. However, it has implications on the second role to specify strong invalid schemata that should appear in some strong invalid test inputs.

> **Running Example** Consider a different example with two parameters and three values for each parameter.
>
> ```
>     P1: { 1, 2, 3 }
>     P2: { 1, 2, 3 }
> ```
> $$c_1 : \neg(\texttt{P1} = 3 \wedge \texttt{P2} = 1) \wedge \neg(\texttt{P1} = 3 \wedge \texttt{P2} = 3) \wedge \neg(\texttt{P1} = 3 \wedge \texttt{P2} = 3)$$
>
> In this case, I_{c_1} encompasses three strong invalid schemata $[\texttt{P1:3, P2:1}]$, $[\texttt{P1:3, P2:2}]$, and $[\texttt{P1:3, P2:3}]$. They are strong invalid but not minimal invalid because a sub-schema $[\texttt{P1:3}]$ exists that does not satisfy c_1.
>
> An alternative formulation $c_1' = \neg(\texttt{P1} = 3)$ is equivalent for the first role. However, for the second role only one strong invalid schema is defined by I_{c_1}' which is $[\texttt{P1:3}]$. Depending on the test adequacy criterion, the two formulations require more or less strong invalid schemata to be covered.

The introduction of error-constraints and its conditions can be justified by the need to model invalid value combinations instead of only invalid values. But the transformation to strong invalid schemata I_{c_i} requires a design decision. Therefore, we contrast the design of error-constraints with the design of exclusion-constraints. When describing irrelevant schemata, exclusion-constraints are local and independent descriptions that specify sets of irrelevant schemata. An exclusion-constraint is local and independent in the sense that other exclusion- and error-constraints do not change its semantics. Independently from any other constraint, an exclusion-

constraint always specifies the same set of schemata that are considered as irrelevant.

The transformation from error-constraints to strong invalid schemata as defined above does not have the same characteristics of locality and independence. The subset of strong invalid schemata varies depending on the descriptions of other exclusion- and error-constraints. This is due to the fact that the definition of strong invalid schemata is not based on single error-constraint but on sets of exclusion- and error-constraints. Nevertheless, it makes the modeling of invalid values and invalid value combinations unnecessary complex.

> **Running Example**   Let's consider one parameter and one invalid value first. Using the workaround, it is modeled via two sets of values.
>
> $$\texttt{Packages:} \{~\texttt{1, 3}~\}^{valid} \cup \{~\texttt{123}~\}^{invalid}$$
>
> Using our approach and the RIPM, it is modeled with one set of values and an error-constraint. In this example, both approaches specify the same minimal invalid schema.
>
> $$\texttt{Packages:} \{~\texttt{1, 3, 123}~\}$$
> $$c_1 : \texttt{Packages} \neq \texttt{123}$$
>
> Although, the invalid value cannot be derived by locally reasoning about the strong invalid schemata I_{c_1} of the single error-constraint c_1. Instead, all constraints must be considered. Suppose a second error-constraint c_2 which interferes with c_1. Then, both error-constraints do not specify any invalid schema, i.e. $I_{c_1} = \emptyset$ and $I_{c_2} = \emptyset$.
>
> $$\texttt{Packages:} \{~\texttt{1, 3, 123}~\}$$
> $$c_1 : \texttt{Packages} \neq \texttt{123}$$
> $$c_2 : \texttt{Packages} \neq \texttt{123}$$

To improve the comprehensibility and to allow local reasoning of error-constraints, we introduce another set of schemata that is specified by a single error-constraint. The set of schemata L_{c_i} denotes the schemata that are *locally-specified* by error-constraint c_i.

$$I_{c_i} \subseteq L_{c_i} \subset X_{\texttt{Indices}(c_i)}$$
$$L_{c_i} = \{s \in X_{\texttt{Indices}(c_i)} | \texttt{SAT}(\{c_i\}, s) = false\}$$

The transformation from error-constraints to locally-specified schemata has the characteristics of locality and independence. Independently from any other constraint, error-constraint c_i always specifies the same set of schemata L_{c_i}.

Although the set of schemata is defined locally and independently, not all schemata are necessarily relevant and strong invalid. On the one side, it is important to maintain local reasoning and comprehensibility of error-constraints. On the other side, it is important to ensure that only strong invalid schemata are described by error-constraints. Therefore, we argue that whenever a schema s is locally-specified by an error-constraint c_i, i.e. $s \in L_{c_i}$, but that schema is not strong invalid, i.e. $s \notin I_{c_i}$, the RIPM is inconsistent and needs to be repaired.

> **Running Example** For the first example with `Packages` and only one error-constraint c_1, both L_{c_1} and I_{c_1} are equivalent, i.e. $L_{c_1} = I_{c_1} = \{ \texttt{[Packages:123]} \}$.
>
> For the second example with two error-constraints c_1 and c_2, the locally-specified schemata are $L_{c_1} = L_{c_2} = \{ \texttt{[Packages:123]} \}$. However, the minimal invalid schemata deviate and indicate inconsistency, i.e. $I_{c_1} = I_{c_2} = \emptyset$, because the error-constraints conflict each other.

By considering both sets of schemata for an error-constraint, the comprehensibility of local reasoning is linked with the global scope of strong invalid schemata. Moreover, the combination of both sets allows detecting and repairing inconsistent RIPMs which would not be possible when considering only one set. In the following, we assume that the RIPMs are consistent and that the sets of locally-specified schemata and strong invalid schemata are equivalent for all error-constraints. A definition of inconsistent RIPMs and techniques to detect and repair inconsistent RIPMs are discussed in a later chapter (Chapter 9, Page 159 and following).

Based on the classification of valid and invalid schemata, different failure-causing schemata can be distinguished. Again, the test input-based concepts are not defined separately since they are subsumed by the schema-based concepts.

Definition 94 (Failure-causing Valid Schema) *A valid schema s is also a failure-causing valid schema if and only if each valid test input τ that covers $s \subseteq \tau$ is failure-causing, i.e. $s \subseteq \tau \Rightarrow \neg OK(\tau, P(\tau))$.*

Definition 95 (Failure-causing Invalid Schema) *An invalid schema s is a failure-causing invalid schema if and only if each strong invalid test input τ that covers $s \subseteq \tau$ is failure-causing, i.e. $s \subseteq \tau \Rightarrow \neg OK(\tau, P(\tau))$.*

Definition 96 (Minimal Failure-causing Valid Schema) *A failure-causing valid schema s is a minimal failure-causing valid schema if and only if no proper subset schema $s' \subset s$ exists that is itself a failure-causing schema.*

Definition 97 (Minimal Failure-causing Invalid Schema) *A failure-causing invalid schema s is a minimal failure-causing invalid schema if and only if no proper subset schema $s' \subset s$ exists that is itself a failure-causing schema.*

Again, the property of minimality is related to the concept of causing failures. It should not be confused with the property of minimality that is related to invalid schemata. A minimal failure-causing invalid schema is an invalid schema but not necessarily a strong or minimal invalid schema. A minimal failure-causing invalid schema that is also a minimal invalid schema activates a configuration-independent robustness fault while a minimal failure-causing invalid schema that is not a minimal invalid schema activates a configuration-dependent robustness fault. That distinction is examined more closely in Section 5.4 (Page 105 and following).

To conclude this chapter, the structure of IPMs is extended to the structure of RIPMs with error-constraints. The error-constraints allow distinguishing valid and invalid schemata as well as failure-causing valid and invalid schemata. In the following, test adequacy criteria and test selection strategies are developed that are based on these newly introduced concepts to target the refined t-factor fault model.

List of Symbols

$RIPM = \langle \tilde{P}, V, C^{ex}, C^{err} \rangle$	input parameter model
C^{err}	set of error-constraints
$r \in [1, n]$	parameter index
$\phi = \{r_1, ..., r_d\}$	set of $d = \lvert \phi \rvert$ parameter indices
$\texttt{Indices}(c_i)$	function to compute the set of parameter indices ϕ for parameters involved in the condition of error-constraint c_i
$\texttt{Degree}(c_i)$	function to compute the number of parameters involved in the condition of error-constraint
$\texttt{Pairs}(r)$	function to compute all parameter-value pairs of parameter with index r
X_ϕ	set of all d-sized schemata of parameters indexed by ϕ
$X_{\texttt{Indices}(c_i)}$	set of all schemata defined by error-constraint c_i
I_{c_i}	set of all strong invalid schemata specified by error-constraint c_i
L_{c_i}	set of invalid schemata locally specified by error-constraint c_i
$\texttt{IsRelevant}(s)$	predicate to check if schema s is relevant
$\texttt{IsValid}(s)$	predicate to check if schema s is valid
$\texttt{IsInvalid}(s)$	predicate to check if schema s is invalid
$\texttt{IsStrongInvalid}(s, c_i)$	predicate to check if schema s is strong invalid for error-constraint c_i
$\texttt{IsStrongInvalid}(s)$	predicate to check if schema s is strong invalid
$\texttt{IsMinimalInvalid}(s)$	predicate to check if schema s is minimal invalid

Chapter 7.

Combinatorial Robustness Test Adequacy Criteria

Contents

Given the RIPM structure as defined in the previous chapter, test adequacy criteria can now incorporate the partitioning of error-constraints and distinct test adequacy criteria can now be defined for testing correctness and robustness.

To discuss new test adequacy criteria, we first introduce existing test adequacy criteria and use a formalism. First, the introduction of existing test adequacy criteria allows to illustrate relationships to the new test adequacy criteria and to differentiate them from the new test adequacy criteria. Second, the formalism enables a precise description of test adequacy criteria and allows to show why test adequacy criteria for strong invalid test inputs cannot be defined analogously.

7.1. Existing t-wise Combinatorial Test Adequacy Criteria

In CT, a test adequacy criterion is defined in relation to a IPM with the parameters, values, and constraints the tester decides to be important [GOA05]. The fault detection effectiveness (FDE) depends on the completeness and correctness of chosen parameters and values as well as their partitions based on exclusion-constraints and – in the case of CRT – error-constraints. The test adequacy criteria cannot influence the completeness and correctness of IPMs and RIPMs. The exhaustive test suite $E = V_1 \times ... \times V_n$ defines the domain of which test inputs are selected and the selection is guided by test adequacy criteria. Although, what is not described in IPMs and RIPMs, is not part of the exhaustive test suite and cannot be taken into account. Instead, test adequacy criteria can influence which and how many test inputs of the exhaustive test suite must be selected into the test suite $T \subseteq E$.

In the case of t-wise coverage, the predicate that test suites must satisfy is solely defined by combinatorics of parameters and values without considering semantic information [GOA05]. As we have argued beforehand, it is important to avoid irrelevant schemata and to separate valid from invalid schemata because covering an irrelevant schema makes an entire test input irrelevant and covering an invalid schema makes an entire test input invalid.

To incorporate the refined t-factor fault model for robustness faults, we define combinatorial robustness test adequacy criteria that consider semantic information. Therefore, we first introduce a definition of t-wise coverage that is based on a formalism by WILLIAMS & PROBERT [WP01]. Afterwards, we extend the definition to relevant t-wise coverage and introduce combinatorial robustness test adequacy criteria in relation to RIPMs that further refine the selection of relevant test inputs into separate test suites for valid and strong invalid test inputs.

t-wise Coverage   The t-wise coverage test adequacy criterion requires a test suite T to cover all schemata with degree $d = t$ at least once [GOA05]. To formally define t-wise coverage, we rely on the following symbols and functions.

Let ϕ again denote a set of $d = |\phi|$ parameter indices.

$$\phi = \{r_1, ..., r_d\} \text{ with each } r \in [1, n] \text{ and } d = |\phi|$$

Let ϕ_n denote a particular set of n parameter indices that contains all parameter indices of P.

$$\phi_n = \{r_1 = 1, ..., r_n = n\}$$

Let Φ_d denote a set of parameter index sets with all parameter index sets containing d indices.

$$\Phi_d = \{\phi_1, \phi_2, ...\} \text{ with } |\phi_1|, |\phi_2|, ... = d$$

Let the function $\texttt{Subsets}(\phi, d)$ compute a set Φ_d that contains all possible d-sized parameter index subsets of a given set ϕ.

$$\texttt{Subsets}(\phi, d) = \Phi_d = \begin{cases} \{\emptyset\} & \text{if } d \leq 0 \\ \{\phi' | \phi' \subseteq \phi \wedge |\phi'| = |\phi|\} & \text{if } d < |\phi| \\ \{\phi\} & \text{if } d \geq |\phi| \end{cases}$$

Let $\texttt{AllSubsets}$ denote a function that computes a set Φ_d that contains all generally possible d-sized sets of parameter indices. Therefore, the $\texttt{Subsets}$ function is applied to ϕ_n.

$$\texttt{AllSubsets}(d) = \texttt{Subsets}(\phi_n, d)$$

Let X_ϕ again denote a set of all schemata with degree $d = |\phi|$ that exist for the parameter indices of ϕ.

$$X_\phi = \{\{\pi_{j_1}, ..., \pi_{j_d}\} | \pi_{j_1} \in \texttt{Pairs}(r_1), ..., \pi_{j_d} \in \texttt{Pairs}(r_d)\}$$

Finally, let X_d denote the set of all schemata with degree d. It can be built as the union of all X_ϕ schema sets for all $\phi \in \texttt{AllSubsets}(d)$.

$$X_d = \bigcup_{\phi \in \texttt{AllSubsets}(d)} X_\phi$$

Running Example   For the IPM of Figure 6.1 (Page 113), six subsets of parameter indices exist for $\phi_n = \{1, 2, 3, 4\}$, i.e. $\texttt{AllSubsets}(2) = \{\phi_1 = \{1, 2\}, \phi_2 =$

$\{1,3\}, \phi_3 = \{1,4\}, \phi_4 = \{2,3\}, \phi_5 = \{2,4\}, \phi_6 = \{3,4\}\}$.

For the set of parameter indices $\phi_5 = \{2,4\}$, the set of all schemata X_{ϕ_5} is as follows.

$$X_{\phi_5} = \{\texttt{[Shipping:Standard, Country:UK],}$$
$$\texttt{[Shipping:Standard, Country:USA],}$$
$$\texttt{[Shipping:Standard, Country:123],}$$
$$\texttt{[Shipping:Express, Country:UK],}$$
$$\texttt{[Shipping:Express, Country:USA],}$$
$$\texttt{[Shipping:Express, Country:123],}$$
$$\texttt{[Shipping:123, Country:UK],}$$
$$\texttt{[Shipping:123, Country:USA],}$$
$$\texttt{[Shipping:123, Country:123]}\}$$

The set of all schemata X_2 encompasses all sets of X_ϕ for all $\phi \in \texttt{AllSubsets}(2)$.

A test suite $T \subseteq E$ satisfies the t-wise coverage test adequacy criterion if and only if each schema in X_t with $d = t$ is covered by at least one test input in T.

$$\forall s \in X_t, \exists \tau \in T \text{ such that } s \subseteq \tau$$

N-wise Coverage   For the sake of completeness, N-wise coverage denotes a special case of t-wise coverage with $t = n$ which can only be satisfied by the exhaustive test suite, i.e. $T = E$ [GOA05]. It can be alternatively defined such that a test suite T satisfies N-wise coverage if and only if each test input of the exhaustive test suite $\tau_a \in E$ is covered by at least one test input of the test suite $\tau_b \in T$.

$$\forall \tau_a \in E, \exists \tau_b \in T \text{ such that } \tau_a \subseteq \tau_b$$

In the following, the above presented formalism is extended to describe combinatorial test adequacy criteria that consider constraints.

N-wise Relevant Coverage   A subset of N-wise coverage is N-wise relevant coverage which excludes irrelevant test inputs and limits testing to relevant test inputs [GOA05].

A test suite T satisfies N-wise relevant coverage if and only if each relevant test input of the exhaustive test suite E is covered by at least one test input of the test suite.

$$T \supseteq \{\tau \in E | \texttt{IsRelevant}(\tau)\}$$

t-wise Relevant Coverage   The t-wise relevant coverage test adequacy criterion requires a test suite $T \subseteq E$ to cover each relevant schema with degree $d = t$ by at least one relevant test input [GOA05]. To describe t-wise relevant coverage more precisely, X_ϕ and X_d are refined to $X_\phi^{relevant}$ and $X_d^{relevant}$ to only cover relevant schemata and to exclude irrelevant schemata.

$$X_{\phi}^{relevant} = \{s \in X_{\phi} | \texttt{IsRelevant}(s)\} \quad \text{and} \quad X_{d}^{relevant} = \bigcup_{\phi \in \texttt{AllSubsets}(d)} X_{\phi}^{relevant}$$

Running Example   When incorporating the exclusion-constraint c_6 of Figure 6.1 (Page 113), the schema [Shipping:Standard, Country:USA] is excluded. Consequently, $X_{\phi}^{relevant}$ for $\phi_5 = \{2, 4\}$ is a subset of X_{ϕ_5} that consists of the eight remaining schemata.

$$
\begin{aligned}
X_{\phi_5} = \{ &\texttt{[Shipping:Standard, Country:UK]},\\
&\texttt{[Shipping:Standard, Country:123]},\\
&\texttt{[Shipping:Express, Country:UK]},\\
&\texttt{[Shipping:Express, Country:USA]},\\
&\texttt{[Shipping:Express, Country:123]},\\
&\texttt{[Shipping:123, Country:UK]},\\
&\texttt{[Shipping:123, Country:USA]},\\
&\texttt{[Shipping:123, Country:123]}\}
\end{aligned}
$$

For the other sets of parameter indices ϕ, the sets of all relevant schemata $X_{\phi}^{relevant}$ are equal to the sets of all schemata X_{ϕ}. Therefore, $X_{2}^{relevant}$ is a subset of X_2 without [Shipping:Standard, Country:USA].

A test suite $T \subseteq E$ satisfies the t-wise relevant coverage test adequacy criterion if and only if each schema in $X_t^{relevant}$ with $d = t$ is covered by at least one relevant test input in T.

$$\forall s \in X_t^{relevant}, \exists \tau \in T \text{ such that } s \subseteq \tau \text{ and } \texttt{IsRelevant}(\tau)$$

N-wise Valid Coverage   To detect t-factor ordinary faults, not only irrelevant schemata but also invalid schemata must be excluded to avoid irrelevant and invalid input masking. Therefore, N-wise valid coverage is a subset of N-wise relevant coverage which further excludes invalid test inputs and limits testing to valid test inputs.

A test suite T satisfies N-wise valid coverage if and only if each valid test input of the exhaustive test suite E is covered by at least one valid test input of the test suite.

$$T \supseteq \{\tau \in E | \texttt{IsValid}(\tau)\}$$

t-wise Valid Coverage   The t-wise valid coverage is another refinement of t-wise relevant coverage that targets t-factor ordinary faults. It covers only valid schemata and excludes all schemata that are irrelevant or invalid [GOA05]. Therefore, $X_{\phi}^{relevant}$ and $X_{d}^{relevant}$ are further refined to X_{ϕ}^{valid} and X_{d}^{valid} to only cover valid schemata and to exclude irrelevant and invalid schemata.

$$X_{\phi}^{valid} = \{s \in X_{\phi}^{relevant} | \texttt{IsValid}(s)\} \quad \text{and} \quad X_{d}^{valid} = \bigcup_{\phi \in \texttt{AllSubsets}(d)} X_{\phi}^{valid}$$

Running Example To define the sets of all valid schemata, the exclusion-and-error-constraints must be taken into account. For $\phi_5 = \{2, 4\}$, $X_{\phi_5}^{valid}$ is a subset of $X_{\phi_5}^{relevant}$ with each schema satisfying not only all exclusion- but also all error-constraints.

$$X_{\phi_5}^{valid} = \{ \texttt{[Shipping:Standard, Country:UK]},$$
$$\texttt{[Shipping:Express, Country:UK]},$$
$$\texttt{[Shipping:Express, Country:USA]} \}$$

The set of all valid schemata X_2^{valid} for all sets of parameter indices ϕ is consequently a subset of $X_2^{relevant}$ with only valid schemata.

A test suite $T \subseteq E$ satisfies the t-wise valid coverage if and only if each schema in X_d^{valid} with $d = t$ is covered by at least one valid test input in T.

$$\forall s \in X_t^{valid}, \exists \tau \in T \text{ such that } s \subseteq \tau \text{ and } \texttt{IsValid}(\tau)$$

In related work such as the survey by GRINDAL et al. [GOA05], relevance and validness are often not clearly distinguished. It depends on the context whether the N-wise and t-wise valid coverage as defined by related work equates to our relevant or valid test adequacy criteria. When the tester uses constraints to avoid all schemata that are not of any interest to the test, it equates to our relevant test adequacy criteria. It equates to our valid test adequacy criteria when certain schemata are not excluded but marked as invalid to test them separately.

Hierarchy of Test Adequacy Criteria Before we discuss and introduce test adequacy criteria for strong invalid test inputs, we illustrate the hierarchy of and relationships between the existing test adequacy criteria. Figure 7.1 (Page 124) depicts the relationships between the test adequacy criteria. The figure is adapted from GRINDAL et al. [GOA05] and the relationships are based on the *subsumption relation*.

The subsumption relation is defined as follows: A test adequacy criterion TAC_1 subsumes a test adequacy criterion TAC_2 if and only if each test suite that satisfies TAC_1 always satisfies TAC_2 as well [FW88]; [Wei90]; [GOA05]; [AO16]. It is widely used to analytically compare test adequacy criteria [Wei89]; [AO16]. It can be seen as an approximation with regards to the FDE of related test adequacy criteria [Wei89]. For accurate comparisons, subsumption and similar relations are viewed critically (cf. [Wei89]; [WWH91]; [Wey02]). Instead of formal analyses, empirical approaches are commonly used to compare test adequacy criteria, test selection strategies, and test methods in general. Therefore, we use the subsumption relation to provide an overview and to support the comprehension. Later on, empirical approaches are used in our evaluation (Part IV, Page 211 and following).

In the context of combinatorial test adequacy criteria, the subsumption relation can be discussed in terms of schemata that a test suite must cover.

N-wise coverage can only be satisfied by the exhaustive test suite $T = E$. Therefore, it subsumes all other RIPM-related test adequacy criteria by definition. The N-wise coverage is equivalent to t-wise coverage with $t = n$ and t-wise coverage with $t \leq n$ is subsumed by N-

Figure 7.1.: Subsumption Hierarchy of Existing Combinatorial Test Adequacy Criteria

wise coverage. Each-choice coverage ($t = 1$) and pair-wise coverage (2-wise) are special cases of t-wise coverage which are subsumed by t-wise coverage with $t = n$ and $t \geq 2$.

Further on, N-wise relevant coverage is also subsumed by N-wise coverage because test suites that satisfy N-wise relevant coverage must cover all relevant test inputs which is a subset of the exhaustive test suite. Analogously, t-wise relevant coverage is subsumed by N-wise relevant coverage. Pair-wise relevant coverage and each-choice relevant coverage are subsumed by t-wise relevant coverage.

Even though N-wise coverage subsumes N-wise relevant coverage, t-wise coverage does not subsume t-wise relevant coverage. The set of all t-sized schemata X_t of t-wise coverage is a superset of the set of all t-sized relevant schemata $X_t^{relevant}$. Therefore, a test suite that satisfies t-wise coverage covers all schemata that a test suite must cover to satisfy t-wise relevant coverage. However, t-wise relevant coverage has an additional requirement that restricts the way how schemata are combined. To satisfy t-wise relevant coverage, all schemata must be covered by relevant test inputs.

When all schemata are relevant and no irrelevant schemata exist, the set of test suites that satisfy N-wise relevant coverage and t-wise relevant coverage are equivalent to N-wise coverage and t-wise coverage.

The N-wise relevant coverage subsumes N-wise valid coverage because all valid test inputs are a subset of all relevant test inputs. Again, t-wise valid coverage is subsumed by N-wise valid coverage and is equivalent for $t = n$. In addition, each-choice valid coverage and pair-wise valid coverage are subsumed by t-wise valid coverage with $t \geq 2$.

The t-wise relevant coverage does not subsume t-wise valid coverage. The set of all valid t-sized schemata X_t^{valid} of t-wise valid coverage is a subset of the set of all t-sized relevant schemata $X_t^{relevant}$. But, t-wise valid coverage requires all schemata to be covered by valid test inputs.

When all relevant schemata are valid and no invalid schemata exist, the set of test suites that satisfy N-wise relevant coverage and t-wise relevant coverage are equivalent to N-wise valid coverage and t-wise relevant coverage.

Furthermore, the subsumption hierarchy depicts base-choice coverage. Base-choice coverage is a test adequacy criterion defined by AMMANN & OFFUTT [AO94] as an alternative to each-choice. It is considered as related work (Chapter 4, Page 58) and has limited support for testing with valid and strong invalid test inputs. Therefore, it is included in this subsumption hierarchy.

Base-choice coverage uses a base test input with default values and test inputs are derived from the base test input by replacing one parameter value at a time. Therefore, base-choice coverage subsumes each-choice coverage because a test suite that satisfies base-choice coverage covers all values. Although, a test suite that satisfies base-choice coverage requires more test inputs than a test suite that only satisfies each-choice coverage since a new test input is derived from the base test input for each value. Base-choice coverage can also subsume each-choice relevant coverage and each-choice valid coverage when all test inputs that are derived for relevant or valid values are itself relevant or valid, respectively.

List of Symbols

$\phi = \{r_1, ..., r_d\}$	set of $d = \|\phi\|$ parameter indices
$\phi_n = \{r_1 = 1, ..., r_n = n\}$	set of all n parameter indices
Φ_d	set of d-sized parameter index sets
$\texttt{Subsets}(\phi, d)$	function to compute all d-sized subsets of parameter indices for ϕ
$\texttt{AllSubsets}(d)$	function to compute all d-sized subsets of parameter indices for ϕ_n
X_ϕ	set of all d-sized schemata of parameters indexed by ϕ
X_d	set of sets of all schemata X_ϕ for all $\phi \in \texttt{AllSubsets}(d)$
$X_\phi^{relevant}$	set of all relevant d-sized schemata of parameters indexed by ϕ
$X_d^{relevant}$	set of sets of all relevant schemata $X_\phi^{relevant}$ for all $\phi \in \texttt{AllSubsets}(d)$
X_ϕ^{valid}	set of all valid d-sized schemata of parameters indexed by ϕ

| X_d^{valid} | set of sets of all valid schemata X_ϕ^{valid} for all $\phi \in$ `AllSubsets`(d) |

7.2. Combinatorial Test Adequacy Criteria for Strong Invalid Test Inputs

Based on the formalism introduced in the previous section, we first discuss why an analogously defined test adequacy criterion for strong invalid test inputs is insufficient. Afterwards, we introduce a family of test adequacy criteria to detect (a, b)-factor robustness faults as defined by the refined t-factor fault model (Section 5.4, Page 105 and following).

To detect (a, b)-factor robustness faults, it is important (1) to avoid irrelevant input masking and invalid input masking that can be caused by EHs with earlier locations in the control-flow, (2) to cover the failure-causing invalid schema of degree a, and (3) to cover the robustness interactions of degree b. Given our fault model, this work is limited to robustness faults that are detected by strong invalid schemata.

N-wise Strong Invalid Coverage N-wise strong invalid coverage defines the upper boundary of all strong invalid test inputs of which other test adequacy criteria can define a meaningful subset. A test suite $T \subseteq E$ satisfies N-wise strong invalid coverage if and only if each strong invalid test input of the exhaustive test suite E is covered by at least one strong invalid test input of the test suite.

$$T \supseteq \{\tau \in E | \texttt{IsStrongInvalid}(\tau)$$

t-wise Strong Invalid Coverage   Analogously to t-wise valid coverage, one could define t-wise strong invalid coverage utilizing the ideas of t-wise valid coverage but selecting strong invalid schemata instead. The test adequacy criterion would require to cover each schema of degree $d = t$ that can appear in a strong invalid test input:

$$X'_\phi = \{s \in X_\phi^{relevant} | \texttt{IsStrongInvalid}(s)\} \quad \text{and} \quad X'_d = \bigcup_{\phi \in \texttt{AllSubsets}(d)} X'_\phi$$

However, this test adequacy criterion would not be useful for the following reasons. First, strong invalid test inputs require exactly one error-constraint to remain unsatisfied. Therefore, the schemata cannot be arbitrarily combined to minimize the overall number of test inputs. This is contrary to the combinatorics of t-wise valid coverage and would result in larger test suites. Second, the criterion would require to cover valid schemata of degree $d = t$ independently of any strong invalid schema simply because they can appear in a strong invalid test input. Third, the degree d of minimal failure-causing invalid schemata is divided into the robustness degree a and the robustness interaction degree b with $d = a + b$. With a fixed testing strength t, the robustness interaction degree would vary depending on the degree of strong invalid schemata that are defined by an error-constraints. Fourth, the test adequacy criterion would require to cover all schemata of size $d = t$ but strong invalid schemata with a higher degree $d > t$ are not guaranteed to be covered. This conversely means that the testing strength is determined by the largest strong invalid schemata as modeled by the RIPM.

Running Example   Suppose a testing strength of $t = 2$ for the example depicted in Figure 6.1 (Page 113). Invalid schemata of error-constraint c_1 have a degree of $d = 1$ and cover a value for the parameter `Indices`$(c_1) = \phi = \{1\}$. Since 2-wise strong invalid coverage would require to cover all schemata of all 2-wise subsets of parameter indices X'_2, the parameter index 1 would be combined with all other parameter indices, e.g. $\Phi_2 = \{\phi_1 = \{1, 2\}, \phi_2 = \{1, 3\}, \phi_3 = \{1, 4\}\}$. In contrast, invalid schemata of error-constraint c_4 have a degree of $d = 2$ and encompass the parameters `Indices`$(c_4) = \phi = \{2, 3\}$. Therefore, the invalid schemata would not be combined with any valid schema of other parameters. For strong invalid schemata of error-constraints c_1, the robustness interaction degree would be $b = 1$ and for strong invalid schemata of error-constraints c_4, the robustness interaction degree would be $b = 0$.

Further on, suppose a testing strength of $t = 1$ for t-wise strong invalid coverage. All invalid schemata of error-constraint c_1 would be considered because Φ_1 contains `Indices`$(c_1) = \phi = \{1\}$. In contrast, the invalid schemata of error-constraints that encompass more parameters like c_4 would only be considered partially. Not `Indices`$(c_4) = \phi = \{2, 3\}$ but $\phi = \{2\}$ and $\phi = \{3\}$ are considered by X'_1.

In the running example, both invalid schemata `[Country:UK, Phone:+1]` and `[Country:USA, Phone:+44]` could appear in a test suite that satisfies 1-wise strong invalid coverage. However, it is not guaranteed because the four values that they cover could have been covered by other strong invalid test inputs.

To summarize the problems of t-wise strong invalid coverage, the robustness interaction degree would be distributed unevenly among specified strong invalid schemata and the coverage of specified strong invalid schemata would be uneven as well. For small strong invalid schemata for sets of parameter indices ϕ with $|\phi| \leq t$, all strong invalid schemata would be covered by a test suite that satisfies t-wise strong invalid coverage. Small strong invalid schemata for sets of parameter indices ϕ with $|\phi| < t$ would appear more often because of the robustness interaction degree. For larger strong invalid schemata for sets of parameter indices ϕ with $|\phi| > t$, the appearance would not be guaranteed. It is even possible to have error-constraints c_i for which none of the specified strong invalid schemata I_{c_i} are covered when their parameter index sets encompass too many parameters. i.e. ϕ with $|\phi| > t$.

For those reasons, t-wise strong invalid coverage is rejected and a different approach to strong invalid coverage is chosen. Since exactly one error-constraint remains unsatisfied for every strong invalid test input and each error-constraint specifies a set of strong invalid schemata of which *some* should appear in the test suite, we define the test adequacy criteria relative to one error-constraint. These test adequacy criteria can be defined once for all error-constraints of a RIPM or individually for each error-constraint of a RIPM.

To circumvent the uneven distribution of robustness interaction degrees and the uneven coverage of strong invalid schemata, the test adequacy criteria can be adjusted in two dimensions that reflect the robustness degree a and the robustness interaction degree b.

In the following, the first dimension regarding the robustness degree a is discussed. It defines how many different invalid schemata must appear in some strong invalid test inputs. Therefore, the perspective is on one particular error-constraint $c_i \in C^{err}$ which specifies a set of strong

invalid schemata I_{c_i} for which *some* strong invalid test inputs should be generated such that all strong invalid test inputs do not satisfy c_i but all other exclusion- and error-constraints.

All Strong Invalid Schemata The most demanding requirement is to expect all strong invalid schemata $s \in I_{c_i}$ as specified by error-constraint $c_i \in C^{err}$ to be covered by some strong invalid test input $s \subseteq \tau$. When the parameters of the error-constraint match the parameters of an exception-condition, the exception-condition is falsified by each specified strong invalid schemata.

Some Strong Invalid Schemata The least demanding requirement is to expect at least one strong invalid schemata $s \in I_{c_i}$ as specified by error-constraint $c_i \in C^{err}$ to be covered by some strong invalid test input $s \subseteq \tau$. When the parameters of the error-constraint match the parameters of an exception-condition, the exception-condition is falsified by at least one strong invalid test input.

a-wise Strong Invalid Schemata In between the least and most demanding requirement, a-wise allows to find a compromise that utilizes the idea of combinatorics similar to the idea of t-wise combinatorics in CT. All a-sized sub-schemata of I_{c_i} are expected to be covered by some strong invalid test inputs. When the parameters of the error-constraint match the parameters of an exception-condition, the exception-condition is falsified by strong invalid schemata that cover each a-sized sub-schema of I_{c_i}.

The first option "All Strong Invalid Schemata" can be considered as a special case of "a-wise Strong Invalid Schemata" with a being equal to the number of parameters, i.e. $a = \texttt{Degree}(c_i)$.

In the following, the second dimension regarding the robustness interaction degree b is discussed. It defines how the different strong invalid schemata as selected by an option of the first dimension are combined with valid schemata of the remaining parameters.

No Robustness Interactions Each selected strong invalid schemata is expected to appear in at least one test input. The other schemata of the same test input are of no significance as long as the test input is strong invalid.

All Robustness Interactions Each selected strong invalid schemata is expected to be combined with all valid schemata of the remaining parameters.

b-wise Robustness Interactions Each selected strong invalid schemata is expected to be combined with all b-wise valid schemata of the remaining parameters.

The first option "No Robustness Interactions" and the second option "All Robustness Interactions" can be considered as special cases of "b-wise Robustness Interactions" with $b = 0$ indicating no robustness interactions and b being equal to the number of remaining parameters indicates all robustness interactions, i.e. $b = n - \texttt{Degree}(c_i)$. Since our focus is on robustness faults, we believe that "All Robustness Interactions" is not a significant option that should be emphasized with a distinct name. Therefore, we only distinguish no robustness interaction ($b = 0$) and b-wise robustness interactions with $b > 0$.

	No Robustness Interactions	b-wise Robustness Interactions
All Strong Invalid Schemata	$(\forall, 0)$-wise Strong Invalid Coverage	$(\forall, b)$-wise Strong Invalid Coverage
Some Strong Invalid Schemata	$(\exists, 0)$-wise Strong Invalid Coverage	$(\exists, b)$-wise Strong Invalid Coverage
a-wise Strong Invalid Schemata	$(a, 0)$-wise Strong Invalid Coverage	(a, b)-wise Strong Invalid Coverage

Table 7.2.: Strong Invalid Coverage Test Adequacy Criteria

Combining the two dimensions results in six possible test adequacy criteria of which we consider five to be meaningful. The combinations are depicted in Table 7.2 (Page 130) and discussed below.

All Strong Invalid Schemata and No Robustness Interactions   The $(\forall, 0)$-wise strong invalid coverage test adequacy criterion focuses on configuration-independent robustness faults and all specified strong invalid schemata must appear in strong invalid test inputs. Thereby, the exception-condition that is described by an error-constraint is falsified several times and robustness faults can be detected if at least one strong invalid schema activates them.

A test suite $T \subseteq E$ satisfies $(\forall, 0)$-wise strong invalid coverage for error-constraint $c_i \in C^{err}$ if and only if for each strong invalid schema $s \in I_{c_i}$ as specified by c_i there exists at least one strong invalid test input $\tau \in T$ that covers $s \subseteq \tau$.

$$\forall s \in I_{c_i}, \exists \tau \in T \text{ such that } s \subseteq \tau \text{ and } \texttt{IsStrongInvalid}(\tau, c_i)$$

> **Running Example**   Since error-constraint c_1 of the example depicted in Figure 6.1 (Page 113) specifies exactly one strong invalid schema [Packages:123], only one test input is required to satisfy $(\forall, 0)$-wise strong invalid coverage. In this case, there is no difference to $(\exists, 0)$-wise strong invalid coverage.
>
> Error-constraint c_4 specifies two strong invalid schemata [Country:UK, Phone:+1] and [Country:USA, Phone:+44]. While only one strong invalid test input is required to satisfy $(\exists, 0)$-wise strong invalid coverage, at least two test inputs are required that cover both strong invalid schemata to satisfy $(\forall, 0)$-wise strong invalid coverage, e.g. [Packages:1, Country:UK, Phone:+1, Shipping: Standard] and [Packages:1, Country:USA, Phone:+44, Shipping: Standard]. Although, the same valid schemata can be used for the remaining parameters.

All Strong Invalid Schemata and b-wise Robustness Interactions   To test interactions with valid schemata of the remaining parameters, the "b-wise robustness interactions" option introduces an additional requirement to the $(\forall, b)$-wise strong invalid coverage. In contrast

to $(\forall, 0)$-wise strong invalid coverage, each strong invalid schema must not only be covered by some strong invalid test inputs but it must be combined all valid schemata of degree $d = b$.

This test adequacy criteria focuses on configuration-dependent robustness faults where a strong invalid schema $s \in I_{c_i}$ only becomes failure-causing in combination with a certain valid schema of degree $d = b$. Since the exact combination of strong invalid schema and valid schema is unknown before detection, each strong invalid schema is combined with all valid schemata.

To formally define $(\forall, b)$-wise strong invalid coverage, we first describe how to compute the sets of parameter indices that reflect the interaction between strong invalid schemata and valid schemata. For error-constraint c_i, the set of parameter indices is computed by $\texttt{Indices}(c_i)$.

Further, we introduce the function $\texttt{Complement}(\phi)$ to compute the complement of ϕ relative to all parameter indices ϕ_n.

$$\texttt{Complement}(\phi) = \{l \in \phi_n | l \notin \phi\}$$

The set of remaining parameter indices is computed by $\texttt{Complement}(\texttt{Indices}(c_i))$ and its degree is $n - \texttt{Degree}(c_i)$. To compute all b-sized subsets of remaining parameter indices with $b \in [0, n - \texttt{Degree}(c_i)]$, the $\texttt{Subsets}(\phi, d)$ function can be applied to the remaining parameter indices. We use the function $\texttt{RemainingSubsets}(c_i, b)$ as an abbreviation for all b-sized subsets of remaining parameter indices.

$$\texttt{RemainingSubsets}(c_i, b) = \texttt{Subsets}(\texttt{Complement}(\texttt{Indices}(c_i)), b)$$

To incorporate the interaction between the parameter indices of the error-constraint and the remaining parameters, we introduce the function $\texttt{AllInteractions}(c_i, b)$ that computes all combinations of parameter indices of $\texttt{Indices}(c_i)$ and the b-sized index sets of the remaining parameters. The computation is based on an element-wise union operation of $\texttt{Indices}(c_i)$ and each $\phi \in \texttt{RemainingSubsets}(c_i, b)$.

$$\texttt{AllInteractions}(c_i, b) = \{\phi \cup \texttt{Indices}(c_i) | \phi \in \texttt{RemainingSubsets}(c_i, b)\}$$

The degree of the resulting parameter index sets is $\texttt{Degree}(c_i) + b$. With $b = 0$, the set of interactions that must be covered is equivalent to the "No Robustness Interactions" option.

Running Example Error-constraint c_1 of the example depicted in Figure 6.1 (Page 113) specifies one strong invalid schema $\texttt{[Packages:123]}$. The parameter indices of error-constraint c_1 are $\texttt{Indices}(c_1) = \phi = \{1\}$ and the remaining parameter indices are $\texttt{Complement}(\texttt{Indices}(c_1)) = \{2, 3, 4\}$.

The b-sized sets of remaining parameter indices with $b = 1$ are $\texttt{RemainingSubsets}(c_1, 1) = \{\phi_1 = \{2\}, \phi_2 = \{3\}, \phi_3 = \{4\}\}$. With $b = 2$, the sets are $\texttt{RemainingSubsets}(c_1, 2) = \{\phi_1 = \{2, 3\}, \phi_2 = \{2, 4\}, \phi_3 = \{3, 4\}\}$.

The parameter index sets for 1-wise robustness interaction are $\texttt{AllInteractions}(c_1, 1) = \{\phi_1 = \{1, 2\}, \phi_2 = \{1, 3\}, \phi_3 = \{1, 4\}\}$. For 2-wise robustness interaction, they are $\texttt{AllInteractions}(c_1, 2) = \{\phi_1 = \{1, 2, 3\}, \phi_2 = \{1, 2, 4\}, \phi_3 = \{1, 3, 4\}\}$.

Please note, the interaction between strong invalid schemata and b-sized valid schemata is an important aspect and the computation of $\texttt{AllInteractions}(c_i, b)$ significantly influences the characteristics of the test adequacy criterion. When choosing $(\forall, b)$-wise strong invalid coverage, one assumes that there might be a configuration-dependent robustness fault where one of the specified strong invalid schemata $\exists s \in I_{c_i}$ of c_i might be failure-causing but only in combination with a b-sized valid schema of the remaining parameters. To ensure that such a robustness fault can be detected reliably, each strong invalid schema is combined with all b-sized valid schemata.

The computation of parameter index sets is based on an element-wise union operation, i.e. $\{\phi \cup \texttt{Indices}(c_i) | \phi_d \in \texttt{RemainingSubsets}(c_i, b)\}$. Other computations like union of sets, i.e. $\texttt{RemainingSubsets}(c_i, b) \cup \texttt{Indices}(c_i)$, are not sufficient. We will show by example that set union covers all strong invalid schemata $s \in I_{c_i}$ and all b-sized valid schemata but no interaction between them.

> **Running Example** Again, the parameter index set for error-constraint c_1 (Figure 6.1, Page 113) is $\texttt{Indices}(c_1) = \{1\}$, the $b = 2$-sized sets of remaining parameter indices are $\texttt{RemainingSubsets}(c_1, 2) = \{\phi_1 = \{2, 3\}, \phi_2 = \{2, 4\}, \phi_3 = \{3, 4\}\}$. And the parameter index sets for $b = 2$ are $\texttt{AllInteractions}(c_1, 2) = \{\phi_1 = \{1, 2, 3\}, \phi_2 = \{1, 2, 4\}, \phi_3 = \{1, 3, 4\}\}$.
>
> Using the set union, i.e. $\{\texttt{Indices}(c_1)\} \cup \texttt{RemainingSubsets}(c_1, 2)\}$, results in $\{\phi_1 = \{1\}, \phi_2 = \{2, 3\}, \phi_3 = \{2, 4\}, \phi_4 = \{3, 4\}\}$, the parameter index set of the error-constraint and the parameter index sets of all remaining parameters appear in the final set but there is no interaction between them. Therefore, a test adequacy criterion that is based on the set union can guarantee that all strong invalid schemata of c_i and all b-sized valid schemata appear at least once. But it cannot guarantee any interaction between them.

In the following, we discuss how to compute the schemata that must be covered by a test suite. Based on the sets of parameter indices that reflect the interaction between strong invalid schemata and valid schemata, all schemata that must be covered by a test suite to satisfy $(\forall, b)$-wise strong invalid coverage can be computed.

For each $\phi \in \texttt{AllInteractions}(c_i, b)$, the set of all schemata for a parameter index set X_ϕ is computed and filtered to contain only schemata that can be covered by strong invalid test inputs for error-constraint c_i. Therefore, X_ϕ is refined to $X_{c_i,\phi}^{invalid}$.

$$X_{c_i,\phi}^{invalid} = \{s \in X_\phi | \texttt{IsStrongInvalid}(s, c_i)\}$$

In addition, $X_{c_i,d}^{invalid}$ denotes the set that contains all sets of all schemata for all parameter index sets $\phi \in \texttt{AllInteractions}(c_i, b)$.

$$X_{c_i,b}^{invalid} = \bigcup_{\phi \in \texttt{AllInteractions}(c_i,d)} X_{c_i,\phi}^{invalid}$$

> **Running Example** To further illustrate the concepts, let's compute $X_{c_i,\phi}^{invalid}$ for error-constraint c_1 and $\phi = \{1, 2, 4\} \in \texttt{AllInteractions}(c_1, 1)$ of the example (Figure 6.1,

Page 113). Therefore, X_ϕ is computed first and filtered afterwards.

$$X_\phi = \{\text{[Packages:1, Country:UK, Shipping:Standard]},$$
$$\text{[Packages:1, Country:UK, Shipping:Express]},$$
$$\text{[Packages:1, Country:USA, Shipping:Standard]},$$
$$\text{[Packages:1, Country:USA, Shipping:Express]},$$
$$\text{[Packages:1, Country:123, Shipping:Standard]},$$
$$\text{[Packages:1, Country:123, Shipping:Express]},$$
$$...$$
$$\text{[Packages:123, Country:UK, Shipping:Standard]},$$
$$\text{[Packages:123, Country:UK, Shipping:Express]},$$
$$\text{[Packages:123, Country:USA, Shipping:Standard]},$$
$$\text{[Packages:123, Country:USA, Shipping:Express]},$$
$$\text{[Packages:123, Country:123, Shipping:Standard]},$$
$$\text{[Packages:123, Country:123, Shipping:Express]}\}$$

The set of all strong invalid schemata $X_{c_i,\phi}^{invalid}$ for $\phi = \{1, 2, 4\}$ is as follows.

$$X_{c_1,\phi}^{invalid} = \{\text{[Packages:123, Country:UK, Shipping:Standard]},$$
$$\text{[Packages:123, Country:UK, Shipping:Express]},$$
$$\text{[Packages:123, Country:USA, Shipping:Express]}\}$$

The set of all strong invalid schemata $X_{c_i,\phi}^{invalid}$ for $\phi = \{1, 2, 3\}$ is as follows.

$$X_{c_1,\phi}^{invalid} = \{\text{[Packages:123, Country:UK, Phone:+44]},$$
$$\text{[Packages:123, Country:USA, Phone:+1]}\}$$

The set of all strong invalid schemata $X_{c_i,\phi}^{invalid}$ for $\phi = \{1, 3, 4\}$ is as follows.

$$X_{c_1,\phi_3}^{invalid} = \{\text{[Packages:123, Phone:+44, Shipping:Standard]},$$
$$\text{[Packages:123, Phone:+44, Shipping:Express]},$$
$$\text{[Packages:123, Phone:+1, Shipping:Express]}\}$$

It is worth mentioning that the schema [Packages:123, Phone:+1, Shipping:Standard] is excluded from the last set because it is irrelevant. Exclusion-constraint c_5 excludes [Country:USA, Shipping:Standard] and because of error-constraint c_4 [Phone:+1] must be combined with [Country:USA] to be valid. Therefore, [Phone:+1, Shipping:Standard] is irrelevant as well.
$X_{c_1,1}^{invalid}$ is the union of all three $X_{c_1,\phi}^{invalid}$ sets.

A test suite $T \subseteq E$ satisfies the $(\forall, b)$-wise strong invalid coverage for error-constraint $c_i \in C^{err}$ if and only if each schema in $s \in X_{c_i,b}^{invalid}$ is covered by at least one strong invalid test input in T.

$$\forall s \in X_{c_i,b}^{invalid}, \exists \tau \in T \text{ such that } s \subseteq \tau \text{ and } \texttt{IsStrongInvalid}(\tau, c_i).$$

While all strong invalid schemata as specified by error-constraint c_i must be covered by separate strong invalid test inputs, the example also illustrates that strong invalid schemata of different ϕ can be combined such that one strong invalid test input $\tau \in T$ covers several strong invalid schemata $s \in X_{c_i,b}^{invalid}$.

Running Example   Continuing with the example (Figure 6.1, Page 113), when two strong invalid schemata of different $\phi \in \texttt{AllInteractions}(c_i, b)$ share a common strong invalid sub-schema, they can often be combined in one strong invalid test input if no other constraint prohibits the combination. For instance, the following three schemata can be combined to `[Packages:123, Country:UK, Phone:+44, Shipping:Standard]`.

> `[Packages:123, Country:UK, Shipping:Standard]`,
> `[Packages:123, Country:UK, Phone:+44]`,
> `[Packages:123, Phone:+44, Shipping:Standard]`

In total, all strong invalid schemata can be covered by the following three test inputs which satisfy $(\forall, 2)$-wise strong invalid coverage for error-constraint c_1.

`[Packages:123, Country:UK, Phone:+44, Shipping:Standard]`,
`[Packages:123, Country:USA, Phone:+1, Shipping:Express]`,
`[Packages:123, Country:UK, Phone:+44, Shipping:Express]`

Some Strong Invalid Schemata and No Robustness Interactions   The $(\exists, 0)$-wise strong invalid coverage test adequacy criterion describes test suites such that an exception-condition that is described by the error-constraint is falsified at least once and that the described exception should occur at least once during testing. This test adequacy criterion focuses on configuration-independent robustness faults and is useful to test if a particular EH was forgotten or is entirely incorrect such that each strong invalid schema $s \in I_{c_i}$ could detect the fault.

A test suite $T \subseteq E$ satisfies $(\exists, 0)$-wise strong invalid coverage for error-constraint $c_i \in C^{err}$ if and only if there exists a strong invalid schema $s \in I_{c_i}$ as specified by c_i that is covered by at least one strong invalid test input $\tau \in T$.

$$\exists s \in I_{c_i}, \exists \tau \in T \text{ such that } s \subseteq \tau \text{ and } \texttt{IsStrongInvalid}(\tau, c_i)$$

> **Running Example** Error-constraint c_1 as listed in Figure 6.1 (Page 113) specifies one strong invalid schema `[Packages:123]`. It must be covered by at least one strong invalid test input, e.g. `[Packages:123, Country:UK, Phone:+44, Ship-ping:  Standard]`. Since it is the only strong invalid schema $s \in I_{c_i}$, the outcome is equivalent to the "All Strong Invalid Schemata" option.
>
> Error-constraint c_4 specifies two strong invalid schemata `[Country:UK, Phone:+1]` and `[Country:USA, Phone:+44]` of which at least one must be covered by a strong invalid test input, e.g. `[Packages:1, Country:UK, Phone:+1, Shipping:Standard]`. If the corresponding EH (Line 8, Listing 2.1, Page 14) was forgotten, both strong invalid schemata would be able to activate the robustness fault.

Some Strong Invalid Schemata and b-wise Robustness Interactions Our emphasis is on covering strong invalid schemata and combining them to achieve low degrees of robustness interactions. Therefore, we consider this combination of "Some Strong Invalid Schemata" and "b-wise Robustness Interactions" as not meaningful. To satisfy the test adequacy criterion, it would only require one random strong invalid schema $s \in I_{c_i}$ to appear which then would be duplicated and combined with all b-sized valid schemata to achieve the robustness interaction. Instead of duplicating a random strong invalid schema, a-wise strong invalid schemata could result in a similar test suite size but with varying strong invalid schemata. For the sake of completeness, the test adequacy criteria is still defined.

The test adequacy criterion requires that some strong invalid schema $s \in I_{c_i}$ is combined with all b-sized valid schemata of the remaining parameters.

To define it, we can reuse the concepts of previous test adequacy criteria. The function `AllInteractions`(c_i, b) computes all parameter index interactions and $X_{c_i,b}^{invalid}$ encompasses the interactions between all strong invalid schemata in I_{c_i} and all b-sized valid schemata. Since the interactions of only one strong invalid schemata $s \in I_{c_i}$ are required, a subset $X_{s,b}^{invalid} \subseteq X_{c_i,b}^{invalid}$ can be used that encompasses all interactions of $s \in I_{c_i}$ with all b-sized valid schemata.

$$X_{s,b}^{invalid} = \{s' \in X_{c_i,b}^{invalid} | s \subseteq s'\}$$

A test suite $T \subseteq E$ satisfies $(\exists, b)$-wise strong invalid coverage for error-constraint $c_i \in C^{err}$ if and only if a strong invalid schema $s \in I_{c_i}$ as specified by c_i exists for which each schema in $s' \in X_{s,b}^{invalid}$ is covered by at least one strong invalid test input in T.

$$\exists s \in I_{c_i}, \forall s' \in X_{s,b}^{invalid}, \exists \tau \in T \text{ such that } s \subseteq \tau \text{ and } \texttt{IsStrongInvalid}(\tau, c_i).$$

a-wise Strong Invalid Schemata and No Robustness Interactions On the one hand, "All Strong Invalid Schemata" as the most demanding requirement expects all strong invalid schemata as specified by $c_i \in C^{err}$ to be covered by some strong invalid test inputs. For error-constraints with conditions that encompass many parameters and parameters that encompass many values, the set of specified strong invalid schemata I_{c_i} can become very large and testing may become cumbersome if many such error-constraints exist. On the other hand, "Some

Strong Invalid Schemata" only requires one schema $s \in I_{c_i}$ to be covered which might not be extensively enough for EH with complex exception conditions.

In between, we incorporate "a-wise Strong Schemata" and $(a, 0)$-wise strong invalid coverage to utilize the idea of combinatorics to provide the right balance between too little and too many schemata that must be tested where a defines the degree of sub-schemata of I_{c_i} that must be covered. Similar to related work where t-wise coverage is used to test logical expressions in general (Page 57), we use combinatorics to test logical expressions of exception conditions.

Again, $\texttt{Indices}(c_i)$ computes the set of parameter indices for the strong invalid schemata specified in $s \in I_{c_i}$. To compute the a-wise parameter index subsets with $a \in [1, \texttt{Degree}(c_i)]$, the $\texttt{Subsets}$ function can be applied. The lower boundary $a = 1$ is chosen because of our fault model since $a = 0$ would not require any interaction between specified parameter indices. The upper boundary $a = \texttt{Degree}(c_i)$ is a natural limit that requires all specified parameter indices to interact with each other. As an abbreviation, we use the function $\texttt{SpecifiedSubsets}(c_i, a)$ to compute all a-sized subsets of parameter indices as specified by $c_i \in C^{err}$.

$$\texttt{SpecifiedSubsets}(c_i, a) = \texttt{Subsets}(\texttt{Indices}(c_i), a)$$

> **Running Example** Error-constraint c_4 of the example (Figure 6.1, Page 113) specifies strong invalid schemata I_{c_4} for two parameters $\texttt{Indices}(c_4) = \phi = \{2, 3\}$. For $a = 1$ and $a = 2$, the specified subsets are as follows.
>
> $$\texttt{SpecifiedSubsets}(c_4, 1) = \{\phi_1 = \{2\}, \phi_2 = \{3\}\}$$
> $$\texttt{SpecifiedSubsets}(c_4, 2) = \{\phi_1 = \{2, 3\}\}$$
>
> For $a = 2$, there is only 2-sized subset which is equivalent to the entire set of parameter indices $\texttt{Indices}(c_4)$.

Based on the a-wise parameter index subsets, the schemata that must be covered by strong invalid test inputs for c_i can be computed. For each a-sized parameter index subset $\phi \in \texttt{SpecifiedSubsets}(c_i, a)$, the set of all invalid schemata $X_{c_i, \phi}^{invalid}$ (Page 132) for parameter indices ϕ is used again.

When using $X_{c_i, \phi}^{invalid}$ for a $\phi \in \texttt{SpecifiedSubsets}(c_i, a)$, it denotes the set of all a-sized sub-schemata that must be covered by strong invalid test inputs for ϕ and error-constraint c_i.

Further, $X_{c_i, a}^{invalid}$ denotes the set of all $X_{c_i, \phi}^{invalid}$ subsets for all $\phi \in \texttt{SpecifiedSubsets}(c_i, a)$.

$$X_{c_i, a}^{invalid} = \bigcup_{\phi \in \texttt{SpecifiedSubsets}(c_i, a)} X_{c_i, \phi}^{invalid}$$

The schemata in $X_{c_i, a}^{invalid}$ are itself not necessarily invalid but they can be extended to strong invalid schemata that contain parameter-value pairs for all parameters in $\texttt{Indices}(c_i)$.

> **Running Example** For error-constraint c_4 of the example (Figure 6.1, Page 113), the 1-sized subsets of specified parameter indices are $\texttt{SpecifiedSubsets}(c_4, 1) = \{\phi_1 =$

$\{2\}, \phi_2 = \{3\}\}$. The sets of invalid schemata $X_{c_i,\phi}^{invalid}$ are as follows.

$$X_{c_4,\phi_1}^{invalid} = \{\texttt{[Country:UK]}, \texttt{[Country:USA]}\} \qquad \text{for } \phi_1 = \{2\}$$
$$X_{c_4,\phi_2}^{invalid} = \{\texttt{[Phone:+44]}, \texttt{[Phone:+1]}\} \qquad \text{for } \phi_2 = \{3\}$$

The set of all schemata that must be covered $X_{c_i,1}^{invalid}$ is the union of $X_{c_i,\phi_1}^{invalid}$ and $X_{c_i,\phi_2}^{invalid}$. The schemata themselves are not necessarily invalid but their combinations must be strong invalid.

A test suite $T \subseteq E$ satisfies the $(a,0)$-wise strong invalid coverage for error-constraint $c_i \in C^{err}$ if and only if each schema in $X_{c_i,a}^{invalid}$ is covered by at least one strong invalid test input in T.

$$\forall s \in X_{c_i,a}^{invalid}, \exists \tau \in T \text{ such that } s \subseteq \tau \land \texttt{IsStrongInvalid}(\tau, c_i).$$

Running Example In the example depicted in Figure 6.1 (Page 113), the schemata of $X_{c_4,1}^{invalid}$ can only be combined in one way to be covered by strong invalid test inputs for error-constraint c_4:

$$\texttt{[Country:UK, Phone:+1]},$$
$$\texttt{[Country:USA, Phone:+44]}$$

The schemata of $X_{c_4,1}^{invalid}$ can be covered by strong invalid test inputs such as the following to satisfy $(1,0)$-wise strong invalid coverage.

```
[Packages:1, Country:UK, Phone:+1, Shipping:Express],
[Packages:1, Country:USA, Phone:+44, Shipping:Express]
```

For this particular example, the strong invalid schemata that must be covered to satisfy $(1,0)$-wise strong invalid coverage are equivalent to the strong invalid schemata that must be covered to satisfy $(\forall, 0)$-wise strong invalid coverage.

To illustrate the difference, we extend the example and introduce Germany as another country with +49 as its phone code. The extended RIPM is depicted in Figure 7.3 (Page 138) where the values GER and +49 are added and the constraints c_4 and c_5 are updated. As a result, I_{c_4} encompasses six instead of two strong invalid schemata. All of them must be covered by six distinct strong invalid test inputs to satisfy $(\forall, 0)$-wise or $(2, 0)$-wise strong invalid coverage.

$$I_{c_4} = \{\texttt{[Country:UK: Phone:+1]},$$
$$\texttt{[Country:UK, Phone:+49]},$$
$$\texttt{[Country:USA, Phone:+44]},$$
$$\texttt{[Country:USA, Phone:+49]},$$

```
Parameters:
  p₁ : Packages:          1, 3, 123
  p₂ : Country:           UK, USA, GER, 123
  p₃ : Phone:             +44, +1, +49, 123
  p₄ : Shipping:          Standard, Express
Error-Constraints:
  c₁ : Packages ≠ 123
  c₂ : Country ≠ 123
  c₃ : Phone ≠ 123
  c₄ : ¬(Country = UK ∧ Phone = +1) ∧ ¬(Country = UK ∧ Phone = +49)
     ∧¬(Country = USA ∧ Phone = +44) ∧ ¬(Country = USA ∧ Phone = +49)
     ∧¬(Country = GER ∧ Phone = +1) ∧ ¬(Country = GER ∧ Phone = +44)
Exclusion-Constraints:
  c₅ : Shipping = Standard ⇒ (Country ≠ USA ∧ Country ≠ GER)
```

Figure 7.3.: RIPM for the Ordering Web-Service Example

```
                    [Country:GER, Phone:+1],
                    [Country:GER, Phone:+44]}
```

In contrast, $(1, 0)$-wise strong invalid coverage requires fewer test inputs for its satisfaction. Again, the specified subsets of parameter indices for error-constraint c_4 are `SpecifiedSubsets(c₄,1) = {φ₁ = {2}, φ₂ = {3}}`. The sets of schemata that must be covered are as follows.

$$X_{c_4,\phi_1}^{invalid} = \big\{\texttt{[Country:UK]}, \texttt{[Country:USA]}, \texttt{[Country:GER]}\big\} \quad \text{for } \phi_1 = \{2\}$$
$$X_{c_4,\phi_2}^{invalid} = \big\{\texttt{[Phone:+44]}, \texttt{[Phone:+1]}, \texttt{[Phone:+49]}\big\} \qquad \text{for } \phi_2 = \{3\}$$

To satisfy $(1, 0)$-wise strong invalid coverage, the schemata can be covered in different ways. For instance, by the following three strong invalid test inputs.

```
[Packages:1, Country:UK: Phone:+44, Shipping:Express],
[Packages:1, Country:USA, Phone:+49, Shipping:Express],
[Packages:1, Country:GER, Phone:+1, Shipping:Express]
```

a-wise Strong Invalid Schemata and b-wise Robustness Interactions   The options "a-wise Strong Invalid Schemata" and "b-wise Robustness Interactions" reflect the idea of t-wise combinatorics in both dimensions of (a, b)-factor robustness faults. The (a, b)-wise strong invalid coverage combines both dimensions into one test adequacy criterion.

Again, $a \in [1, \texttt{Degree}(c_i)]$ determines the degree of sub-schemata of I_{c_i} that must be covered by strong invalid test inputs. In addition, $b \in [0, n - \texttt{Degree}(c_i)]$ determines the degree of valid

schemata that must interact with the a-sized sub-schemata of I_{c_i}.

> **Running Example**   Continuing with the extended example of Figure 7.3 (Page 138), the parameter index sets that must interact to satisfy $(1, 1)$-wise strong invalid coverage for error-constraint c_4 are as follows:
>
> $$\texttt{SpecifiedSubsets}(c_4, 1) = \{\phi_1 = \{2\}, \phi_2 = \{3\}\}$$
> $$\texttt{RemainingSubsets}(c_4, 1) = \{\phi_1 = \{1\}, \phi_2 = \{4\}\}$$

To compute parameter interactions, $\texttt{AllInteractions}(c_i, b)$ cannot be used since it relies on a fixed set of all specified parameter indices, i.e. $\texttt{SpecifiedSubsets}(c_i, \texttt{Indices}(c_i)) = \texttt{Indices}(c_i)$, instead of a-sized sub-schemata. Therefore, we introduce $\texttt{Interactions}(c_i, a, b)$ as a generalization.

$$\texttt{Interactions}(c_i, a, b) = \{\phi_a \cup \phi_b | \phi_a \in \texttt{SpecifiedSubsets}(c_i, a)$$
$$\wedge\, \phi_b \in \texttt{RemainingSubsets}(c_i, b)\}$$

> **Running Example**   Continuing with the extended example of Figure 7.3 (Page 138), the interactions between the parameter index sets $\texttt{RemainingSubsets}(c_4, 1)$ and $\texttt{SpecifiedSubsets}(c_4, 1)$ are as follows.
>
> $$\texttt{Interactions}(c_4, 1, 1) = \{\phi_1 = \{2, 1\}, \phi_2 = \{2, 4\}, \phi_3 = \{3, 1\}, \phi_4 = \{3, 4\}\}$$

With $a = \texttt{Degree}(c_i)$, the computation of $\texttt{Interactions}$ is equivalent to $\texttt{AllInteractions}$ and reflects the "All Strong Invalid Schemata" option. With $b = 0$, the test adequacy criterion reflects the "No Robustness Interactions" option. The "All Robustness Interactions" option is reflected by $b = n - \texttt{Degree}(c_i)$. When $a = \texttt{Degree}(c_i)$ and $b = n - \texttt{Degree}(c_i)$ are used for all error-constraints, the test adequacy criteria are equivalent to the N-wise strong invalid coverage. For each interaction $\phi \in \texttt{Interactions}(c_i, a, b)$, the set of strong invalid all schemata that must be covered by strong invalid test inputs is again determined by $X_{c_i, \phi}^{invalid}$. Additionally, $X_{c_i, a, b}^{invalid}$ encompasses all sets of strong invalid schemata $X_{c_i, \phi}^{invalid}$ for all $\phi \in \texttt{Interactions}(c_i, a, b)$.

$$X_{c_i, a, b}^{invalid} = \bigcup_{\phi \in \texttt{Interactions}(c_i, a, b)} X_{c_i, \phi}^{invalid}$$

A test suite $T \subseteq E$ satisfies (a, b)-wise strong invalid coverage for error-constraint $c_i \in C^{err}$ if and only if each schema in $X_{c_i, a, b}^{invalid}$ is covered by at least one strong invalid test input in T.

$$\forall s \in X_{c_i, a, b}^{invalid}, \exists \tau \in T \text{ such that } s \subseteq \tau \wedge \texttt{IsStrongInvalid}(\tau, c_i).$$

> **Running Example**   Analogously to previous test adequacy criteria, the sets $X_{c_i, \phi}^{invalid}$ for all $\phi \in \texttt{Interactions}(c_i, a, b)$ must be computed and a test suite must be found that

Figure 7.4.: Subsumption Hierarchy extended with Combinatorial Test Adequacy Criteria for
          Strong Invalid Test Inputs

covers all schemata as defined by them. For instance, the schemata that must be covered for error-constraint c_4 and $\phi = \{2, 1\} \in \texttt{Interactions}(c_4, 1, 1)$ of the extended example (Figure 7.3, Page 138) are as follows.

$$X_{c_4,\phi}^{invalid} = \big\{ \texttt{[Packages:1, Country:UK]},$$
$$\texttt{[Packages:1, Country:USA]},$$
$$\texttt{[Packages:1, Country:GER]},$$
$$\texttt{[Packages:3, Country:UK]},$$
$$\texttt{[Packages:3, Country:USA]},$$
$$\texttt{[Packages:3, Country:GER]}\big\}$$

Hierarchy of Test Adequacy Criteria   In the previous paragraphs, six new test adequacy criteria for testing with strong invalid test inputs are introduced. Depending on the expectations, the rigor of a test suite can be controlled and the two types of configuration-independent and configuration-dependent robustness faults can be targeted. In order to better understand the test adequacy criteria, we illustrate the hierarchy of and relationships between them and relate them to existing test adequacy criteria. Figure 7.4 (Page 140) depicts the extended subsumption

hierarchy. The previously discussed test adequacy criteria are indicated by a gray background.

The new test adequacy criteria can be selected individually for each error-constraint while the previously discussed test adequacy criteria are selected for the entire RIPM. Therefore, they cannot be directly compared to the previously discussed test adequacy criteria.

The N-wise strong invalid coverage can be selected for the entire RIPM and subsumes all six new test adequacy criteria. $(\forall, b)$-wise strong invalid coverage is the most comprehensive test adequacy criteria. When choosing $(\forall, b)$-wise strong invalid coverage with $b = n - \texttt{Degree}(c_i)$ for all error-constraints, it is equivalent to N-wise strong invalid coverage. When chosen with smaller b values, it is subsumed by N-wise strong invalid coverage. When chosen with $b > 0$, it subsumes $(\forall, 0)$-wise strong invalid coverage and with $b = 0$, it is equivalent to $(\forall, 0)$-wise strong invalid coverage.

The (a, b)-wise strong invalid coverage is the most versatile test adequacy criteria which is equivalent to $(\forall, b)$-wise strong invalid coverage with $a = \texttt{Degree}(c_i)$. (a, b)-wise strong invalid coverage subsumes $(a, 0)$-wise strong invalid coverage with $b > 0$ and is equivalent to $(a, 0)$-wise strong invalid coverage with $b = 0$. For smaller a values, (a, b)-wise strong invalid coverage is subsumed by $(\forall, b)$-wise strong invalid coverage and $(a, 0)$-wise strong invalid coverage is subsumed by $(\forall, 0)$-wise strong invalid coverage.

$(\exists, b)$ strong invalid coverage and $(\exists, 0)$ strong invalid coverage are the least demanding test adequacy criteria since only one invalid schema $s \in I_{c_i}$ must be covered for their satisfaction. If there is only one invalid schema defined by I_{c_i}, $(\exists, b)$ strong invalid coverage and $(\exists, 0)$ are equivalent to $(\forall, b)$ strong invalid coverage and $(\forall, 0)$ and also equivalent to (a, b) and $(a, 0)$ strong invalid coverage. Otherwise, they are subsumed by (a, b)-wise strong invalid coverage and $(a, 0)$-wise strong invalid coverage.

Not only the relationships between the new test adequacy criteria can be discussed. The new test adequacy criteria can also be related to base-choice and single error coverage.

Again, base-choice coverage can subsume each-choice relevant coverage and each-choice valid coverage when all derived test inputs for relevant or valid values are itself relevant or valid, respectively. Therefore, it is required that the base test input is relevant or valid as well. In addition, base-choice coverage can also subsume single error coverage when the base test input is valid and when all derived test inputs for invalid values are strong invalid.

Strictly speaking, single error coverage is defined for invalid values only. To ensure a minimum of interaction $(a, 0)$-wise strong invalid coverage is defined for $a \in [1, \texttt{Degree}(c_i)]$. When chosen for all error-constraints, $a \geq 1$ requires each invalid value to be covered which means that $(a, 0)$-wise strong invalid coverage subsumes single error coverage. When no invalid value combinations exist, both test adequacy criteria are equivalent.

When all invalid value combinations can be composed by replacing one value of the base test input, base-choice coverage can be equivalent to $(a, 0)$-wise strong invalid coverage. Since this is a border case, the relationship is not depicted in Figure 7.4 (Page 140).

Shielding parameters and existential coverage are not directly relatable for the reasons discussed in the chapter on related work (Chapter 4, Page 58 and following). As discussed in the chapter on related work as well, variable strength coverage is a generalization of t-wise coverage which can be parameterized to define any parameter interaction. Therefore, it can also be parameterized with the parameter interactions that are computed and defined by the new test adequacy

criteria. Then, variable strength coverage is equivalent to the new test adequacy criteria.

In the next chapter, we discuss and introduce test selection strategies to select strong invalid test inputs such that they satisfy these new test adequacy criteria.

List of Symbols

$\texttt{Complement}(\phi)$	function to compute the complement of ϕ_d relative to ϕ_n
$\texttt{RemainingSubsets}(c_i, b)$	function to compute the set of all sets of b-sized remaining parameter indices
$\texttt{AllInteractions}(c_i, b)$	function to compute the sets of parameter indices with b-wise robustness interaction for c_i
$X_{c_i,\phi}^{invalid}$	set of all strong invalid schemata for parameters in ϕ and error-constraint c_i
$X_{c_i,b}^{invalid}$	set of sets of all strong invalid schemata $X_{c_i,\phi}^{invalid}$ for all $\phi \in \texttt{AllInteractions}(c_i, b)$
$X_{s,b}^{invalid}$	set of all strong invalid schemata for $s \in I_{c_i}$
$\texttt{SpecifiedSubsets}(c_i, a)$	function to compute the set of all sets of a-sized parameter indices for $\texttt{Indices}(c_i)$
$X_{c_i,a}^{invalid}$	set of sets of all schemata $X_{c_i,\phi}^{invalid}$ for all $\phi \in \texttt{SpecifiedSubsets}(c_i, a)$
$\texttt{Interactions}(c_i, a, b)$	function to compute the a- and b-wise parameter index interactions for error-constraint c_i
$X_{c_i,a,b}^{invalid}$	set of sets of all schemata $X_{c_i,\phi}^{invalid}$ for all $\phi \in \texttt{Interactions}(c_i, a, b)$

Chapter 8.

Combinatorial Robustness Test Selection Strategies

Contents

With the RIPM structure and the defined test adequacy criteria for valid strong invalid test inputs, the next step is to define test selection strategies to select valid and strong invalid test inputs such that the test adequacy criteria are satisfied.

Therefore, we first discuss how existing test selection strategies can be used to select valid test inputs. Afterwards, we discuss modifications and present a family of test selection strategies to select strong invalid test inputs to satisfy the six newly introduced test adequacy criteria.

8.1. Test Selecting Strategies for Valid Test Inputs

The challenge of test selection strategies is to find a small test suite $T \subseteq E$ within a reasonable time such that T satisfies the chosen test adequacy criteria [CDS07]. Due to the formal definitions of the strong invalid coverage test adequacy criteria, it is already clear which schemata must be covered and which schemata must not be combined to a single test input. The remaining challenge is to combine all the schemata to test inputs within a reasonable time.

LEI & TAI [LT98] proved that the selection of minimal test suites is an NP-complete problem. As a consequence, many different test selection strategies are proposed. In general, the classes of algebraic construction, meta-heuristic search, and greedy algorithms can be distinguished [CDS07]. The surveys of GRINDAL et al. [GOA05] and NIE & LEUNG [NL11a] provide a comprehensive overview and classification of existing test selection strategies.

Greedy algorithms are a class of algorithms that typically produce small but not minimal test suites with reasonable time requirements [CDS07]; [NL11a]. Greedy algorithms are applicable to a wide range of IPMs if not to all IPMs and they support constraint handling.

Many greedy algorithms generate "one test input at a time" and are variations of the `AETG` (cf. [Coh+94]; [Coh+97]) test selection strategy [CDS07]. A second type of greedy algorithm follows the "one parameter at a time" principle. These algorithms are typically variations of the `IPO` test selection strategy (cf. [LT98]). `IPO`-based test selection strategies are deterministic,

i.e. the test suites selected for a particular IPM are always the same [GOA05]. In contrast, AETG-based test selection strategies are non-deterministic. Further on, LEI & TAI [LT98]; [Lei+07] showed that IPO has a superior time complexity to AETG [GOA05]; [KS18].

Therefore, we develop our test selection strategies as greedy algorithms and base them on the IPO family of test selection strategies.

IPO [LT98] was initially introduced as a test selection strategy that had a fixed pair-wise testing strength. It was later extended to IPOG [Lei+07] which is a generalization with an adjustable testing strength $t \in [1, n]$ and to IPOG-C [Yu+13a] which incorporates constraint handling such that the selected test inputs satisfy all constraints in $\mathbb{C}$. In order to generate a test suite that satisfies t-wise valid coverage, it is sufficient to use IPOG-C with $\mathbb{C} = C^{ex} \cup C^{err}$.

In the following, the IPOG-C test selection strategy is explained in more detail. Since the RIPM and strong invalid coverage test adequacy criteria also rely on constraints, it is modified in the next section to create strong invalid test inputs that satisfy our test adequacy criteria.

```
 1 algorithm: IPOG-C
 2 input: ℙ, 𝕍, ℂ, t
 3 output: A set of test inputs 𝕋
 4 let  φ_n ← {r_1,...,r_n}
 5 let  φ_initial ← {r_1,...,r_t}
 6 let  X_{φ_initial} ← compute-schemata(φ_initial, 𝕍, ℂ)
 7 let  𝕋 ← X_{φ_initial}
 8
 9 foreach j in [t+1,n]
10    let  φ_previous ← {r_1,...,r_{j-1}}
11    let  Φ_t ← next-interactions(φ_previous, r_j, t)
12    let  X_t ← compute-all-schemata(Φ_t, 𝕍, ℂ)
13    𝕋 ← horizontal-extension(𝕋, X_t)
14    if(uncovered-schemata-exist(𝕋, X_t))
15       𝕋 ← vertical-extension(𝕋, X_t)
16
17 𝕋 ← replace-do-not-care-values(𝕋)
18
19 return 𝕋
```

Listing 8.1: IPOG-C Test Selection Strategy

Listing 8.1 (Page 144) depicts the IPOG-C test selection strategy in pseudo-code. It is adapted from YU et al. [Yu+13a] and KLEINE & SIMOS [KS18]. The description reuses the formalism that we introduced in the discussion about test adequacy criteria.

The first two statements (Lines 3 and 4) define two variables that contain sets of parameter indices. The algorithm requires all parameter indices to be sorted in decreasing order of the value domain sizes, i.e. $\forall r_i, r_{i+1} \in \phi, |V_{r_i}| \geq |V_{r_{i+1}}|$. The variable ϕ_n contains all n parameter indices and $\phi_{initial}$ contains the initial t parameter indices.

The set $X_{\phi_{initial}}$ contains the initial schemata for the parameter indices in $\phi_{initial}$.

Therefore, the function compute-schemata computes the subset of schemata for ϕ that

satisfies the constraints in $\mathbb{C}$.

$$\texttt{compute-schemata}(\phi, \mathbb{V}, \mathbb{C}) = \{s \in X_\phi | \texttt{SAT}(\mathbb{C}, s) = true\}$$

Although comparable to $X_\phi^{relevant}$ and X_ϕ^{valid} for $\phi = \phi_{initial}$ (See Page 120), the algorithm relies on a different set of constraints $\mathbb{C}$ which makes it independent from the IPM or RIPM in the sense that it is not fixed to C^{ex} or C^{err}. Depending on the context, the user can choose which constraint are passed to the algorithm. If the user chooses $\mathbb{C} = C^{ex}$, $X_{\phi_{initial}}$ is equivalent to $X_{\phi_{initial}}^{relevant}$. If the user chooses $\mathbb{C} = C^{ex} \cup C^{err}$, $X_{\phi_{initial}}$ is equivalent to $X_{\phi_{initial}}^{valid}$.

The set $X_{\phi_{initial}}$ functions as the initial test suite $\mathbb{T}$ for the first t parameters. Afterwards, the initial test suite is extended with parameter-value pairs for the remaining $n - t$ parameters one after another. In each iteration, the variable $\phi_{previous}$ denotes a set of parameter indices for the previously extended parameters and r_j denotes the current parameter index. It starts with the parameter indices of $\phi_{initial}$. With each iteration of the loop, it is then continuously extended.

In total, all t-sized interactions between the parameter indices as computed by $\texttt{Subsets}(\phi_\texttt{n}, \texttt{t})$ must be considered. Due to the step-wise iterative extension of an initial test suite, only the interactions between the previous parameter indices $\phi_{previous}$ and the current parameter index r_j are of interest. These interactions are computed by the $\texttt{next-interactions}$ function. It is the subset of $\texttt{Subsets}(\phi_\texttt{n}, \texttt{t})$ that is exclusively related to r_j and does not cover any other parameter indices than r_j and $\phi_{previous}$.

$$\texttt{next-interactions}(\phi_{previous}, r_j, t)$$
$$= \{s \in \texttt{Subsets}(\phi_\texttt{n}, \texttt{t}) | r_j \in s \land s \backslash (\{r_j\} \cup \phi_{previous}) = \emptyset\}$$

Running Example   To illustrate the "one parameter at a time" principle, we oppose the interactions between parameters as computed by t-wise coverage with the interactions as computed by the $\texttt{IPOG-C}$ test selection strategy.

For the test adequacy criterion, all 2-wise parameter index subsets are computed by $\texttt{Subsets}(\phi_n, 2)$ in one step. For the four parameters of the running example, the 2-wise interactions are as follows.

$$\Phi_2 = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$$

For the $\texttt{IPOG-C}$ test selection strategy, the interactions are computed step by step. First of all, ϕ_n is ordered according to the sizes of the value domains. For the RIPM of Figure 6.1 (Page 113), the ordering is $\phi_n = \{r_1 = 2, r_2 = 3, r_3 = 1, r_4 = 4\}$. The initial test suite covers values for $\phi_{initial} = \{2, 3\}$.

$\phi_{previous} = \{2, 3\}, r_3 = 1, \Phi_2 = \{\{1, 2\}, \{1, 3\}\}$	for iteration j = 3
$\phi_{previous} = \{2, 3, 1\}, r_4 = 4, \Phi_2 = \{\{2, 4\}, \{3, 4\}, \{1, 4\}\}$	for iteration j = 4

The same interactions are computed for the test adequacy criterion and for the test selection strategy. For the algorithm, it is simply divided into several iterations.

Based on the interactions Φ_t, all schemata X_t that must be covered during this iteration can be computed. The function `compute-all-schemata` computes the schemata for all $\phi \in \Phi_t$ such that they satisfy the constraints $\mathbb{C}$.

$$\texttt{compute-all-schemata}(\Phi_d, \mathbb{V}, \mathbb{C}) = \bigcup_{\phi \in \Phi_d} \texttt{compute-schemata}(\phi, \mathbb{V}, \mathbb{C})$$

Up to this point, the algorithm computes – in an iterative manner – all schemata that must be covered by test inputs similar to what is defined by X_t^{valid}. In the remaining part of the algorithm, the schemata of the initial test suite $\tau \in \mathbb{T}$ are – again in an iterative manner – extended with parameter-value pairs such that all schemata $s \in X_t$ are covered and all schemata of the test suite $s \in \mathbb{T}$ satisfy the constraints $\mathbb{C}$.

The extension of $\mathbb{T}$ with parameter-value pairs for r_j is further divided into a horizontal extension and a vertical extension. During the horizontal extension, each existing schema of the test suite $\tau \in \mathbb{T}$ is extended with a value for parameter p_{r_j}. Ideally[1], a value $v \in V_{r_j}$ is selected such that the extended schema $\tau' \in \mathbb{T}$ covers many yet uncovered schemata $s \in X_t$.

The horizontal extension ends when all schemata $s \in X_t$ are covered or when all schemata $\tau \in \mathbb{T}$ are extended. In the first case, all schemata $s \in X_t$ are covered while perhaps not all schemata $\tau \in \mathbb{T}$ are yet extended. Then, the remaining schemata $\tau \in \mathbb{T}$ are extended with a "do not care" value for the current parameter because the actual assignment is not important for the coverage.

When all schemata $s \in X_t$ are covered after horizontal extension, the vertical extension is skipped. Otherwise, when there are some schemata $s \in X_t$ left uncovered by $\mathbb{T}$, the vertical extension adds them to the test suite. First, it is checked if the uncovered schemata can be added to existing schemata $\tau \in \mathbb{T}$ by replacing "do not care" values with actual values. Afterwards, the remaining uncovered schemata are added as new schemata to $\mathbb{T}$. The new schemata $\tau \in \mathbb{T}$ only cover the remaining schemata of $s \in X_t$. For all other previous parameters, the "do not care" is assigned.

After all parameter indices have been iterated and the test suite $\mathbb{T}$ is extended such that it covers parameter-value pairs for all n parameters, the function `replace-do-not-care-values` replaces "do not care" values with actual values.

The algorithm also supports two edge cases $t = n$ and $t = 0$. For $t = n$, the initial test suite (Listing 8.1, Page 144, Lines 5-7) already covers all schemata and the loop is not entered since there are no remaining parameter indices. For $t = 0$, the initial set of parameter indices $\phi_{initial}$, the initial test suite $\mathbb{T}$ as well as the computed interactions Φ_t are all empty. Using the algorithm in this form with $t = 0$ does not make sense because it produces an empty test suite. However, it is important to acknowledge that the computation for $t = 0$ produces a valid result because it becomes relevant for the generation of strong invalid test inputs.

---

[1] Please note, this is an over-simplification on how to choose a value to extend a particular schema. A more detailed discussion is beyond this description. Please refer to related work for more information (cf. [Gao+14]; [Gao+15]).

8.2. **Test Selection Strategies for Strong Invalid Test Inputs**

In the following, a family of test selection strategies for the six test adequacy criteria is discussed. The strategies are based on the `IPOG-C` algorithm and utilizes the same horizontal and vertical extension. But, the computations of interactions between parameter indices and the construction of the initial test suites are modified.

Analogously to the test adequacy criteria, the test selection strategy for (a, b)-wise strong invalid coverage is the most versatile strategy that can be used to generate test suites that satisfy all six test adequacy criteria. For greater comprehensibility, all six test selection strategies are discussed consecutively to introduce individual aspects.

Since strong invalid coverage is defined in relation to a single error-constraint, the entire selection of strong invalid test inputs can be understood as an iteration over all error-constraints. In Listing 8.2 (Page 147), such an extensive test selection strategy is depicted. We denote it as `ROBUSTA` which combines the test selection to satisfy t-wise valid coverage with the test selection to satisfy the combinatorial robustness test adequacy criteria for strong invalid test inputs.

```
1  algorithm: ROBUSTA
2  input: ℙ, 𝕍, ℂ^ex, ℂ^err, t, 𝔸, 𝔹
3  output: A set of test inputs 𝕋
4  let 𝕋⁺ ← IPOG-C(ℙ, 𝕍, ℂ^ex ∪ ℂ^err, t)
5  let 𝕋⁻ ← ∅
6
7  foreach c_i in ℂ^err
8     if (a_i ≠ 0)
9        let ℂ̄ ← ℂ^ex ∪ modify-⟨∗,∗⟩(ℂ^err, c_i)
10       𝕋⁻ ← 𝕋⁻ ∪ IPOG-NEG-⟨∗,∗⟩(ℙ, 𝕍, ℂ̄, a_i, b_i)
11
12 return 𝕋⁺ ∪ 𝕋⁻
```

Listing 8.2: `ROBUSTA` Test Selection Strategy

The `ROBUSTA` test selection strategy is a higher level test selection strategy that delegates the actual test input selection to `IPOG-C` and variations of `IPOG-C`.

After selecting valid test inputs via `IPOG-C` (Line 3), a loop iterates over all error-constraints to select strong invalid test inputs for each error-constraint separately. Therefore, two new sets $\mathbb{A}$ and $\mathbb{B}$ are introduced which contain the robustness degree a_i and robustness interaction degree b_i for each error-constraint c_i. The robustness degree $a_i = 0$ can be used to indicate that the selection for one error-constraint should be skipped. If the selection is not skipped, $\overline{\mathbb{C}}$ represents a set of constraints with a modified error-constraint c_i that allows selecting strong invalid test inputs for c_i. The strong invalid test inputs are selected by invoking a `IPOG-C` variation. The details of `modify-⟨∗,∗⟩` and `IPOG-NEG⟨∗,∗⟩` depend on the chosen test adequacy criterion for c_i. They are discussed subsequently.

All Strong Invalid Schemata and No Robustness Interactions   To select test inputs such that $\langle \forall, 0 \rangle$-wise strong invalid coverage is satisfied for error-constraint c_i, the negation of c_i must be used.

$$\text{modify-}\langle \forall, 0 \rangle (\mathbb{C}^{err}, c_i) = (\mathbb{C}^{err} \backslash \{c_i\}) \cup \{\overline{c_i}\}$$

Listing 8.3 (Page 148) depicts the IPOG-NEG-$\langle \forall, 0 \rangle$ test selection strategy. For $\langle \forall, 0 \rangle$-wise strong invalid coverage, the values for a and b are always fixed to $a = \text{Degree}(c_i)$ and $b = 0$. But to achieve a common interface among all IPOG-NEG-$\langle *, * \rangle$ variations, the values are still passed as arguments. Although, the arguments are not used within the test selection strategy.

The negation of c_i ensures that only strong invalid test inputs are selected. But the negation is not sufficient to ensure that each specified strong invalid schemata $s \in I_{c_i}$ appears in at least one strong invalid test input. Further on, the robustness interaction degree $b = 0$ must be incorporated. Therefore, additional modifications of IPOG-C are required. The changes to the algorithm are highlighted by borders.

```
 1   algorithm: IPOG-NEG-⟨∀,0⟩
 2   input: ℙ, 𝕍, ℂ, a, b
 3   output: A set of test inputs 𝕋⁻
 4   let  φₙ ←  {r₁,...,r_degree(c̄ᵢ),...,rₙ}
 5   let  φ_initial ←  indices(c̄ᵢ)
 6   let  X_φ_initial ← compute-schemata(φ_initial, 𝕍, ℂ)
 7   let  𝕋⁻ ← X_φ_initial
 8
 9   foreach j in  [degree(c̄ᵢ)+1,n]
10      let  φ_previous ← {r₁,...,r_{j-1}}
11      let  Φ₀ ←  ∅
12      let  X₀ ← compute-all-schemata(Φ₀, 𝕍, ℂ)
13      𝕋⁻ ← horizontal-extension(𝕋⁻, X₀)
14      if(uncovered-schemata-exist(𝕋⁻, X₀))
15         𝕋⁻ ← vertical-extension(𝕋⁻, X₀)
16
17   𝕋⁻ ← replace-do-not-care-values(𝕋⁻)
18
19   return 𝕋⁻
```

Listing 8.3: IPOG-NEG-$\langle \forall, 0 \rangle$ Test Selection Strategy

The first modification ensures that each specified strong invalid schemata $s \in I_{c_i}$ appears in at least strong invalid test input. It encompasses the definition of the initial test suite $\mathbb{T}^-$ (Lines 4 to 7). Instead of using the Cartesian product of the value domains of the first t parameters, all value domains of parameters that are involved in the condition of error-constraint c_i are used (Line 5). Because of the negation of error-constraint c_i, the resulting Cartesian product is equivalent to the set of specified strong invalid schemata I_{c_i} (Line 6).

Since $\phi_{initial}$ is changed, the ordering of parameter indices (Line 4) must be adjusted accordingly to ensure an iteration over all remaining parameter indices $\phi_n \backslash \phi_{initial}$. In `IPOG-C`, it was ensured by ordering of parameter indices $r_j \in \phi_n$ such that $\phi_{initial} = \{r_1, ..., r_t\}$ covers the first t parameter indices. To maintain the same behavior, we refine the ordering such that the first parameter indices are covered by $\phi_{initial} = \{r_1, ..., r_{\mathtt{degree}(\overline{c_i})}\}$. Further on, the loop starts iterating at $j = \mathtt{degree}(\overline{c_i}) + 1$ instead of $j = t + 1$.

> **Running Example** For the RIPM of Figure 6.1 (Page 113), the `IPOG-C` ordering of parameter indices and the initial parameter indices for $t = 2$ are as follows.
>
> $$\phi_n = \{r_1 = 2, r_2 = 3, r_3 = 1, r_4 = 4$$
> $$\phi_{initial} = \{r_1 = 2, r_2 = 3\}$$
>
> For the `IPOG-NEG-`$\langle \forall, 0 \rangle$ variations, the parameter index sets depend on the current error-constraint c_i. For error-constraint c_1, they are as follows.
>
> $$\phi_n = \{r_1 = 1, r_2 = 2, r_3 = 3, r_4 = 4$$
> $$\phi_{initial} = \{r_1 = 1\}$$
>
> For error-constraint c_4, they are as follows.
>
> $$\phi_n = \{r_1 = 2, r_2 = 3, r_3 = 1, r_4 = 4$$
> $$\phi_{initial} = \{r_1 = 2, r_2 = 3\}$$

The second modification removes the computation of interactions because no interactions are required. It encompasses the definition of parameter index interactions Φ_0 for schemata X_0 that must be covered by the test suite (Lines 10 to 12). Since no robustness interactions are required, zero interactions are computed and no additional schemata must be covered by the test suite.

The horizontal extension is still required to extend the initial test suite until it contains parameter-value pairs for all n parameters. Since no additional schemata of X_0 must be covered, all schemata $\tau \in \mathbb{T}^-$ are extended with "do not care" values. The vertical extension is skipped. Afterwards, all "do not care" values are replaced before the final test suite is returned.

All Strong Invalid Schemata and b-wise Robustness Interactions To select test inputs such that $\langle \forall, b \rangle$-wise strong invalid coverage is satisfied for error-constraint c_i, the negation of c_i must be used again.

$$\mathtt{modify\text{-}}\langle \forall, b \rangle (\mathbb{C}^{err}, c_i) = (\mathbb{C}^{err} \backslash \{c_i\}) \cup \{\overline{c_i}\}$$

Further on, the first modification of the `IPOG-NEG-`$\langle \forall, 0 \rangle$ variation can also be reused. But the second modification must be revised. The `IPOG-NEG-`$\langle \forall, b \rangle$ test selection strategy is depicted in Listing 8.4 (Page 150) and the modifications are again highlighted by borders.

In total, all b-wise interactions between $\mathtt{indices}(\overline{c_i})$ and all other parameters must be considered as computed by $\mathtt{AllInteractions}(c_i, b)$. Due to the step-wise extension, only the

```
 1   algorithm: IPOG-NEG-⟨∀, b⟩
 2   input: ℙ, 𝕍, ℂ, α, b
 3   output: A set of test inputs 𝕋⁻
 4   let φ_n ←  {r_1, ..., l_degree(c̄_i), ..., r_n}
 5   let φ_initial ←  indices(c̄_i)
 6   let X_{φ_initial} ← compute-schemata(φ_initial, 𝕍, ℂ)
 7   let 𝕋⁻ ← X_{φ_initial}
 8
 9   foreach j in  [degree(c̄_i)+1, n]
10      let φ_previous ← {r_1, ..., r_{j-1}}
11      let Φ_{degree(c̄_i)+b} ←  next-b-wise-interactions(φ_initial, φ_previous, r_j, b)
12      let X_{degree(c̄_i)+b} ← compute-all-schemata(Φ_{degree(c̄_i)+b}, 𝕍, ℂ)
13      𝕋⁻ ← horizontal-extension(𝕋⁻, X_{degree(c̄_i)+b})
14      if(uncovered-schemata-exist(𝕋⁻, X_{degree(c̄_i)+b}))
15         𝕋⁻ ← vertical-extension(𝕋⁻, X_{degree(c̄_i)+b})
16
17   𝕋⁻ ← replace-do-not-care-values(𝕋⁻)
18
19   return 𝕋⁻
```

Listing 8.4: $\texttt{IPOG-NEG-}\langle\forall, b\rangle$ Test Selection Strategy

interactions that involve the previous parameter indices $\phi_{previous}$ and the current parameter index r_j are of interest. Therefore, the $\texttt{next-b-wise-interactions}$ function cannot simply compute a subset of $\texttt{AllInteractions}(c_i, b)$.

For $b = 0$, the $\texttt{AllInteractions}$ function returns a singleton set including the empty set, i.e. $\{\emptyset\}$. Then, the interactions computed by $\texttt{next-b-wise-interactions}$ only contain the parameter indices of the error-constraint, i.e. $\{\phi_{initial}\}$. Since these interactions are already covered by the initial test suite, the result is equivalent to $\texttt{IPOG-NEG-}\langle\forall, 0\rangle$ and it becomes apparent that $\texttt{IPOG-NEG-}\langle\forall, b\rangle$ is a generalization of $\texttt{IPOG-NEG-}\langle\forall, 0\rangle$.

For $b > 0$, the interactions are computed as described by the $\texttt{AllInteractions}(c_i, b)$ function. The filtering may cause the first iterations to only cover interactions that are smaller than indicated by b. For instance, the first iteration only covers interactions between the parameter indices of $\overline{c_i}$ and r_j even for $b > 1$. Later iterations may contain duplicate interactions of the first iteration.

Running Example   For the RIPM of Figure 6.1 (Page 113), the $\texttt{IPOG-NEG-}\langle\forall, b\rangle$ the initial parameter index sets for error-constraint c_1 and $b = 2$ and the interactions that must be covered are as follows.

$$\phi_n = \{r_1 = 1, r_2 = 2, r_3 = 3, r_4 = 4$$
$$\phi_{initial} = \{r_1 = 1\}$$
$$\texttt{AllInteractions}(c_1, 2) = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}\}$$

> Since the RIPM encompasses four parameters, the initial test suite is extended three times.
>
> $$r_j = 2, \phi_{previous} = \{1\}, \Phi_{\texttt{degree}(\overline{c_i})+b} = \{\{1,2\}\} \qquad \text{for iteration j = 2}$$
> $$r_j = 3, \phi_{previous} = \{1,2\}, \Phi_{\texttt{degree}(\overline{c_i})+b} = \{\{1,2,3\}\} \qquad \text{for iteration j = 3}$$
> $$r_j = 4, \phi_{previous} = \{1,2,3\}, \Phi_{\texttt{degree}(\overline{c_i})+b} = \{\{1,2,4\},\{1,3,4\}\} \quad \text{for iteration j = 4}$$
>
> The interaction of the first iteration with $j = 2$ is only a subset of an actual interaction which is then subsumed by the second iteration with $j = 3$. This is due to the step-wise extension. At the end, all required interactions are covered. Since $\{1,2\}$ is not a subset of $\texttt{AllInteractions}(c_1, 2)$, a computation different than $\texttt{AllInteractions}$ is required.

To implement the `next-b-wise-interactions` function, the `Subsets` function is first applied to a set of parameter indices that contains all previous parameter indices $\phi_{previous}$, the current parameter index r_j, but excludes the parameter indices of the error-constraint $\phi_{initial}$, i.e. $(\phi_{previous} \cup \{r_j\}) \setminus \phi_{initial}$. The resulting set of b-sized subsets is element-wise united with the parameter indices of the error-constraint.

$$\texttt{next-b-wise-interactions}(\phi_{initial}, \phi_{previous}, r_j, b) =$$
$$= \{\phi_{initial} \cup \phi_b | \phi_b \in \texttt{Subsets}((\phi_{previous} \cup \{r_j\}) \setminus \phi_{initial}, b)\}$$

After computing the interactions, the schemata that must be covered as well as the horizontal and vertical extension remain the same.

Some Strong Invalid Schemata and No Robustness Interactions   To satisfy $\langle \exists, 0 \rangle$-wise strong invalid coverage for error-constraint c_i, the `IPOG-NEG-`$\langle \forall, 0 \rangle$ test selection strategy can be reused. But it is not necessary to include all specified strong invalid schemata. Instead, only one of the specified strong invalid schemata is sufficient. The algorithm is depicted in Listing 8.5 (Page 152).

In comparison to `IPOG-NEG-`$\langle \forall, 0 \rangle$, one additional modification to the test selection strategy is necessary to limit the initial test suite to exactly one specified strong invalid schemata. Therefore, the function `first` is introduced such that the result of `compute-schemata` is limited to exactly one schema. The final test suite only covers one strong invalid test input since the loop extends the test suite only horizontally with "do not care" values.

Some Strong Invalid Schemata and b-wise Robustness Interactions   To select test inputs such that $\langle \exists, b \rangle$-wise strong invalid coverage is satisfied for error-constraint c_i, the `IPOG-NEG-`$\langle \forall, b \rangle$ test selection strategy can be reused. But again, it is not necessary to include all specified strong invalid schemata.

Using the function `first` is not enough because the computation of b-wise interactions and the vertical extension may introduce additional test inputs and the resulting test suite could cover more than one of the specified strong invalid schemata. Alternatively, the constraints can be adjusted to ensure that only of the specified strong invalid schemata is used. In other words,

```
 1│  algorithm: IPOG-NEG-⟨∃, 0⟩
 2│  input: ℙ, 𝕍, ℂ, 𝑎, 𝑏
 3│  output: A set of test inputs 𝕋⁻
 4│  let ϕₙ ← {l₁, ..., l_degree(c̄ᵢ), ..., lₙ}
 5│  let ϕ_initial ← indices(c̄ᵢ)
 6│  let X_ϕ_initial ← first(compute-schemata(ϕ_initial, 𝕍, ℂ))
 7│  let 𝕋⁻ ← X_ϕ_initial
 8│
 9│  foreach j in [degree(c̄ᵢ)+1, n]
10│    let ϕ_previous ← {r₁, ..., r_{j−1}}
11│    let Φ₀ ← ∅
12│    let X₀ ← compute-all-schemata(Φ₀, 𝕍, ℂ)
13│    𝕋⁻ ← horizontal-extension(𝕋⁻, X₀)
14│    if(uncovered-schemata-exist(𝕋⁻, X₀))
15│      𝕋⁻ ← vertical-extension(𝕋⁻, X₀)
16│
17│  𝕋⁻ ← replace-do-not-care-values(𝕋⁻)
18│
19│  return 𝕋⁻
```

Listing 8.5: IPOG-NEG-⟨∃, 0⟩ Test Selection Strategy

the error-constraint is modified such that it only specifies one strong invalid schema. Then, this one strong invalid schema is used in all computations.

$$\texttt{modify-}\langle\exists, b\rangle(\mathbb{C}^{err}, c_i) = (\mathbb{C}^{err}\backslash\{c_i\}) \cup \{\overline{c'_i}\} \text{ with } \overline{c'_i} = \bigwedge_{\forall(p,v)\in s}(p = v) \text{ for one } s \in I_{c_i}$$

Running Example   To illustrate the alternative constraint modification, consider error-constraint c_4 and its $\overline{c_4}$ of the RIPM example (Figure 6.1, Page 113).

$c_4 : \neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+1}) \wedge \neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+49}) \wedge ...$

$\overline{c_4} : (\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+1}) \vee (\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+49}) \vee ...$

All specified strong invalid schemata $s \in I_{c_4}$ can be used to satisfy the negated error-constraint. To ensure that there is exactly one strong invalid schema that can be used to satisfy the negated error-constraint, the negated error-constraint can be re-modeled such that it is only satisfied by exactly one specified strong invalid schema.

For instance, when selecting [Country:UK, Phone:+1], the re-modeled and negated error-constraint $\overline{c'_4}$ is as follows.

$$\overline{c'_4} : (\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+1})$$

This modification allows reusing the `IPOG-NEG-`$\langle \forall, b \rangle$ test selection strategy which is depicted in Listing 8.4 (Page 150) without further modifications. Because of the modified constraint, only one strong invalid schemata is used as the initial test suite and only that one strong invalid schemata can appear in additional test inputs. Since `IPOG-NEG-`$\langle \forall, b \rangle$ is a generalization of `IPOG-NEG-`$\langle \forall, 0 \rangle$, this modification makes it a generalization of `IPOG-NEG-`$\langle \exists, 0 \rangle$ as well.

```
1   algorithm: IPOG-NEG-⟨a,0⟩
2   input: ℙ, 𝕍, ℂ, a, b̶
3   output: A set of test inputs 𝕋⁻
4   let φₙ ←  {r₁,...,rₐ,...,r_degree(c̄ᵢ),...,rₙ}
5   let φ_initial ←  {r₁,...,rₐ}
6   let X_φ_initial ← compute-schemata(φ_initial, 𝕍, ℂ)
7   let 𝕋⁻ ← X_φ_initial
8
9   foreach j in  [a+1, degree(c̄ᵢ)]
10      let φ_previous ← {r₁,...,r_{j-1}}
11      let Φₐ ←  next-interactions(φ_previous, rⱼ, a)
12      let Xₐ ← compute-all-schemata(Φₐ, 𝕍, ℂ)
13      𝕋⁻ ← horizontal-extension(𝕋⁻, Xₐ)
14      if(uncovered-schemata-exist(𝕋⁻, Xₐ))
15         𝕋⁻ ← vertical-extension(𝕋⁻, Xₐ)
16
17  foreach j in  [degree(c̄ᵢ)+1, n]
18      let φ_previous ← {r₁,...,r_{j-1}}
19      let Φ₀ ←  ∅
20      let X₀ ← compute-all-schemata(Φ₀, 𝕍, ℂ)
21      𝕋⁻ ← horizontal-extension(𝕋⁻, X₀)
22      if(uncovered-schemata-exist(𝕋⁻, X₀))
23         𝕋⁻ ← vertical-extension(𝕋⁻, X₀)
24
25  𝕋⁻ ← replace-do-not-care-values(𝕋⁻)
26
27  return 𝕋⁻
```

Listing 8.6: `IPOG-NEG-`$\langle a, 0 \rangle$ Test Selection Strategy

a-wise Strong Invalid Schemata and No Robustness Interactions For $\langle a, 0 \rangle$-wise strong invalid coverage, the `IPOG-NEG-`$\langle \forall, 0 \rangle$ test selection strategy can be adjusted. Instead of covering all specified strong invalid schemata $s \in I_{c_i}$, only all a-wise subsets are covered by modifying the computation of parameter indices for the initial test suite and for further extensions.

Therefore, the modification of error-constraints only encompasses the negation of c_i.

$$\texttt{modify-}\langle a, 0\rangle(\mathbb{C}^{err}, c_i) = (\mathbb{C}^{err}\backslash\{c_i\}) \cup \{\overline{c_i}\}$$

The $\texttt{IPOG-NEG-}\langle a, 0\rangle$ algorithm is depicted in Listing 8.6 (Page 153). The ordering of parameter indices ϕ_n again prioritizes the parameter indices of the error-constraint. But in contrast to the $\texttt{IPOG-NEG-}\langle\forall, 0\rangle$ test selection strategy, the initial test suite does not encompass the Cartesian product of all parameter indices of $\texttt{indices}(c_i)$. It only encompasses the Cartesian product of the first a parameter indices (Line 5). Although for $a = \texttt{degree}(\overline{c_i})$, the $\texttt{IPOG-NEG-}\langle\forall, 0\rangle$ and $\texttt{IPOG-NEG-}\langle a, 0\rangle$ test selection strategies are equal.

The extension of the initial test suite is further divided into two separate loops. First, the test suite is extended with all a-wise interactions for the remaining parameters of the error-constraint c_i. Therefore, the $\texttt{next-interactions}$ function can be reused that computes all a-sized interactions between $\phi_{previous}$ which – up to this point – only contains parameter indices of c_i and r_j which is another parameter index of c_i. Afterwards, the test suite is extended with values for all parameters. Since no further interactions must be covered, the vertical extension is skipped here.

a-wise Strong Invalid Schemata and b-wise Robustness Interactions   To select test inputs such that $\langle a, b\rangle$-wise strong invalid coverage is satisfied for error-constraint c_i, the mechanism for a-wise interactions as implemented by $\texttt{IPOG-NEG-}\langle a, 0\rangle$ must be combined with a mechanism for b-wise interactions. Although, the mechanism for b-wise interactions cannot be copied from $\texttt{IPOG-NEG-}\langle\forall, b\rangle$ and $\texttt{IPOG-NEG-}\langle\exists, b\rangle$ because they only work for $a = \texttt{degree}(\overline{c_i})$.

The $\texttt{IPOG-NEG-}\langle a, b\rangle$ test selection strategy extends the $\texttt{IPOG-NEG-}\langle a, 0\rangle$ test selection strategy. The same modification of error-constraints is used that only encompasses the negation of c_i.

$$\texttt{modify-}\langle a, b\rangle(\mathbb{C}^{err}, c_i) = (\mathbb{C}^{err}\backslash\{c_i\}) \cup \{\overline{c_i}\}$$

The algorithm is depicted in Listing 8.7 (Page 156). It utilizes the same ordering of ϕ_n as described before where the parameters of the error-constraint have the highest priority. The initial test suite consists of schemata for the first a parameters (Line 7). The initial test suite is extended with the remaining parameters of the error-constraint (Lines 9 to 15).

Afterwards, the test suite encompasses all parameters of the error-constraint similar to the $\texttt{IPOG-NEG-}\langle\forall, b\rangle$ and $\texttt{IPOG-NEG-}\langle\exists, b\rangle$ strategies. But in this case, the b-wise interactions are defined in relation to the a-sized schemata of error-constraint parameters as computed by $\texttt{Interactions}(c_i, a, b)$ instead of $\texttt{AllInteractions}(c_i, b)$ (See Page 139). Because of the step-wise extension, an alternative approach to the computation of $\texttt{next-a-b-wise-in-teractions}$ is required.

$\texttt{next-a-b-wise-interactions}(\phi_{initial}, \phi_{previous}, r_j, a, b)$

$\quad = \{\phi_a \cup \phi_b | \phi_a \in \texttt{Subsets}(\phi_{initial}, a) \wedge \phi_b \in \texttt{Subsets}((\phi_{previous} \cup \{r_j\})\backslash\phi_{initial}, b)\}$

Running Example For error-constraint c_4 of the RIPM example (Figure 6.1, Page 113), $\langle 1, 1 \rangle$-wise strong invalid coverage requires to cover schemata for the following parameter index sets.

$$\text{Interactions}(c_4, 1, 1) = \{\{2, 1\}, \{2, 4\}, \{3, 1\}, \{3, 4\}\}$$

When using the IPOG-NEG-$\langle a, b \rangle$ test selection strategy, the following parameter index sets are compute for the initial test suite.

$$\phi_n = \{r_1 = 2, r_2 = 3, r_3 = 1, r_4 = 4$$
$$\phi_{initial} = \{2\}$$

Afterwards, the initial test suite is extended by one more parameter of the error-constraint (j=2) an d all remaining parameters.

$r_j = 3, \phi_{previous} = \{2\}, \Phi_a = \{\{3\}\}$ for iteration j = 2
$r_j = 1, \phi_{previous} = \{2, 3\}, \Phi a + b = \{\{2, 1\}, \{3, 1\}\}$ for iteration j = 3
$r_j = 4, \phi_{previous} = \{2, 3, 1\}, \Phi a + b = \{\{2, 4\}, \{3, 4\}\}$ for iteration j = 4

As can be seen, all parameter index sets as computed by $\text{Interactions}(c_4, 1, 1)$ are covered by IPOG-NEG-$\langle a, b \rangle$.

The conceptual foundations (Chapter 3, Page 17 and following) establishes the distinction of valid and invalid schemata to detect ordinary and robustness failures. The discussion of the fault model (Chapter 5, Page 73 and following) further emphasizes the need to distinguish valid and invalid schemata during testing and presents hypothetical faults that are targeted by our CRT test method. Since the authoritative specification is not available during testing, we first introduce a RIPM structure that allows distinguishing valid and invalid schemata during testing (Chapter 6, Page 111 and following). Based on the RIPM structure, we discuss existing test adequacy criteria and develop a family of six new test adequacy criteria for strong invalid test inputs that correspond to the hypothetical faults of the fault model (Chapter 7, Page 119 and following).

In this chapter, test selection strategies are presented to select test inputs that satisfy the afore-mentioned test adequacy criteria. First, we discuss IPO-based test selection strategies and how they can be applied to select valid test inputs. Afterwards, we present a family of six test selection strategies that correspond to the six test adequacy criteria for strong invalid test inputs. With these test selection strategies, the CRT test method is complete and can be used to test SUTs.

An evaluation of the test selection strategies is conducted in the final part (Part IV, Page 211 and following). In the next part, supporting techniques are discussed (1) to ensure that the RIPMs are consistent and represent the authoritative specification correctly (Chapter 9, Page 159 and following), (2) to automate the construction of error-constraints (Chapter 10, Page 187 and following), and (3) to automate the CRT based on a test automation framework (Chapter 11, Page 199 and following).

```
 1│  algorithm: IPOG-NEG-⟨a, b⟩
 2│  input: ℙ, 𝕍, ℂ, a, b
 3│  output: A set of test inputs 𝕋⁻
 4│  let φₙ ←  {r₁, ..., rₐ, ..., r_degree(c̄ᵢ), ..., lₙ}
 5│  let φ_initial ←  {r₁, ..., rₐ}
 6│  let X_φ_initial ← compute-schemata(φ_initial, 𝕍, ℂ)
 7│  let 𝕋⁻ ← X_φ_initial
 8│
 9│  foreach j in  [a + 1, degree(c̄ᵢ)]
10│     let φ_previous ← {r₁, ..., r_{j−1}}
11│     let Φₐ ←  next-interactions(φ_previous, rⱼ, a)
12│     let Xₐ ← compute-all-schemata(Φₐ, 𝕍, ℂ)
13│     𝕋⁻ ← horizontal-extension(𝕋⁻, Xₐ)
14│     if(uncovered-schemata-exist(𝕋⁻, Xₐ))
15│       𝕋⁻ ← vertical-extension(𝕋⁻, Xₐ)
16│
17│  foreach j in  [degree(c̄ᵢ) + 1, n]
18│     let φ_previous ← {r₁, ..., r_{j−1}}
19│     let Φ_{a+b} ←  next-a-b-wise-interactions(φ_initial, φ_previous, rⱼ, a, b)
20│     let X_{a+b} ← compute-all-schemata(Φ_{a+b}, 𝕍, ℂ)
21│     𝕋⁻ ← horizontal-extension(𝕋⁻, X_{a+b})
22│     if(uncovered-schemata-exist(𝕋⁻, X_{a+b}))
23│       𝕋⁻ ← vertical-extension(𝕋⁻, X_{a+b})
24│
25│  𝕋⁻ ← replace-do-not-care-values(𝕋⁻)
26│
27│  return 𝕋⁻
```

Listing 8.7: IPOG-NEG-$\langle a, b \rangle$ Test Selection Strategy

Part III.

Supporting Techniques for Combinatorial Robustness Testing

Chapter 9.

Detection and Repair of Over-Constrained RIPMs

Contents

The previously discussed CRT test method allows defining a RIPM with error-constraints. Based on the partition into valid and strong invalid schemata that is imposed by the error-constraints, test selection strategies can be applied to select valid and strong invalid test inputs. But to ensure the right quality of selected test inputs, it is important that the partitioning of error-constraints is consistent with the partitioning of the authoritative specification.

In the following, we discuss techniques to detect and identify such inconsistencies. Thereby, we focus on a special case of over-constrained RIPMs where constraints are modeled too strict. As a result, some truly-strong invalid schemata are classified as not-strong invalid schemata and cannot appear in any strong invalid test input. The case of under-constrained RIPMs where constraints are modeled too relaxed and truly-strong invalid schemata are classified as valid schemata will be covered in the next chapter (Chapter 10, Page 187 and following).

In the first section, we extend the test process to include necessary activities to detect and repair over-constrained RIPMs. In addition, we define over-constrained RIPMs and discuss how to automatically identify incorrectly classified schemata and how to automatically explain the cause of the incorrect classification. Until now, repair is a fully-manual activity.

Therefore, we discuss a technique to automatically repair over-constrained RIPMs in the second section. The discussion unfortunately points out that a fully-automated approach is not meaningful and a semi-automated approach seems better.

But semi-automated repair can still be rejected because it may be perceived as complex and time-consuming. Therefore, we discuss an alternative technique in Section 3 that allows selecting invalid test inputs from over-constrained RIPMs without repairing them beforehand.

9.1. Detection and Manual Repair of Over-Constrained RIPMs

9.1.1. A Process for Detection and Manual Repair

The CRT test method as presented in Part II (Page 73 and following) works in general to test for ordinary failures and robustness failures. Depending on the test adequacy criterion, all or some modeled invalid schema should appear in strong invalid test inputs. However, it can be easy to create over-constrained RIPMs when applying it in practice. Especially when modeling invalid value combinations. In fact, real-world models are often over-constrained and when constraints are not correctly modeled, an invalid value or invalid value combination might not appear in strong invalid test inputs and robustness faults may remain undetected.

Here, we focus on a special case of over-constrained RIPMs. Therefore, we assume that the parameters and values are correctly modeled and the constraints are conflict-free for valid test input selection. But, when generating invalid test inputs, conflicts between constraints arise and prevent invalid values or invalid value combinations from appearing. Then, a tester must repair the RIPM by relaxing some constraints.

Running Example   To illustrate the issue of over-constrained RIPMs, we slightly modify error-constraint c_4 of the RIPM example for the ordering web-service (Figure 6.1, Page 113). The error-constraints are depicted below.

c_1 : Packages $\neq$ 123

c_2 : Country $\neq$ 123

c_3 : Phone $\neq$ 123

c_4 : $\neg$(Country = UK $\wedge$ Phone = +1) $\wedge$ $\neg$(Country = USA $\wedge$ Phone = +44)

First, we split the error-constraint c_4 into two error-constraints $c_4^{correct}$ and $c_5^{correct}$.

$$c_4^{correct} : \neg(\text{Country} = \text{UK} \wedge \text{Phone} = +1)$$
$$c_5^{correct} : \neg(\text{Country} = \text{USA} \wedge \text{Phone} = +44)$$

This transformation does not make the RIPM over-constrained. Its purpose is a better illustration of over-constrained RIPMs only. In fact, the partition of valid and strong invalid schemata is equivalent to the original unsplit error-constraint c_4. To create an over-constrained RIPM, consider slightly rewritten error-constraints c_4 and c_5 that could be the result of small mistakes made by the tester. The complete over-constrained example RIPM is depicted in Figure 9.1 (Page 161). The slightly rewritten error-constraints are depicted as follows.

$$c_4 : \text{Country} = \text{UK} \Rightarrow \text{Phone} = +44$$

```
Parameters:
  p₁ : Packages:           1, 3, 123
  p₂ : Country:            UK, USA, 123
  p₃ : Phone:              +44, +1, 123
  p₄ : Shipping:           Standard, Express
Error-Constraints:
  c₁ : Packages ≠ 123
  c₂ : Country ≠ 123
  c₃ : Phone ≠ 123
  c₄ : Country = UK ⇒ Phone = +44
  c₅ : Country = USA ⇒ Phone = +1
Exclusion-Constraints:
  c₆ : Shipping = Standard ⇒ Country ≠ USA
```

Figure 9.1.: Over-Constrained RIPM for the Ordering Web-Service Example

$$c_5 : \text{Country} = \text{USA} \Rightarrow \text{Phone} = \text{+1}$$

For the selection of valid test inputs, these slightly rewritten error-constraints are se-mantically equivalent to the original error-constraints. But the error-constraints are more restrictive for the selection of strong invalid test inputs. For instance, error-constraint c_3 originally defined the strong invalid schema $I_{c_3} = \{[\text{Phone:123}]\}$ that should appear in some strong invalid test inputs. Since the schema cannot appear in a strong invalid test input for the over-constrained RIPM, this is one example of a conflict caused by a over-constrained RIPM.

When considering the original RIPM as consistent with the authoritative specification, the partition of schemata into relevant, valid, and strong invalid schemata as imposed by the constraints is equivalent to the partition of truly-relevant, truly-valid, and truly-strong invalid schemata as imposed by the authoritative specification. To maintain consistency, it is important to repair the over-constrained RIPM.

In the following, it is further explained why the schema cannot appear in a strong invalid test input and how to repair the RIPM.

To further explain the absence of schemata, we use another view on error-constraints. There-fore, recall the initial discussion about error-constraints (Chapter 6, Page 111 and following). Error-constraints have a dual role. During the selection of valid test inputs, error-constraints represent conditions that must be satisfied by all valid test inputs. During the selection of strong invalid test inputs for error-constraint c_i, the error-constraint is negated ($\overline{c_i}$) to specify schemata that should appear in some strong invalid test inputs. A single negated error-constraint $\overline{c_i}$ can be transformed into a set of strong invalid schemata I_{c_i} and the selected test adequacy criteria determines whether a subset or all schemata of I_{c_i} must appear in strong invalid test inputs. Be-sides the strong invalid schemata I_{c_i}, a set of locally-specified schemata L_{c_i} can be derived. The locally-specified schemata L_{c_i} improve the comprehensibility of error-constraints because they

allow local reasoning of a single error-constraint independently of any other constraint.

$$I_{c_i} = \{s_j \in X_{\mathtt{Indices}(c_i)} | \mathtt{SAT}(C^{ex} \cup (C^{err}\backslash\{c_i\} \cup \{\overline{c_i}\}), s_j) = true\}$$
$$L_{c_i} = \{s_j \in X_{\mathtt{Indices}(c_i)} | \mathtt{SAT}(\{\overline{c_i}\}, s_j) = true\}$$

Please note, in order to distinguish different schemata, we label schemata consecutively with an integer $j \geq 1$. For the sake of simplicity, we assume that s_{j_1} and s_{j_2} refer to the same schema when $j_1 = j_2$ even across different sets for the same error-constraint c_i: I_{c_i}, L_{c_i}, and M_{c_i} which is introduced subsequently. In other words, $s_{j_1} \in I_{c_i}$ and $s_{j_2} \in L_{c_i}$ refer to the same schema when $j_1 = j_2$.

The locally-specified schemata are not necessarily relevant and strong invalid because a global scope that includes all constraints is required to derive strong invalid schemata. To maintain local reasoning and comprehensibility of error-constraints, we argue that whenever a schema s_j is locally-specified by an error-constraint c_i, i.e. $s_j \in L_{c_i}$, but that schema is not strong invalid, i.e. $s_j \in I_{c_i}$, the RIPM is inconsistent and needs to be repaired. By considering both sets of schemata, the comprehensibility of local reasoning is linked with the global scope of strong invalid schemata. Furthermore, the combination of both sets allows detecting and repairing inconsistent RIPMs.

To further investigate the issue, conflicts, over-constrained RIPMs, and missing invalid schema (MIS) are defined first.

Definition 98 (Conflict) *When selecting strong invalid test inputs for error-constraint c_i, a conflict is a contradiction between the negated error-constraint $\overline{c_i}$ and some other constraints $C^{ex} \cup (C^{err}\backslash\{c_i\})$. The interaction between negated error-constraint $\overline{c_i}$ and some other constraints explicitly or implicitly prevents a schema $s_j \in L_{c_i}$ from being covered by at least one strong invalid test input.*

Definition 99 (Missing Invalid Schema) *A locally-specified schema $s_j \in L_{c_i}$ for error-constraint c_i is a MIS if a conflict with some other constraints $C^{ex} \cup (C^{err}\backslash\{c_i\})$ prevents it from appearing in any strong invalid test input.*

$$s_j \in L_{c_i} \text{ is a MIS for error-constraint } c_i \text{ iff } \mathtt{SAT}(C^{ex} \cup (C^{err}\backslash\{c_i\} \cup \{\overline{c_i}\}), s_j) = false$$

Definition 100 (Over-Constrained RIPM) *A RIPM is over-constrained if and only if at least one conflict exists that prevents at least one locally-specified schema $s \in L_{c_i}$ of any error-constraint $c_i \in C^{err}$ from appearing in a strong invalid test input.*

$$\exists c_i \in C^{err}, \exists s_j \in L_{c_i}, \mathtt{SAT}(C^{ex} \cup (C^{err}\backslash\{c_i\} \cup \{\overline{c_i}\}), s_j) = false$$

Running Example The strong invalid schemata for error-constraints c_3, c_4, and c_5 of the over-constrained example 9.1 (Page 161) are depicted below.

$$I_{c_3} = \{\emptyset\}$$
$$I_{c_4} = \{s_1 = [\mathtt{Country:UK, \ Phone:+1}]\}$$

$$I_{c_5} = \{s_1 = \texttt{[Country:USA, Phone:+44]}\}$$

The locally-specified schemata are as follows.

$$L_{c_3} = \{s_1 = \texttt{[Phone:123]}\}$$
$$L_{c_4} = \{s_1 = \texttt{[Country:UK, Phone:+1]},$$
$$s_2 = \texttt{[Country:UK, Phone:123]}\}$$
$$L_{c_5} = \{s_1 = \texttt{[Country:USA, Phone:+44]},$$
$$s_2 = \texttt{[Country:USA, Phone:123]}\}$$

Although five schemata are locally-specified, only two of them ([Country:UK, Phone:+1] and [Country:USA, Phone:+44]) can be selected in strong invalid test inputs. The remaining three schemata [Phone:123], [Country:UK, Phone:123], and [Country:USA, Phone:123] are MISs.

The absence of the three MISs can be explained by two different types of issues that can be observed. Because of error-constraint c_3, some invalid test inputs should contain [Phone:123]. But due to error-constraints c_4 and c_5, [Phone:123] cannot be combined with [Country:UK] and [Country:USA]. Further on, error-constraint c_2 prevents the combination with [Country:123]. Therefore, error-constraint c_3 *implicitly* conflicts with c_2, c_4, and c_5 which needs to be repaired.

Two other contradictions are related to [Country:UK, Phone:123] and [Country:USA, Phone:123] as specified by error-constraints c_4 and c_5. Both schemata are forbidden by error-constraint c_3. Therefore, error-constraints c_4 and c_5 *explicitly* conflict with c_3 which needs to be repaired as well.

In order to remove conflicts, a tester must repair the RIPM by relaxing some constraints. Relaxation is often a labor-intensive and complicated task. Therefore, we propose a process to repair an over-constrained RIPM that is supported by an automatic identification of so-called MIS and by an automatic identification of suspect constraints (explanation). The process is depicted in Figure 9.2 (Page 164). The gray background indicates manual activities while the other activities can be automated.

After modeling the parameters with values and constraints, the RIPM is analyzed to identify MIS that cannot appear in strong invalid test inputs because of contradicting constraints. When no MISs are identified, the *normal* CT process continues with test input selection (cf. Figure 3.2, Page 29).

When MISs are identified, the tester needs an explanation, e.g. a list of constraints which are suspected to be incorrect, that helps to understand why an invalid schema is missing. Using the explanation, the tester can analyze and update the suspect constraints. The tester must decide to either remove the MIS from the test or to relax conflicting constraints. Afterwards, the RIPM is again analyzed and either remaining MISs are identified or test input are selected.

In the following, the activities of identifying MISs and providing explanations are described in more detail. Since both activities are labor-intensive and complicated, we discuss (1) the

Figure 9.2.: Extended CRT Test Process with Detection and Manual Repair Activities (Excerpt)

automatic identification of MIS and (2) the automatic explanation of contradictions between constraints that require relaxation.

9.1.2. Identifying Missing Invalid Schemata

To repair an over-constrained RIPM, all conflicts must be resolved by relaxing one or more constraints. Therefore, all MISs must be identified first. Then, the tester can decide for each MIS if it should be removed from the test or if the constraints that cause its absence should be relaxed.

Therefore, we introduce the set of all MISs for error-constraint c_i which is denoted as $M_{c_i} \subseteq L_{c_i}$. A locally-specified $s_j \in L_{c_i}$ is missing if and only if it is not a strong invalid schema as defined by error-constraint c_i, i.e. $s_j \notin I_{c_i}$. M_{c_i} can be computed by checking for each locally-specified schemata $s_j \in L_{c_i}$ if at least one strong invalid test input can be selected. If this assessment fails, the schema is a MIS.

$$
\begin{aligned}
M_{c_i} &= L_{c_i} \backslash I_{c_i} \\
&= \{s_j \in L_{c_i} | \texttt{SAT}(C^{ex} \cup (C^{err} \backslash \{c_i\} \cup \{\overline{c_i}\}), s_j) = false\}
\end{aligned}
$$

Please note again, for the sake of simplicity we let $s_{j_1} \in L_{c_i}, s_{j_2} \in I_{c_i}, s_{j_3} \in M_{c_i}$ refer to the same schema when $j_1 = j_2$ or $j_1 = j_3$. The indices $j_2 = j_3$ cannot refer to the same schema because I_{c_i} and M_{c_i} are mutually exclusive.

When selecting test inputs with a test selection strategy like `IPOG-C` (cf. [Yu+13a]) or `IPOG-NEG`, schemata of size t or a are created and extended until they consist of n parameter-value pairs for all n parameters.

```
Parameters:
  p₁ : Pa                          V₁ = {1, 2, 3}
  p₂ : C                           V₂ = {1, 2, 3}
  p₃ : Ph                          V₃ = {1, 2, 3}
  p₄ : S                           V₄ = {1, 2}
Error-Constraints:
  c₁ : Pa ≠ 3
  c₂ : C ≠ 3
  c₃ : Ph ≠ 3
  c₄ : C = 1 ⇒ Ph = 1
  c₅ : C = 2 ⇒ Ph = 2
Exclusion-Constraints:
  c₆ : S = 1 ⇒ C ≠ 2
```

Figure 9.3.: Internal Representation of Over-Constrained RIPM for the Ordering Web-Service Example

Every time a schema $s \in A$ is created or extended, the satisfiability function $\text{SAT} : C^* \times A \rightarrow Bool$ is involved to check if s can be extended to contain n parameter-value pairs while satisfying all constraints of a set C, i.e. $\text{SAT}(C, s) = true$. Any extension of s that does not satisfy all constraints is rejected. In order to check a schema s, the parameters, values, all constraints, and schema s are transformed into a CSP.

In order to better understand the computation of MISs, we discuss how the satisfiability function works when the RIPM and its constraints are transformed into a CSP and a constraint solver determines whether or not a schema satisfies the constraints.

A CSP consists of three components $\mathbb{X}$, $\mathbb{D}$ and $\mathbb{C}$ [RN10]. $\mathbb{X}$ is a set of variables, $\mathbb{D}$ is a set of domains with one domain for each variable and $\mathbb{C}$ is a set of constraints that restricts value combinations of variables. A solution for a CSP is an assignment of values to variables which is both consistent and complete. An assignment that does not violate any constraint is consistent. Otherwise, the assignment is inconsistent. An assignment is complete if every variable has a value assigned. Otherwise, it is partial. A constraint solver is applied to find a solution for the CSP. If a solution is found, the schema is accepted and can be further used during test input selection. If no solution exists, the schema cannot be used since one or more constraints cannot be satisfied.

To determine whether a schema $s \in A$ satisfies a set of constraints $C \in C^*$, the parameters and values of the RIPM, the set of constraints C and the schema under change s are transformed into a CSP [Yu+13a]. For the sake of clarity, Figure 9.3 (Page 165) depicts the internal representation of our over-constrained example (Figure 9.1, Page 161). Based on that internal representation, the translation into a CSP with $C = C^{ex} \cup C^{err}$ and $s = [\texttt{Packages:1, Country:UK}]$ is done as follows [Yu+13a].

Each parameter $p_i \in \tilde{P}$ of the RIPM is represented as a variable $x_i \in \mathbb{X}$. The domain of x_i represents the values V_i of parameter p_i as integers $\mathbb{D}_{x_i} = \{1, ..., m_i\}$. All constraints of the RIPM are translated to constraints of the CSP. The parameter $\texttt{Packages}$ is represented as the

variable `Pa` and its values are $D_{Pa} = \{1, 2, 3\}$. Variable `C` represents `Country`, variable `Ph` represents `Phone`, and `S` represents `Shipping`. Constraints are translated accordingly. For instance, `Packages` $\neq$ `123` becomes `Pa` $\neq 3$.

Furthermore, the values of the schema under change s are also added as constraints. We refer to them as schema-constraints. A schema $s = $ `[Packages:1, Country:UK]` translates to the constraint of the CSP $\{$`Pa` $= 1 \wedge$ `C` $= 1\}_s$. Given the CSP, the satisfiability function evaluates to $true$ if a CSP solver finds a solution. Otherwise, it evaluates to $false$. The complete CSP is depicted below.

$$\begin{aligned}
\mathbb{X} &= \{\texttt{Pa}, \texttt{C}, \texttt{Ph}, \texttt{S}\} \\
\mathbb{D} &= \{D_{Pa} = \{1, 2, 3\}, D_C = \{1, 2, 3\}, D_{Ph} = \{1, 2, 3\}, D_S = \{1, 2\}\} \\
\mathbb{C} &= \{\texttt{Pa} \neq 3, \texttt{C} \neq 3, \texttt{Ph} \neq 3, \texttt{C} = 1 \Rightarrow \texttt{Ph} = 1, \texttt{C} = 2 \Rightarrow \texttt{Ph} = 2, \\
&\quad\;\; \texttt{S} = 1 \Rightarrow \texttt{C} \neq 2\} \\
&\cup \{\texttt{Pa} = 1 \wedge \texttt{C} = 1\}_s
\end{aligned}$$

To identify all MISs for an error-constraint c_i with one or more conflicts, analogously created CSPs can be used prior to the selection of strong invalid test inputs. For each locally-specified schema $s_j \in L_{c_i}$, it is checked if at least one strong invalid test input exists that covers schema s_j and satisfies the negated error-constraint $\overline{c_i}$ as well as all other constraints. Therefore, a CSP as shown above is created and solved for each locally-specified schema $s_j \in L_{c_i}$. Every time, the current schema s_j is modeled as a schema-constraint $\{...\}_{s_j}$ and the specifying error-constraint c_i is negated, i.e. $\texttt{SAT}(C^{ex} \cup (C^{err} \backslash \{c_i\} \cup \{\overline{c_i}\}), s_j)$.

> **Running Example** When computing the MISs for the over-constrained example (Figure 9.1, Page 161), the following three schemata are identified as expected.
>
> $$\begin{aligned}
> M_{c3} &= \{s_1 = \texttt{[Phone:123]}\} \\
> M_{c4} &= \{s_2 = \texttt{[Country:UK, Phone:123]}\} \\
> M_{c5} &= \{s_2 = \texttt{[Country:USA, Phone:123]}\}
> \end{aligned}$$

> **List of Symbols**
>
> $M_{c_i} \subseteq L_{c_i}$ set of missing invalid schemata for error-constraint c_i

9.1.3. Explaining Conflicting Constraints

Knowing the MISs is a necessary first step to repair an over-constrained RIPM. Understanding the absence of a MIS is the next step. Then, the tester must either discard the MIS from testing by relaxing the error-constraint c_i whose negation error-constraint $\overline{c_i}$ is contradictory or the tester must relax some other constraints $C^{ex} \cup (C^{err} \backslash \{c_i\})$.

To further explain the absence of a MIS caused by a conflict, we introduce the notion of conflict sets.

Definition 101 (Conflict Set) *A conflict set $\mathcal{O}_{c_i,j} \subseteq C^{ex} \cup (C^{err} \backslash \{c_i\})$ is a set of constraints that explains the absence of a MIS $s_j \in M_{c_i}$. No strong invalid test input exists that covers s_j while satisfying the constraints of the conflict set.*

$$\forall s_j \in M_{c_i}, \exists \mathcal{O}_{c_i,j}, \mathtt{SAT}(\mathcal{O}_{c_i,j}, s_j) = \mathtt{false}$$

Technically, a conflict a for MIS $s_j \in M_{c_i}$ can be explained by the set of all other constraints $\mathcal{O}_{c_i,j} = C^{ex} \cup (C^{err} \backslash \{c_i\})$. But since repairing is a labor-intensive task, dealing with many constraints is often not useful because conflicts can become unclear and confusing. In addition, only a subset of constraints is responsible for the conflict and not every relaxation of constraints resolves the conflict. Rather than dealing with all constraints, it is more useful to identify and deal with a subset of constraints that explains the conflict more precisely. Therefore, we search for a minimal conflict set as an explanation that consists of as few constraints as possible.

Definition 102 (Minimal Conflict Set) *A conflict set $\mathcal{O}_{c_i,j}$ for a MIS $s_j \in M_{c_i}$ is a minimal conflict set for s_j if and only if there exists no proper subset $\mathcal{O}'_{c_i,j} \subset \mathcal{O}_{c_i,j}$ that is a conflict set for s_j.*

$\mathcal{O}_{c_i,j}$ is minimal for $s_j \in M_{c_i}$ iff $\nexists \mathcal{O}'_{c_i,j} \subset \mathcal{O}_{c_i,j}$ for which $\mathtt{SAT}(\mathcal{O}'_{c_i,j}, s) = \mathtt{false}$ holds

> **Running Example** For instance, consider the strong invalid test input selection for error-constraint c_4 of the over-constrained example (Figure 9.1, Page 161). To have an explanation for the one MIS [Country:UK, Phone:123], the minimal conflict set $\mathcal{O}_{c4,2} = \{c_3\}$ can be identified since it prohibits the selection of [Phone:123]. Without searching, the conflict set that contains all other constraints $\mathcal{O}_{c4,2} = \{c_1, c_2, c_3, c_5, c_6\}$ could function as an explanation.
>
> Because of the small example, the difference between the two conflict sets is small. However, the benefit of minimal conflict sets increases for real-world models with many more constraints.

Constraints and conflicts are interconnected, e.g. a constraint can belong to several conflict sets and a relaxation of one constraint can solve several conflicts. Therefore, the iterative process is reasonable where the tester focuses on one MIS and one minimal conflict set at a time. Afterwards, the updated model is again analyzed to identify further MIS that still exist.

To identify minimal conflict sets, we rely on *conflict detection techniques* for CSPs. Generally speaking, if there exists no solution for a CSP, some constraints must be relaxed to restore consistency [Jun04]. Therefore, constraints are partitioned into two disjoint groups: Constraints that can be relaxed (denoted as $\mathcal{C}$) and so-called *background* constraints that cannot be relaxed (denoted as $\mathcal{B}$) are distinguished. If no solution exists for both sets of constraints $\mathcal{C} \cup \mathcal{B}$, the model is over-constrained. A proper subset $\mathcal{R} \subset \mathcal{C}$ is a relaxation if and only if a solution exists for $\mathcal{R} \cup \mathcal{B}$. But no relaxation exists if $\mathcal{B}$ is already inconsistent without any relaxable constraints. A subset of relaxable constraints $\mathcal{C}$ denotes a conflict set $\mathcal{O}_{c_i,j} \subseteq \mathcal{C}$ if and only if no solution exists for $\mathcal{O}_{c_i,j} \cup \mathcal{B}$.

To automatically find a minimal conflict set, a conflict detection algorithm like QuickX-Plain [Jun04] can be used. Briefly explained, it is a divide and conquer approach that inspects constraints and determines whether they belong to the conflict or to the relaxation. To utilize QuickXPlain, it is necessary to identify the background constraints $\mathcal{B}$. Since the conflict set should explain why a schema $s_j \in M_{c_i}$ is missing, $\overline{c_i}$ and the schema-constraint $\{...\}_{s_j}$ should not be relaxed: $\mathcal{B} = \{\overline{c_i}\} \cup \{...\}_{s_j}$. In contrast, the remaining exclusion- and error-constraints are potentially too strict and should be relaxed: $\mathcal{C} = C^{ex} \cup (C^{err} \setminus \{c_i\})$.

> **Running Example**   For the over-constrained example (Figure 9.1, Page 161) with $s_2 = $ [Country:UK, Phone:123] missing for error-constraint c_4, the constraints are separated as shown below.
>
> $$M_{c_4} = \{s_2 = [\text{Country:UK, Phone:123}]\}$$
> $$\mathcal{B} = \{\overline{c_4}\} \cup \{T = 1 \wedge G = 3\}_{s_2}$$
> $$\mathcal{C} = \{c_6\} \cup \{c_1, c_2, c_3, c_5\}$$
>
> When applying QuickXPlain, the minimal conflict set $\mathcal{O}_{c_4,2} = \{c_3\}$ is computed.

Once MISs and minimal conflict sets are identified, the RIPM must be updated. Either the negated error-constraints $\overline{c_i}$ or the conflicting constraints $\mathcal{O}_{4,2}$ must be manually relaxed until no more schemata are missing. Afterwards, strong invalid test inputs can be selected such that they satisfy the chosen strong invalid coverage test adequacy criteria.

To summarize the contributions of this section, the issue of MISs is introduced and the necessity of repairing over-constraints RIPMs is discussed. To support the labor-intensive and complex task of repairing over-constrained RIPMs, necessary activities are identified and integrated into an extended CRT test process (Figure 9.2, Page 164). Although the task of repairing over-constrained RIPMs is perceived as a manual task until now, automated techniques are presented to identify MISs and to provide explanations in form of minimal sets of conflicting constraints.

In the next section, we focus on techniques that allow to repair over-constrained RIPMs semi-automatically to further reduce the manual effort.

List of Symbols

$\mathcal{O}_{c_i,j}$	conflict set for MIS $s_j \in L_{c_i}$
$\mathcal{C}$	set of relaxable constraints
$\mathcal{B}$	background constraints
$\mathcal{R} \subset \mathcal{C}$	relaxation

9.2. Semi-Automatic Repair of Over-Constrained RIPMs

In the previous section, we discussed techniques to explain over-constrained RIPMs by identifying MISs and conflicting constraints. Based on the explanations, a tester can manually repair the RIPMs and remove one or more conflicts by relaxing identified constraints.

Although the repair activities are already supported by automated identification and explanation, manual repair could still be rejected since it can be perceived as too time-consuming, too costly or too complex. Instead, further automation of repair activities would be desirable.

In this section, we extend the manual repair of over-constrained RIPMs and present a technique for automatic and semi-automatic repair. Even though the technique can be applied fully-automatically, we also provide an argument why the semi-automatic application is preferable.

Using the previously described concepts, minimal conflict sets explain the absence of the MISs. Due to the minimality property, only one constraint of a minimal conflict set must be relaxed to remove a conflict. However, it remains unclear (1) *which* constraint to select for relaxation and (2) *how* the selected constraint must be relaxed.

> **Running Example**   Until now, it is the tester's responsibility to select a constraint and to decide how to relax it. For a better illustration, we discuss the relaxation of the over-constrained example (Figure 9.1, Page 161) with $s = $ [Phone:123] that is missing for error-constraint c_3.
>
> Based on the application of a conflict detection algorithm, the following minimal conflict set can be computed.
>
> $$M_{c_3} = \{s_1 = \texttt{[Phone:123]}\}$$
> $$\mathcal{O}_{c_3,1} = \{c_2, c_4, c_5\}$$
>
> To remove the conflict, a tester must choose to either discard [Phone:123] of negated error-constraint $\overline{c_3}$ or to relax one of the error-constraints in $\mathcal{O}_{c_3,1}$.
>
> When choosing to relax c_4 or c_5, the tester must decide how the chosen constraint is relaxed. Either the entire constraint can be discarded or it can be loosened such that fewer schemata are locally-specified. For instance, either c_4 or c_5 could be replaced with the original $c_4^{correct}$ or $c_5^{correct}$ as depicted below.
>
> $$c_4^{correct} : \neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+1})$$
> $$c_5^{correct} : \neg(\texttt{Country} = \texttt{USA} \wedge \texttt{Phone} = \texttt{+44})$$
>
> This example also highlights the existence of several conflicts. A relaxation of c_4 to $c_4^{correct}$ removes the conflicts related to c_3 and c_4. But for error-constraint c_5, still one MIS exists.
>
> $$M_{c_5} = \{s_2 = \texttt{[Country:USA, Phone:123]}\}$$

Both aspects are discussed in the next subsections. Afterwards, we argue why semi-automatic repair is preferable over a fully-automatic repair.

Figure 9.4.: Example HS-tree

9.2.1. Automatic Diagnosis of Over-Constrained RIPMs

Conflict detection algorithms like `QuickXplain` [Jun04] allow to find a minimal conflict set $\mathcal{O}_{c_i,j}$ for a given MIS $s_j \in M_{c_i}$. While a minimal conflict set helps to identify subsets of constraints that explain a conflict, a tester must still manually decide which constraint of the subset to relax in order to solve the conflict. Since the absence of a MIS can be caused by more than one conflict, it may be necessary to relax more than one constraint.

To determine all constraints that must be relaxed, a so-called diagnosis set can be computed by a conflict diagnosis algorithm [Fel+14]: In the context of CSPs, a diagnosis set is a set of constraints such that a model is repaired if all constraints of the diagnosis set are relaxed. If $\mathcal{B}$ is consistent and $\mathcal{C} \cup \mathcal{B}$ has no solution, a diagnosis set $\triangle \subseteq \mathcal{C}$ is a set of constraints such that $\mathcal{B} \cup \mathcal{C} \backslash \triangle$ is consistent.

Definition 103 (Diagnosis Set) *For a MIS $s_j \in M_{c_i}$, a diagnosis set $\triangle_{c_i,j} \subseteq C^{ex} \cup (C^{err} \backslash \{c_i\})$ is a set of constraints such that all conflicts between negated error-constraint $\overline{c_i}$ and some other constraints can be removed by relaxing all constraints in $\triangle_{c_i,j}$. Formally, $\triangle_{c_i,j}$ is a diagnosis set if and only if $\mathtt{SAT}(C^{ex} \cup (C^{err} \backslash \{c_i\}) \backslash \triangle_{c_i,j}, s_j) = \mathtt{true}$.*

Technically, all constraints ($\triangle_{c_i,j} = C^{ex} \cup (C^{err} \backslash \{c_i\})$) form a diagnosis set. However, it is preferable to relax a smaller subset of constraints instead. Therefore, we introduce the notion of minimal diagnosis sets [Fel+14].

Definition 104 (Minimal Diagnosis Set) *A diagnosis set $\triangle_{c_i,j}$ for a MIS $s_j \in M_{c_i}$ is a minimal diagnosis set if and only if no proper subset $\triangle'_{c_i,j} \subset \triangle_{c_i,j}$ is a diagnosis set for s_j.*

Different algorithms to find minimal diagnosis sets exist [Fel+14]. A common approach is to compute a hitting set tree (HS-tree) from which all minimal diagnosis sets can be read off [Fel+14]; [Rei87]. The conflict diagnosis algorithm as introduced by REITER [Rei87] is based on the repeated application of a conflict detection algorithm [Fel+14]. Starting with an initial minimal conflict set, a breadth-first search is conducted by relaxing one constraint of the conflict set at a time. Then, the resulting set of constraints is again checked for consistency and either a solution or a new minimal conflict set is found. Minimal diagnosis sets can be created by following the path from a solution to the root conflict set.

Figure 9.5.: HS-trees for MISs of the Over-constrained Example

Running Example Figure 9.4 (Page 170) depicts an example HS-Tree adopted from [JSS15]. An initial minimal conflict set forms the tree's root node. The edges label the constraint which is relaxed. The node, to which the edge is pointing, is either another minimal conflict set, a diagnosis ($\sqrt{}$) or pruned ($\mathcal{X}$). Three minimal diagnosis sets can be created, i.e. $\triangle = \{c_1, c_4\}$, $\triangle = \{c_2\}$, and $\triangle = \{c_3, c_4\}$. The paths $\{c_1, c_2\}$ and $\{c_3, c_2\}$ are pruned because of $\triangle = \{c_2\}$.

It is important to note that the application of one diagnosis set only partially repairs the over-constrained model. For each missing invalid schema $s_j \in M_{c_i}$, either one of the computed minimal diagnosis sets $\triangle_{c_i,j}$ must be applied such that s_j is not missing anymore or error-constraint c_i itself must be relaxed ($\triangle_{c_i,j} = \{c_i\}$) such that s_j is discarded and not locally-specified anymore.

Therefore, let $\triangle_{c_i,j}^{all}$ denote the set of all minimal diagnosis sets $\triangle_{c_i,j} \in \triangle_{c_i,j}^{all}$ for a particular MIS $s_j \in M_{c_i}$ which also includes $\triangle_{c_i,j} = \{c_i\}$ as another minimal diagnosis set that relaxes error-constraint c_i to discard s_j.

Running Example Figure 9.5 shows the HS-trees for the three MISs of the over-constrained example (Figure 9.1, Page 161). The root nodes are annotated with $\mathcal{O}_{c_i,j}$ to indicate the corresponding MISs.

Five different minimal diagnosis sets can be computed. They are depicted below including $\triangle_{c_i,j} = \{c_i\}$.

$$\triangle_{c_3,1}^{all} = \{\triangle_{c_3,1}^1 = \{c_2\}, \triangle_{c_3,1}^2 = \{c_4\}, \triangle_{c_3,1}^3 = \{c_5\}\} \cup \{\triangle_{c_3,1}^4 = \{c_3\}\}$$
$$\triangle_{c_4,2}^{all} = \{\triangle_{c_4,2}^1 = \{c_3\}\} \cup \{\triangle_{c_4,2}^2 = \{c_4\}\}$$
$$\triangle_{c_5,2}^{all} = \{\triangle_{c_5,2}^1 = \{c_3\}\} \cup \{\triangle_{c_5,2}^2 = \{c_5'\}\}$$

Applying $\triangle_{c_3,1}^1 = \{c_2\}$ only removes the conflict for $s_1 \in M_{c_3}$ such that [Country:123, Phone:123] can be selected as a strong invalid schema for error-constraint $\overline{c_3}$. But conflicts for error-constraints $\overline{c_4}$ and $\overline{c_5}$ still exist. They require additional relaxation since there are still MISs of M_4 and M_5.

In contrast, relaxing c_3 as proposed by $\triangle_{c_4,2}^1$ and $\triangle_{c_5,2}^1$ resolves two conflicts for error-constraints c_4 and c_5. Further on, $s_1 \in M_{c_3}$ is discarded and the conflict for error-constraints c_3 is also removed.

Therefore, a set of constraints is required such that their relaxation removes *all* conflicts among *all* constraints.

Definition 105 (Diagnosis Hitting Set) *A diagnosis hitting set $\triangle_{HS} \subseteq C^{ex} \cup C^{err}$ is a set of constraints such that the relaxation of all constraints in $\triangle_{HS}$ repairs the complete RIPM.*

Formally, let $RIPM = \langle \tilde{P}, V, C^{ex}, C^{err} \rangle$ denote an over-constrained RIPM for which $\triangle_{HS}$ is a diagnosis hitting set if and only if $RIPM' = \langle \tilde{P}, V, C^{ex} \backslash \triangle_{HS}, C^{err} \backslash \triangle_{HS} \rangle$ is not over-constrained anymore, i.e. no MISs exist $\forall c_i \in C^{err}$ of $RIPM'$, $M_{c_i} = \emptyset$.

A diagnosis hitting set $\triangle_{HS}$ can be composed of minimal diagnosis sets if $\triangle_{HS}$ subsumes at least one minimal diagnosis for each MIS. Therefore, let $\triangle^{all} = \triangle^{all}_{c_{i_1}, j_1} \times \triangle^{all}_{c_{i_2}, j_2} \times \ldots$ denote the Cartesian product of all sets of all minimal diagnosis sets $\triangle^{all}_{c_i, j}$ for all MISs $s_j \in M_{c_i}$. Then, each tuple $e \in \triangle^{all}$ contains one minimal diagnosis set for each MIS and can be used to repair the complete RIPM. A tuple $e \in \triangle^{all}$ can be further transformed into a diagnosis hitting set $\triangle_{HS}$ by unifying all contained minimal diagnosis sets.

$$\triangle_{HS}(e) = \bigcup_{\triangle_{c_i,j} \in e} \triangle_{c_i,j}$$

Running Example   For the over-constrained example (Figure 9.1, Page 161), minimal diagnosis sets for three MISs must be subsumed.

$$M_{c3} = \{s_1 = \texttt{[Phone:123]}\}$$
$$M_{c4} = \{s_2 = \texttt{[Country:UK, Phone:123]}\}$$
$$M_{c5} = \{s_2 = \texttt{[Country:USA, Phone:123]}\}$$

The sets of all minimal diagnosis sets for the three MISs are again shown below.

$$\triangle^{all}_{c3,1} = \{\triangle^1_{c3,1} = \{c_2\}, \triangle^2_{c3,1} = \{c_4\}, \triangle^3_{c3,1} = \{c_5\}\} \cup \{\triangle^4_{c3,1} = \{c_3\}\}$$
$$\triangle^{all}_{c4,2} = \{\triangle^1_{c4,2} = \{c_3\}\} \cup \{\triangle^2_{c4,2} = \{c_4\}\}$$
$$\triangle^{all}_{c5,2} = \{\triangle^1_{c5,2} = \{c_3\}\} \cup \{\triangle^2_{c5,2} = \{c_5\}\}$$

$\triangle^{all}$ is the Cartesian product $\triangle^{all} = \triangle^{all}_{c3,1} \times \triangle^{all}_{c4,2} \times \triangle^{all}_{c5,2}$ that contains, for instance, the following two selections e_1 and e_2.

$$e_1 = \langle \triangle^1_{c3,1} = \{c_2\}, \triangle^1_{c4,2} = \{c_3\}, \triangle^1_{c5,2} = \{c_3\}\rangle$$
$$e_2 = \langle \triangle^1_{c3,1} = \{c_2\}, \triangle^1_{c4,2} = \{c_3\}, \triangle^2_{c5,2} = \{c_5\}\rangle$$

Both selections can be transformed into diagnosis hitting sets:

$$\triangle_{HS}(e_1) = \{c_2, c_3\}$$
$$\triangle_{HS}(e_2) = \{c_2, c_3, c_5\}$$

The first three columns of Table 9.6 (Page 173) depict all possible selections and the fourth column depicts all diagnosis hitting sets.

$\triangle_{c_3,1}$	$\triangle_{c_4,2}$	$\triangle_{c_5,2}$	$\triangle_{HS}$
$\{c_2\}$	$\{c_3\}$	$\{c_3\}$	$\{c_2, c_3\}$
$\{c_2\}$	$\{c_3\}$	$\{c_5\}$	$\{c_2, c_3, c_5\}$
$\{c_2\}$	$\{c_4\}$	$\{c_3\}$	$\{c_2, c_3, c_4\}$
$\{c_2\}$	$\{c_4\}$	$\{c_5\}$	$\{c_2, c_4, c_5\}$
$\{c_4\}$	$\{c_3\}$	$\{c_3\}$	$\{c_3, c_4\}$
$\{c_4\}$	$\{c_3\}$	$\{c_5\}$	$\{c_3, c_4, c_5\}$
$\{c_4\}$	$\{c_4\}$	$\{c_3\}$	$\{c_3, c_4\}$
$\{c_4\}$	$\{c_4\}$	$\{c_5\}$	$\{c_4, c_5\}$
$\{c_5\}$	$\{c_3\}$	$\{c_3\}$	$\{c_3, c_5\}$
$\{c_5\}$	$\{c_3\}$	$\{c_5\}$	$\{c_3, c_5\}$
$\{c_5\}$	$\{c_4\}$	$\{c_3\}$	$\{c_3, c_4, c_5\}$
$\{c_5\}$	$\{c_4\}$	$\{c_5\}$	$\{c_4, c_5\}$
$\{c_3\}$	$\{c_3\}$	$\{c_3\}$	$\{c_3\}$
$\{c_3\}$	$\{c_3\}$	$\{c_5\}$	$\{c_3, c_5\}$
$\{c_3\}$	$\{c_4\}$	$\{c_3\}$	$\{c_3, c_4\}$
$\{c_3\}$	$\{c_4\}$	$\{c_5\}$	$\{c_3, c_4, c_5\}$

Table 9.6.: Diagnosis Hitting Sets for the Over-Constrained Example

The application of each diagnosis hitting set results in a repaired model that is not over-constrained anymore. But to avoid unnecessary changes to the model, only minimal diagnosis hitting sets are further considered.

Definition 106 (Minimal Diagnosis Hitting Set) *A diagnosis hitting set $\triangle_{HS}$ is a minimal diagnosis hitting set if and only if no proper subset $\triangle'_{HS} \subset \triangle_{HS}$ is a diagnosis hitting set.*

Definition 107 (Cardinality-minimal Diagnosis Hitting Set) *A diagnosis hitting set $\triangle_{HS}$ is a cardinality-minimal diagnosis hitting set if and only if there exists no diagnosis hitting set $\triangle'_{HS}$ that contains fewer constraints.*

$$\triangle_{HS} \text{ is cardinality-minimal iff } \nexists \triangle'_{HS} \text{with } |\triangle'_{HS}| < |\triangle_{HS}|$$

Running Example For the over-constrained example (Figure 9.1, Page 161), the diagnosis hitting sets are depicted in Table 9.6 (Page 173). The minimal diagnosis hitting sets for the example are $\triangle^1_{HS} = \{c_4, c_5\}$ and $\triangle^2_{HS} = \{c_3\}$.

Furthermore, $\triangle_{HS} = \{c_3\}$ is a cardinality-minimal diagnosis hitting set.

List of Symbols

$\triangle_{c_i,j}$	diagnosis set for MIS $s_j \in M_{c_i}$
$\triangle^{all}_{c_i,j}$	set of all minimal diagnosis sets for MIS $s_j \in M_{c_i}$

$\triangle^{all}$	Cartesian product of all sets of all minimal diagnosis sets
$e \in \triangle^{all}$	tuple with one minimal diagnosis set for each MIS
$\triangle_{HS}$	diagnosis hitting set

9.2.2. Automatic Relaxation of Conflicting Constraints

Minimal diagnosis hitting sets answer the question *which* constraints to relax but not *how* to relax them. When a minimal diagnosis hitting set references a constraint c_i that encompasses several schemata $s_j \in L_{c_i}$, it remains unclear whether the relaxation should remove all or only a subset of schemata. Therefore, we discuss appropriate changes to the diagnostic in order to support automatic relaxation.

Running Example Error-constraint c_4 of the over-constrained example (Figure 9.1, Page 161) specifies two schemata:

$$L_{c_4} = \{s_1 = [\text{Country:UK, Phone:+1}],$$
$$s_2 = [\text{Country:UK, Phone:123}]\}$$

When c_4 is selected for relaxation – for instance to resolve the conflict for [Phone:123] of M_{c_3} – it is unclear whether both or only one of the two schemata should be removed.

To provide more information about the specific schemata $s_j \in L_{c_i}$ that must be relaxed, a finer-grained schema-level representation of diagnosis hitting sets is necessary. Therefore, the transformation into CSP is adjusted. Instead of translating constraints of the RIPM directly into constraints of the CSP, a schema-based CSP can be used where a separate constraint is created for each schema that is either irrelevant or invalid.

Therefore, let the concept of locally-specified schemata $s_j \in L_{c_i}$ be extended from error-constraints to exclusion-constraints. For an exclusion-constraint $c_i \in C^{ex}$, the computation of L_{c_i} is equivalent to the computation for error-constraints. But the computed schemata are irrelevant and are not supposed to appear in any valid or strong invalid test input.

Based on the locally-specified schemata L_{c_i} of each constraint $c_i \in C^{ex} \cup C^{err}$, a separate error- or exclusion-constraint $c_{i,j}$ is created for each irrelevant or invalid schema $s_j \in L_{c_i}$ where j in $c_{i,j}$ indicates the j-th schema $s_j \in L_{c_i}$. Each parameter-value pair $\pi = (p, v)$ of an irrelevant or invalid schema s_j is translated into a proposition $p = v$, all propositions are combined via logical conjunction, and the conjunction is negated such that irrelevant and invalid schemata are avoided.

$$c_{i,j} = \neg\Big(\bigwedge_{(p,v) \in s_j} (p = v) \Big)$$

Running Example Error-constraint c_4 of the over-constrained example (Figure 9.1, Page 161) is modeled as an implication $\texttt{Country} = \texttt{UK} \Rightarrow \texttt{Phone} = \texttt{+44}$ that specifies the following two schemata.

$$L_{c_4} = \{s_1 = \texttt{[Country:UK, Phone:+1]},$$
$$s_2 = \texttt{[Country:UK, Phone:123]}\}$$

When transforming the RIPM into a schema-based CSP, the error-constraint is transformed into two error-constraints $c_{4,1}$ and $c_{4,2}$.

$$c_{4,1} : \neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+1})$$
$$c_{4,2} : \neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{123})$$

Conflict detection and diagnosis with schema-based CSPs provide more details for relaxation as they compute minimal conflict sets, minimal diagnosis sets, and minimal diagnosis hitting sets that reference some schema-level constraints $c_{i,j}$. Since each schema-level constraint $c_{i,j}$ refers to exactly one schema, the question whether the entire constraint c_i or only a subset must be relaxed is unambiguously resolved.

When a schema-level constraint $c_{i,j}$ must be relaxed so that one or more schemata are not missing anymore, the original constraint c_i must be rewritten such that it excludes the $s_j \in L_{c_i}$. The constraint can be either manually or automatically rewritten to exclude s_j by adding a logical disjunction for each parameter-value pair of s_j to c_i.

$$\texttt{Exclude}(c_i, s_j) = c_i \vee \Big(\bigwedge_{(p,v)\in s_j} (p = v) \Big)$$

Running Example To conclude the section, we walk through the transformations and computations with schema-level constraints for the over-constrained example (Figure 9.1, Page 161). First, the constraints are transformed into schema-level constraints as follows:

$$c_{1,1} : \neg(\texttt{Packages} = \texttt{123})$$
$$c_{2,1} : \neg(\texttt{Country} = \texttt{123})$$
$$c_{3,1} : \neg(\texttt{Phone} = \texttt{123})$$
$$c_{4,1} : \neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+1})$$
$$c_{4,2} : \neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{123})$$
$$c_{5,1} : \neg(\texttt{Country} = \texttt{USA} \wedge \texttt{Phone} = \texttt{+44})$$
$$c_{5,2} : \neg(\texttt{Country} = \texttt{USA} \wedge \texttt{Phone} = \texttt{123})$$
$$c_{6,1} : \neg(\texttt{Shipping} = \texttt{Standard} \wedge \texttt{Country} = \texttt{USA})$$

The same schemata are still MISs but the assignment to schema-level constraints

makes it more precise.

$$M_{c_{3,1}} = \{s_1 = \texttt{[Phone:123]}\}$$
$$M_{c_{4,2}} = \{s_1 = \texttt{[Country:UK, Phone:123]}\}$$
$$M_{c_{5,2}} = \{s_1 = \texttt{[Country:USA, Phone:123]}\}$$

Based on the MISs, three initial minimal conflict sets can be computed. They can also be more precisely assigned to schema-level constraints.

$$\mathcal{O}_{c_{3,1},1} = \{c_{2,1}, c_{4,2}, c_{5,2}\}$$
$$\mathcal{O}_{c_{4,2},1} = \{c_{3,1}\}$$
$$\mathcal{O}_{c_{5,2},1} = \{c_{3,1}\}$$

With these minimal conflict sets, the following two minimal diagnosis hitting sets can be computed.

$$\triangle^1_{HS} = \{c_{4,2}, c_{5,2}\}$$
$$\triangle^2_{HS} = \{c_{3,1}\}$$

Since the schema-level constraints are assigned to single schemata, the required relaxation is unambiguous. When choosing to apply $\triangle^1_{HS}$, the error-constraints c_4 and c_5 must be relaxed such that they exclude the schema-level constraints $c_{4,2}$ and $c_{5,2}$. Using the technique based on logical disjunction as described above, the error-constraints can be rewritten as follows.

$$c'_4 : (\texttt{Country} = \texttt{UK} \Rightarrow \texttt{Phone} = \texttt{+44})$$
$$\qquad \vee (\texttt{Country} = \texttt{USA} \wedge \texttt{Phone} = \texttt{123})$$
$$c'_5 : (\texttt{Country} = \texttt{USA} \Rightarrow \texttt{Phone} = \texttt{+1})$$
$$\qquad \vee (\texttt{Country} = \texttt{USA} \wedge \texttt{Phone} = \texttt{123})$$

These rewritten error-constraints are again semantically equivalent to the correct but split error-constraints $c_4^{correct}$ and $c_5^{correct}$ (See running example on Page 160).

$$c_4^{correct} : \neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = \texttt{+1})$$
$$c_5^{correct} : \neg(\texttt{Country} = \texttt{USA} \wedge \texttt{Phone} = \texttt{+44})$$

List of Symbols

$c_{i,j}$ schema-level constraint for $s_j \in L_{c_i}$

$\texttt{Exclude}(c_i, s_j)$ function to exclude schema $s_j \in L_{c_i}$ from constraint c_i

9.2.3. Selection and Application of Diagnosis Hitting Sets

Combining the previously discussed techniques allows completely automated repairs of over-constrained RIPMs: Schema-based CSP provide all information to automatically relax constraints and diagnosis hitting sets describe sets of constraints that must be relaxed. Focusing on minimal diagnosis hitting sets reduces the number of options, but still one out of several minimal diagnosis hitting sets must be selected.

While choosing the cardinality-minimal diagnosis hitting set, i.e. the diagnosis hitting set that contains the fewest constraints, seems appealing, it is not necessarily the best choice. Unfortunately, not every repaired RIPM is *equivalent* to the correct RIPM in the sense that it represents the partitioning of the authoritative specification and that selected test inputs covers all expected schemata.

In order to repair an over-constrained RIPM such that its partitioning is consistent with the authoritative specification, an appropriate minimal diagnosis hitting set must be selected. Since there is no further information that can be utilized by a fully-automated approach, we argue that semi-automation is preferable. During the evaluation, this phenomenon is further investigated.

Running Example In order to illustrate the remaining problem of selecting one out of several minimal diagnosis hitting sets, we apply the cardinality-minimal diagnosis hitting set $\triangle^2_{HS} = \{c_{3,1}\}$ to repair the over-constrained example (Figure 9.1, Page 161) and to contrast it with the previous repair of applying $\triangle^1_{HS} = \{c_{4,2}, c_{5,2}\}$.

Below the error-constraint c_3, the schema-level constraint $c_{3,1}$, and the rewritten error-constraint c'_3 are shown.

$$c_3 : \texttt{Phone} \neq \texttt{123}$$
$$c_{3,1} : \neg(\texttt{Phone} = \texttt{123})$$
$$c'_3 : (\texttt{Phone} \neq \texttt{123}) \vee (\texttt{Phone} = \texttt{123})$$

After the repair, the RIPM is not over-constrained anymore and no schemata are MISs. However, the repair makes error-constraint c'_3 ineffective since it is always satisfied. As a consequence, no dedicated strong invalid test input for [Phone:123] must be selected.

Instead, two strong invalid test inputs must be selected for [Country:UK, Phone:123] and [Country:USA, Phone:123] of error-constraints c_4 and c_5. Since [Phone:123] is not marked as invalid, schemata like [Packages:123, Phone:123] or [Country:123, Phone:123] are incorrectly considered strong invalid schemata.

For this example, it is desirable to choose $\triangle^1_{HS} = \{c_{4,2}, c_{5,2}\}$ over $\triangle^2_{HS} = \{c_{3,1}\}$. However, since no further information is available, the selection cannot be automated.

9.3. Strong Invalid Test Input Selection from Over-Constrained RIPMs

While the application of conflict diagnosis techniques as presented in the previous section allows to increase the level of automation in the repair process for over-constrained RIPMs, the selection of appropriate minimal diagnosis hitting sets demonstrates its limit for automation. In practice, the manual and semi-automatic repair could be rejected because it could be perceived as too time-consuming, too costly or too complex.

Since a fully-automated approach is still desirable, we discuss a different approach in this section to select invalid test inputs directly from over-constrained RIPMs without repairing them beforehand. Therefore, the idea of alternative constraint handling strategies is introduced and different constraint handling strategies are presented to automatically deal with conflicts during the test input selection.

In the following subsections, we first discuss the general idea of soft constraint handling strategies. Afterwards, we discuss different soft constraint handling strategies that are based on that idea.

9.3.1. General Idea of Soft Constraint Handling Strategies

As stated before, when generating test inputs, schemata of size t or a are created and extended until they consist of n parameter-value pairs for all n parameters. Every time a schema s is created or extended, constraint handling is used to check if possible extensions of $s' \supset s$ satisfy all constraints. Any extension that does not satisfy all constraints is rejected. The satisfiability function $\texttt{SAT} : C^* \times A \rightarrow Bool$ is responsible to accept or reject a schema $s \in A$.

In order to realize the evaluation function, the parameters and values of the RIPM, all constraints $C \in C^*$, and the schema under change $s \in A$ can be transformed into a CSP and a constraint solver can be applied to search for a solution. If a solution is found, the schema is accepted. Otherwise, the schema is rejected.

Different approaches for the transformation into CSPs exist which we denote as **constraint handling strategies**. In Section 9.1.2 (Page 164 and following), a first transformation into CSPs is presented. It is the hard constraint handling (HCH) strategy which is the *default* constraint handling strategy that is used in test selection strategies like $\texttt{IPOG-C}$ [Yu+13a] and $\texttt{IPOG-NEG}$.

The HCH strategy is already discussed in Section 9.1.2 (Page 166). However, it is briefly repeated here. Each parameter $p_i \in P$ of the RIPM is represented as a variable $x_i \in \mathbb{X}$. The domain of x_i represents the values V_i of parameter p_i as integers $\mathbb{D}_{x_i} = \{1, ..., m_i\}$. All constraints of the RIPM are directly translated to constraints of the CSP. Furthermore, the schema under change s is also added as a so-called schema-constraints.

> **Running Example**   The resulting CSP for a given schema under change $s = $ `[Packages:1, Country:UK]` and the internal representation of the over-constrained example (Figure 9.3, Page 165) is depicted below.
>
> $\mathbb{X} = \{\texttt{Pa}, \texttt{C}, \texttt{Ph}, \texttt{S}\}$

$$\mathbb{D} = \{D_{Pa} = \{1, 2, 3\}, D_C = \{1, 2, 3\}, D_{Ph} = \{1, 2, 3\}, D_S = \{1, 2\}\}$$

$$\mathbb{C} = \{\texttt{Pa} \neq 3, \texttt{C} \neq 3, \texttt{Ph} \neq 3, \texttt{C} = 1 \Rightarrow \texttt{Ph} = 1, \texttt{C} = 2 \Rightarrow \texttt{Ph} = 2,$$

$$\texttt{S} = 1 \Rightarrow \texttt{C} \neq 2\}$$

$$\cup \{\texttt{Pa} = 1 \wedge \texttt{C} = 1\}_s$$

When using the HCH strategy with an over-constrained RIPM, the test adequacy criteria for strong invalid test inputs cannot be satisfied because some specified schemata are MISs and the HCH strategy excludes them. To cover all specified schemata, the HCH strategy requires the tester to repair the RIPM and to remove all conflicts.

Therefore, we introduce soft constraint handling (SCH) strategies that does not require any repair prior to the test input selection. The idea of SCH is based on *partial* constraint satisfaction as described by FREUDER & WALLACE [FW92]. Instead of requiring all constraints to be satisfied by a solution, a partial solution that satisfies as many constraints as possible is accepted as well.

For SCH, hard-constraints and soft-constraints are distinguished. A solution must satisfy all hard-constraints (denoted as $\mathcal{H}$) but not all soft-constraints (denoted as $\mathcal{S}$). Although, it is desirable to satisfy as many soft-constraints as possible.

To take into account that the satisfaction of soft-constraints is optional, each soft-constraint $c_i \in \mathcal{S}$ is *reified* when transforming it into a CSP. Therefore, an additional boolean variable $r_i \in \mathbb{X}$ is created for each soft-constraint c_i and c_i is modified to capture whether or not it is satisfied. The variable r_i is $true$ if the constraint c_i is satisfied by an assigment and $false$ otherwise, i.e. $r_i \Leftrightarrow c_i$.

The CSP is transformed into an optimization problem and a utility function $\mathcal{U}(\alpha)$ counts all reified soft-constraints r_i that are satisfied by assignment α. Then, the constraint solver searches for an assignment α that satisfies all hard-constraints and calculates the utility value.

$$\mathcal{U}(\alpha) = \sum_{c_i \in \mathcal{S}} 1 \times \alpha.r_i$$

When checking whether or not to reject a schema, a binary answer $true$ or $false$ is required. However, an optimization problem always has a solution. Even zero satisfied soft-constraints with a utility value of zero may represent a consistent assignment. Therefore, a threshold η with $0 \leq \eta \leq |\mathcal{S}|$ is introduced to decide whether or not α is acceptable. The threshold is a lower boundary for the utility function $\eta \leq \mathcal{U}(\alpha)$. An assignment is only consistent if at most η soft-constraints remain unsatisfied. In other words, a solution must satisfy at least $|\mathcal{S}| - \eta$ soft-constraints.

In the following, two SCH strategies are discussed that are based on the idea of soft constraint handling.

List of Symbols

$\mathcal{H}$	set of hard-constraints
$\mathcal{S}$	set of soft-constraints

α	assignment of a CSP
$\mathcal{U}(\alpha)$	function to compute the utility of soft-constraints
η	threshold for the utility of soft-constraints

9.3.2. Basic Soft Constraint Handling Strategy

Following the idea of SCH, the transformation for the basic soft constraint handling (B-SCH) strategy is realized as follows. The transformation of parameters, values and constraints that are considered as hard-constraints is analogous to the HCH strategy.

Each input parameter p_i is represented as a variable $x_i \in \mathbb{X}$ and x_i represents its domain as integers $D_{x_i} = \{1, ..., m_i\} \in \mathbb{D}$.

Hard- and soft-constraints are distinguished as follows. The set of hard-constraints $\mathcal{H}$ is equivalent to the set of background constraints $\mathcal{B}$. In both cases, the sets encompass the schema-constraint and negated error-constraints $\overline{c_i}$. The remaining constraints are modeled as soft-constraints which affect the utility value.

$$\mathcal{H} = \{\overline{c_i}\} \cup \{...\}_\tau$$
$$\mathcal{S} = C^{ex} \cup C^{err} \backslash \{c_i\}$$

An additional constraint enforces the threshold.

$$|\mathcal{S}| - \eta \leq \mathcal{U}(\alpha)$$

Running Example   Below, an example CSP is depicted that is the result of a B-SCH transformation. Therefore, error-constraint c_3 is chosen to be negated for the invalid input selection. The transformation and the CSP is based on the internal representation of the over-constrained example (Figure 9.3, Page 165).

Error-constraint c_3 : Phone $\neq$ 123 is transformed into the hard-constraint $\overline{c_3}$ with $\neg$(Phone $\neq$ 123) which is $\neg$(PH $\neq$ 3) in the internal representation. Constraints c_1, c_2, c_4, c_5, and c_6 are modeled as soft-constraints. The threshold is enforced by $|\{r_1, r_2, r_4, r_5, r_6\}| - \eta \leq (r_1 + r_2 + r_4 + r_5 + r_6)$.

$$
\begin{aligned}
\mathbb{X} = \{&\text{Pa}, \text{C}, \text{Ph}, \text{S}\} \\
\cup &\{r_1, r_2, r_4, r_5, r_6\} \quad\quad && \text{// reified variables} \\
\mathbb{D} = \{&D_{Pa} = \{1,2,3\}, D_C = \{1,2,3\}, \\
&D_{Ph} = \{1,2,3\}, D_S = \{1,2\}\} \\
\cup &\{D_{r_{1,2,4,5,6}} = \{0,1\}\} \quad\quad && \text{// domains for reified variables} \\
\mathbb{C} = \{&r_1 \Leftrightarrow (\text{Pa} \neq 3), && \text{// soft-constraint } c_1 \\
&r_2 \Leftrightarrow (\text{C} \neq 3), && \text{// soft-constraint } c_2 \\
&r_4 \Leftrightarrow (\text{C} = 1 \Rightarrow \text{Ph} = 1), && \text{// soft-constraint } c_4
\end{aligned}
$$

$$r_5 \Leftrightarrow (\texttt{C} = 2 \Rightarrow \texttt{Ph} = 2),\qquad \text{// soft-constraint } c_5$$
$$r_6 \Leftrightarrow (\texttt{S} = 1 \Rightarrow \texttt{C} \neq 2),\qquad \text{// soft-constraint } c_6$$
$$\neg(\texttt{Ph} \neq 3),\qquad\qquad\qquad\ \text{// negated error-constraint } \overline{c_3}$$
$$5 - \eta \leq (r_1 + r_2 + r_4 + r_5 + r_6)\}\quad \text{// constraint to enforce threshold}$$
$$\cup\{...\}_s\qquad\qquad\qquad\qquad\qquad \text{// schema-constraint}$$

Furthermore, an actual value for the threshold η must be determined. With a threshold of $\eta = 0$, the B-SCH strategy is identical to the HCH strategy since all five soft-constraints must be satisfied. For a threshold of $\eta = 1$, a solution must only satisfy four out of five soft-constraints.

Running Example With a threshold of $\eta = 1$, the following schemata could be selected for error-constraint c_3 because one of the other constraints may remain unsatisfied:

$$[\texttt{Country:UK, Phone:123}],$$
$$[\texttt{Country:USA, Phone:123}],$$
$$[\texttt{Country:123, Phone:123}]$$

Each of them leaves one constraint unsatisfied (c_4, c_5, or c_2). But even though the RIPM is over-constrained, two out of three schemata are truly-strong invalid schemata and should be selected.

To fully automate B-SCH, the threshold η must be determined automatically. It is important to use an appropriate threshold. Otherwise, some MISs could remain absent or too many constraints could remain unsatisfied.

A cardinality-minimal diagnosis set $\triangle_{c_i,j}$ describes a minimal number of constraints to relax in order to resolve all conflicts that prevent $s_j \in M_{c_i}$ from appearing in a strong invalid test input for error-constraint c_i. Hence, a strong invalid test input for $s_j \in M_{c_i}$ can be generated if at least $|\triangle_{c_i,j}|$ constraints may remain unsatisfied.

Since strong invalid test inputs are selected for one error-constraint at a time, the constraint handling strategy and threshold can also be adjusted individually for each error-constraint.

Instead of using a global η for all error-constraints, individual thresholds η_{c_i} can be determined for each error-constraint c_i. To determine one threshold η_{c_i}, the size of cardinality-minimal diagnosis sets $|\triangle_{c_i,j}|$ for each MIS $s_j \in M_{c_i}$ must be checked and the size of the largest cardinality-minimal diagnosis set can be used as the threshold.

$$\eta_{c_i} = \max_{s_j \in M_{c_i}} |\triangle_{c_i,j}|$$

If no conflict exists for error-constraint c_i, no MISs exists ($M_{c_i} = \emptyset$) and the threshold $\eta_{c_i} = 0$ is used which is equivalent to the HCH strategy.

9.3.3. Diagnostic Soft Constraint Handling Strategy

Utilizing the size of cardinality-minimal diagnosis sets allows a complete automation of the B-SCH strategy with individual thresholds for each error-constraint. But, there are two more problems with B-SCH.

First, except for the error-constraint c_i for which invalid inputs are selected, all constraints are modeled as soft-constraints. Since soft-constraints may remain unsatisfied, it can result in unnecessary relaxations. Soft-constraints also add complexity to the CSP by introducing an additional indirection and additional variables for reification.

> **Running Example** An example CSP as the result of a B-SCH transformation was discussed previously (Page 180 and following). The discussed invalid test input selection for error-constraint c_3 has one MIS $s_1 = $ [Phone:123] that is explained by the minimal conflict set $\mathcal{O}_{c_3,1} = \{c_2, c_4, c_5\}$.
>
> Even though it is not involved in any conflict, one can observe that the exclusion-constraint c_6 is modeled as a soft-constraint. For an $eta = 1$, exclusion-constraint c_6 will always be satisfied because either c_2, c_4, or c_5 must remain unsatisfied because of the conflict.
>
> Therefore, it is not necessary to model exclusion-constraint c_6 as a soft-constraint.

Second, the threshold still increases the solution space because thresholds for different invalid schemata of the same error-constraint are not distinguished.

> **Running Example** When selecting invalid test inputs for error-constraint c_4 of the over-constrained example (Figure 9.1, Page 161), two schemata are locally-specified of which one is a MIS.
>
> $$M_{c_4} = \{s_2 = \text{[Country:UK, Phone:123]}\}$$
> $$L_{c_4} = \{s_1 = \text{[Country:UK, Phone:+1]},$$
> $$s_2 = \text{[Country:UK, Phone:123]}\}$$
>
> While [Country:UK, Phone:123] requires relaxation or a threshold of $\eta_{c_4} = 1$, [Country:UK, Phone:+1] is not a MIS and does not require any soft-constraint handling. Since the threshold $\eta_{c_4} = 1$ is effective for all schemata, [Country:UK, Phone:123] may be combined with a schema that violates another constraint like [Packages:123] which is undesirable and should be avoided.

To address both problems, the diagnostic soft constraint handling (D-SCH) strategy exploits the information from minimal diagnosis sets to adjust the constraint handling individually for each locally-specified schema. Both improvements are discussed in the following.

Minimize Number of Soft-Constraints It is desirable to have as few soft-constraints as possible. To distinguish hard- and soft-constraints, we utilize the information from minimal diagnosis sets. For each MIS $s_j \in M_{c_i}$, a set of minimal diagnosis sets $\triangle_{c_i,j}^{all} = \{\triangle_{c_i,j}^1, \triangle_{c_i,j}^2, ...\}$ can be computed.

A constraint $c \in C^{ex} \cup C^{err} \backslash \{c_i\}$ that is not contained in any minimal diagnosis set does not require relaxation. Therefore, it should be modeled as a hard-constraint. A constraint $c \in C^{ex} \cup C^{err} \backslash \{c_i\}$ that is contained in at least one minimal diagnosis set may require relaxation and should be modeled as a soft-constraint.

$$\mathcal{H} = \{\overline{c_i}\} \cup \{...\}_s$$
$$\cup \{c \in C^{ex} \cup C^{err} \backslash \{c_i\} \mid \nexists \triangle_{c_i,j} \in \triangle_{c_i,j}^{all}, c \in \triangle_{c_i,j}\}$$
$$\mathcal{S} = \{c \in C^{ex} \cup C^{err} \backslash \{c_i\} \mid \exists \triangle_{c_i,j} \in \triangle_{c_i,j}^{all}, c \in \triangle_{i,j}\}$$

Running Example   To illustrate the new partitioning of hard- and soft-constraints, we contrast the partitions as computed by B-SCH and D-SCH.

For error-constraint c_3 of the over-constrained example (Figure 9.1, Page 161), the partitioning of B-SCH is as follows.

$$\mathcal{H} = \{\overline{c_3}\} \cup \{...\}_s$$
$$\mathcal{S} = \{c_1, c_2, c_4, c_5, c_6\}$$

The minimal diagnosis sets for the one MIS are again listed below.

$$\triangle_{c3,1}^{all} = \{\triangle_{c3,1}^1 = \{c_2\}, \triangle_{c3,1}^2 = \{c_4\}, \triangle_{c3,1}^3 = \{c_5\}\} \cup \{\triangle_{c3,1}^4 = \{c_3\}\}$$

Based on these minimal diagnosis sets, the D-SCH partitioning is as follows.

$$\mathcal{H} = \{\overline{c_3}\} \cup \{...\}_s \cup \{c_1, c_6\}$$
$$\mathcal{S} = \{c_2, c_4, c_5\}$$

The resulting CSP is shown below.

$$\mathbb{X} = \{\text{Pa}, \text{C}, \text{Ph}, \text{S}\}$$

$\cup \{r_2, r_4, r_5\}$ // reified variables

$$\mathbb{D} = \{D_{Pa} = \{1,2,3\}, D_C = \{1,2,3\},$$
$$D_{Ph} = \{1,2,3\}, D_S = \{1,2\}\}$$

$\cup \{D_{r_{2,4,5}} = \{0,1\}\}$ // domains for reified variables

$\mathbb{C} = \{r_2 \Leftrightarrow (\text{C} \neq 3),$ // soft-constraint c_2

$\quad r_4 \Leftrightarrow (\text{C} = 1 \Rightarrow \text{Ph} = 1),$ // soft-constraint c_4

$\quad r_5 \Leftrightarrow (\text{C} = 2 \Rightarrow \text{Ph} = 2),$ // soft-constraint c_5

$\quad \text{Pa} \neq 3,$ // hard-constraint c_1

$\quad \text{S} = 1 \Rightarrow \text{C} \neq 2,$ // hard-constraint c_6

$\quad \neg(\text{Ph} \neq 3),$ // negated error-constraint $\overline{c_3}$

$\quad 3 - \eta \leq (r_2 + r_4 + r_5)\}$ // constraint to enforce threshold

$\cup \{...\}_s$ // schema-constraint

For error-constraint c_4 of the same example, the partitioning of B-SCH is as follows.

$$\mathcal{H} = \{\overline{c_4}\} \cup \{...\}_s$$
$$\mathcal{S} = \{c_1, c_2, c_3, c_5, c_6\}$$

The minimal diagnosis sets for the one MIS are again listed below.

$$\triangle_{c_4,2}^{all} = \{\triangle_{c_4,2}^1 = \{c_3\}\} \cup \{\triangle_{c_4,2}^2 = \{c_4\}\}$$

Based on these minimal diagnosis sets, the D-SCH partitioning is as follows.

$$\mathcal{H} = \{\overline{c_4}\} \cup \{...\}_s \cup \{c_1, c_2, c_5, c_6\}$$
$$\mathcal{S} = \{c_3\}$$

The resulting CSP is shown below.

$\mathbb{X} = \{\texttt{Pa}, \texttt{C}, \texttt{Ph}, \texttt{S}\}$
$\quad \cup \{r_3\}$ // reified variable
$\mathbb{D} = \{D_{Pa} = \{1, 2, 3\}, D_C = \{1, 2, 3\},$
$\quad D_{Ph} = \{1, 2, 3\}, D_S = \{1, 2\}\}$
$\quad \cup \{D_{r_3} = \{0, 1\}\}$ // domain for reified variable
$\mathbb{C} = \{r_3 \Leftrightarrow (\texttt{Ph} \neq 3),$ // soft-constraint c_3
$\quad \texttt{Pa} \neq 3,$ // hard-constraint c_1
$\quad \texttt{C} \neq 3,$ // hard-constraint c_2
$\quad \texttt{C} = 2 \Rightarrow \texttt{Ph} = 2,$ // hard-constraint c_5
$\quad \texttt{S} = 1 \Rightarrow \texttt{C} \neq 2,$ // hard-constraint c_6
$\quad \neg(\texttt{C} = 1 \Rightarrow \texttt{Ph} = 1),$ // negated error-constraint $\overline{c_4}$
$\quad 1 - \eta \leq (r_3)\}$ // constraint to enforce threshold
$\quad \cup \{...\}_s$ // schema-constraint

As can be seen, this new partitioning can greatly reduce the number of soft-constraints.

Individual Thresholds   To further reduce the solution space, each invalid schema should have an individual threshold.

Invalid schemata $s_j \in \mathcal{I}_{c_i}$ that are not missing ($s_j \notin M_{c_i}$) do not require any relaxation. The threshold should be $\eta_{c_i,j} = 0$ which is equal to the HCH strategy. In contrast, each missing invalid schema $s_j \in M_{c_i}$ should have an individual threshold that corresponds to the size of the

minimal cardinality-minimal diagnosis sets.

$$\eta_{c_i,j} = \min_{\triangle_{c_i,j} \in \triangle^{all}_{c_i,j}} |\triangle_{c_i,j}|$$

To distinguish several individual thresholds for different actual schema under change in one CSP, implications with the following pattern can be used.

$$(\text{<invalid schema>}) \Rightarrow \eta = \text{<value>}$$

Running Example Error-constraint c_3 only specifies one schema $s_1 \in M_{c_3}$ with an individual threshold of $\eta_{c_3,1} = 1$. Therefore, the CSP is semantically equivalent to a CSP without this improvement. Although the changes to define the individual threshold are depicted below.

$\mathbb{X} = \{\text{Pa}, \text{C}, \text{Ph}, \text{S}\}$
    $\cup \{r_2, r_4, r_5\}$ // reified variables
$\mathbb{D} = \{D_{Pa} = \{1, 2, 3\}, D_C = \{1, 2, 3\},$
    $D_{Ph} = \{1, 2, 3\}, D_S = \{1, 2\}\}$
    $\cup \{D_{r_{2,4,5}} = \{0, 1\}\}$ // domains for reified variables
$\mathbb{C} = \{r_2 \Leftrightarrow (\text{C} \neq 3),$ // soft-constraint c_2
      $r_4 \Leftrightarrow (\text{C} = 1 \Rightarrow \text{Ph} = 1),$ // soft-constraint c_4
      $r_5 \Leftrightarrow (\text{C} = 2 \Rightarrow \text{Ph} = 2),$ // soft-constraint c_5
      $\text{Pa} \neq 3,$ // hard-constraint c_1
      $\text{S} = 1 \Rightarrow \text{C} \neq 2,$ // hard-constraint c_6
      $\neg(\text{Ph} \neq 3),$ // negated error-constraint $\overline{c_3}$
      $3 - \eta \leq (r_2 + r_4 + r_5),$ // constraint to enforce threshold
      $(\text{Ph} = 3) \Rightarrow \eta = 1\}$ // constraint for threshold $\eta c_3, 1$
    $\cup \{...\}_s$ // schema-constraint

Error-constraint c_4 specifies two schemata of which one is missing $s_2 \in M_{c_3}$. The individual threshold of the strong invalid schema $s_1 \in I_{c_4}$ is $\eta_{c_4,1} = 0$ while the individual threshold of the MIS is $\eta_{c_4,2} = 1$.

Due to the improvements, $s_1 = $ [Country:UK, Phone:+1] is treated similarly to the HCH strategy which prevents it from being combined with other invalid schemata. While the MIS $s_2 = $ [Country:UK, Phone:123] can still be combined with a schema of the conflicting error-constraint c_3. The corresponding CSP is depicted below.

$\mathbb{X} = \{\text{Pa}, \text{C}, \text{Ph}, \text{S}\}$
    $\cup \{r_3\}$ // reified variable

$\mathbb{D} = \{D_{Pa} = \{1, 2, 3\}, D_C = \{1, 2, 3\},$
$\quad\quad D_{Ph} = \{1, 2, 3\}, D_S = \{1, 2\}\}$
$\quad \cup \{D_{r_3} = \{0, 1\}\}$ // domain for reified variable
$\mathbb{C} = \{r_3 \Leftrightarrow (\texttt{Ph} \neq 3),$ // soft-constraint c_3
$\quad\quad \texttt{Pa} \neq 3,$ // hard-constraint c_1
$\quad\quad \texttt{C} \neq 3,$ // hard-constraint c_2
$\quad\quad \texttt{C} = 2 \Rightarrow \texttt{Ph} = 2,$ // hard-constraint c_5
$\quad\quad \texttt{S} = 1 \Rightarrow \texttt{C} \neq 2,$ // hard-constraint c_6
$\quad\quad \neg(\texttt{C} = 1 \Rightarrow \texttt{Ph} = 1),$ // negated error-constraint $\overline{c_4}$
$\quad\quad 1 - \eta \leq (r_3),$ // constraint to enforce threshold
$\quad\quad (\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = +1) \Rightarrow \eta = 0,$ // constraint for threshold $\eta c_4, 1$
$\quad\quad (\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = 123) \Rightarrow \eta = 1\}$ // constraint for threshold $\eta c_4, 2$
$\quad \cup \{...\}_s$ // schema-constraint

To conclude this section, two alternative constraint handling strategies B-SCH and D-SCH are developed. They are based on the idea of partial constraint satisfaction which allows some constraints to remain unsatisfied during test input selection. Thereby, it is possible to select invalid test inputs for MISs without repairing the over-constrained RIPM beforehand. By utilizing the information provided by minimal diagnosis sets, the CSPs can be formulated much more specific to the individual conflicts as shown above. In the evaluation, both B-SCH and D-SCH strategies are further compared to the HCH strategy.

Both B-SCH and D-SCH strategies enable a fully-automated approach to select invalid test inputs from over-constrained RIPMs. With the combination of alternative constraint handling strategies, manual, and semi-automatic repair activities, three related techniques are presented that allow postponing the repair activities when necessary and that support the repair activities when due. Thereby, the effort of maintaining error-constraints can be reduced.

In the following chapter, another technique is introduced to reduce the initial effort of modeling error-constraints.

Chapter 10.

Automatic Generation of Error-Constraints

Contents

Although the CRT test method promises to avoid the invalid input masking effect by improving test selection of valid and strong invalid test inputs, additional modeling effort is required by the tester to construct error-constraints. Since the manual construction of error-constraints can be considered a error-prone and tedious task, automated support is desirable.

In the previous chapter, we introduced techniques to identify over-constrained RIPMs that are helpful to repair existing and inconsistent error-constraints. However, they are not helpful during the initial modeling of RIPMs when error-constraints are incomplete and under-constrained rather than over-constrained. To also support the initial modeling of error-constraints, an additional technique to automatically support the modeling of error-constraints is introduced in this chapter. The technique relies on FC. Its idea was implemented and its appropriateness was evaluated in the context of a master thesis (cf. [Maß20]).

In the following sections, the general idea is motivated and the process based on FC is presented first. Afterwards, the identification of non-conforming schemata (Section 10.2) and the generation of error-constraints (Section 10.3) is discussed.

10.1. A Process for Automatic Generating of Error-Constraints

The objective is to automatically generate error-constraints such that a tester must not model them manually any longer. In comparison to CT, CRT requires additional effort to model error-constraints. The effort to model parameters, values, and exclusion-constraints is equal for CT and CRT. To focus solely on error-constraints as the missing piece, we assume that parameters, values, and exclusion-constraints already exist. The set of error-constraints is then generated in an iterative manner. Therefore, we assume that the set of error-constraints is either empty, i.e. $C^{err} = \emptyset$, or incomplete and under-constrained.

To automatically generate error-constraints, another system is required which automatically classifies a relevant test input as either truly-valid or truly-invalid. Based on strategically se-

RIPM-based Classification of Test Input	System-based Classification of Test Input	Conclusion
Valid	Truly-Valid	Conforming
Valid	Truly-Invalid	Non-Conforming (C^{err} is under-constrained)
Invalid	Truly-Valid	Non-Conforming (C^{err} is over-constrained)
Invalid	Truly-Invalid	Conforming

Table 10.1.: Conclusions for Conformance between a RIPM and another System

lected test inputs, we expect this to be able to even identify truly-valid and truly-strong invalid schemata.

In general, this approach can be understood as a conformance check between a RIPM with its error-constraints and another system. The result is *conformance* or *non-conformance* between the RIPM and the other system such that both classify a test input or a schema as either valid or strong invalid in an analogous manner. The other system functions as an authoritative specification and could be an executable specification that is "queried" with relevant test inputs. Since executable specifications seldom exist, the other system could also be a SUT that is assumed to be correct. In that case, the behavior of the SUT is observed and a test oracle classifies the test input as valid if it observes standard behavior and it classifies the test input as invalid if it observes exception behavior. As an example, the classification could be based on whether the SUT remains in the normal control-flow (valid test input) or an exception is propagated (invalid test input).

Until the generation of error-constraints is complete, non-conformance between the classifications of the SUT and the other system is expected. Every time both classifications differ, a *conformance fault* exists. Since we assume the other system (executable specification, SUT) to be correct, the only way to fix a conformance fault is to update the error-constraints of the RIPM.

Table 10.1 (Page 188) depicts the four possible classification outcomes and their conclusions. The two non-conforming cases are as follows. First, a test input that is classified as valid by the RIPM but classified as truly-invalid by the other system depicts non-conformance due to under-constrained error-constraints. One or more schemata exist which are truly-minimal invalid according to the other system but no error-constraint exists yet that marks the schema as invalid. This is the typical case that is expected when error-constraints are initially incomplete and under-constrained.

Second, a test input that is classified as invalid by the RIPM but classified as truly-valid by the other system depicts non-conformance due to over-constrained error-constraints. One or more schemata exist which are invalid as specified by some error-constraints. But the schemata are not truly-invalid as specified by the other system. This is possible when error-constraints are modeled manually. Since this case does not occur when error-constraints are generated in an iterative manner, we focus on the first case.

In order to generate error-constraints, it is first necessary to identify test inputs which are non-conforming for the current version of the RIPM. Afterwards, it is necessary to identify the particular cause, i.e. the minimal schemata that explain the non-conformation. Finally, error-

Figure 10.2.: Idealized CRT Process for Automatic Generation of Error-Constraints

constraints must be generated to establish conformance between the RIPM and the other system.

Figure 10.2 (Page 189) depicts the idealized process for automatic generation of error-constraints in more detail. It is an extension of the idealized CT and CRT test process (Figure 3.2, Page 29). The gray background highlights activities that are updated or newly introduced.

Since the process for automatic generation of error-constraints starts with an empty or at least under-constrained set of error-constraints, test inputs that are valid according to the current version of the RIPM can be selected to identify non-conforming test inputs.

Therefore, a test selection strategy for valid test inputs must be chosen first which is a restriction compared to the idealized CT and CRT test process. Further on, the RIPM is modeled. But only parameters, values, and exclusion-constraints are expected to be modeled. Thereby, the initial effort of modeling RIPMs is comparable to the effort of modeling IPMs.

The first feedback circle is equal to the idealized CT and CRT test process. The test selection strategy is applied and the resulting test suite is evaluated. If something is wrong with the test suite, the test selection strategy or the initial RIPM are updated and the test selection strategy is applied again. Otherwise, tests are executed to obtain test results for the test suite. Therefore, a test oracle is required that is capable of identifying the truly-valid or truly-invalid classification of the other system. Since all test inputs are valid according to the RIPM, the other system is expected to label the test inputs as truly-valid. Whenever the other system classifies test inputs as truly-invalid, non-conformance can be deduced.

After executing tests, the test results are evaluated and it is checked whether non-conforming test inputs exist. If at least one non-conforming test input exists, the minimal schemata that cause non-conformance must be identified and error-constraints must be generated to mark the

non-conforming schemata as strong invalid. Both activities are explained in more detail in the following two sections. Afterwards, the RIPM must be updated to contain the new error-constraints and the process must be executed again until no more non-conforming test inputs exist.

If no non-conforming test inputs exist, the current version of the RIPM can be considered as conforming. However, this is no general statement and it is not valid for the entire exhaustive test suite that is defined by the RIPM. It is only valid for the selected and evaluated test suite. To increase the informative value of the statement, the conformance of more test inputs must be checked. Therefore, the testing strength of t-wise valid coverage can be increased or the test input selection can be randomized to select different test inputs that satisfy the same test adequacy criterion. Then, the process for automatic generation of error-constraints starts again. Although, the set of error-constraints is not empty and contains the already generated error-constraints. The process can be completed if, on the other hand, the informative value is considered sufficient.

In the following two sections, the two activities of identifying minimal non-conforming schemata and generating error-constraints are explained in more detail. Since this CRT process for automatic generation of error-constraints is idealized to illustrate and motivate the general idea, solutions for the new activities and its challenges are discussed in-depth.

10.2. **Identifying Non-Conforming Schemata**

After having discussed the idealized process for automatically generating error-constraints, we now discuss how to identify non-conforming schemata. We base the idea of identifying non-conforming schemata on FC which is an additional activity in the CT and CRT test process. First, we explain how FC works and then we discuss how to adapt FC to identify non-conforming schemata.

FC is concerned with identifying MFCSs that can be held responsible for causing one or more test inputs to fail [NL11a]. By knowing the MFCSs that cause test inputs to fail, finding and fixing the related faults can be greatly simplified. To identify MFCSs, FCAs exist that describe in detailed steps how to identify MFCSs and also how to automate the identification of MFCSs. Roughly speaking, FCAs *search* for MFCSs by, first of all, considering all sub-schemata of a failing test input as suspicious and then by searching for evidence to rule out as many sub-schemata as possible (cf. [SNX05]; [LNL12]; [Gha+12]; [Niu+18]; [AGR19]). To rule out a suspicious sub-schema, it is typically searched for a passing test input that covers the suspicious sub-schema. Each sub-schema that is covered by a passing test input is not suspicious anymore because a counter-example to the hypothesis is found. At the end, the remaining set of suspicious sub-schemata is considered as failure-causing (static FCAs, cf. [YCP06]; [CM08]; [LCM19]) or additional test inputs are selected and executed to get more information and to rule out even more suspicious sub-schemata (adaptive FCAs and interleaving FCAs) (cf. [SNX05]; [LNL12]; [Gha+12]; [Niu+18]; [AGR19]). Unfortunately, the FCAs only compute approximate solutions because an exact solution is often times infeasible. Therefore, a variety of FCAs exist. The FCAs can be assigned to one of three different types of FCAs: static, adaptive, or interleaving FCA [NL11a]; [JK18].

With static FCAs, the entire test suite is selected and executed first [CM08]. Then, the information that can be derived from the test results is used to determine MFCSs. When using conventional test selection strategies, not enough information is available to precisely determine MFCSs. Therefore, dedicated test selection strategies exist that select test inputs in a way that supports the identification of MFCSs. However, these dedicated test selection strategies typically require more test inputs than conventional test selection strategies to satisfy the same test adequacy criteria. Further on, they have only limited support for constraints (cf. [JST20]; [JT20]) and can only determine a limited number of MFCSs (cf. [CM08]; [LCM19]).

With adaptive FCAs, the entire test suite is selected and executed first. In the case of failing test inputs, the test results are analyzed by an FCA and maybe more test inputs are selected and executed until the FCA determines the MFCSs. With interleaving FCAs, test inputs are selected and executed one at a time [Niu+18]; [AGR19]. In the case of a failing test input, an FCA analyses the already executed test inputs and their results. Maybe the FCA selects and execute more test inputs to determine the MFCSs. After the MFCSs are determined, the primary test input selection and execution can continue until the next test input fails or the chosen test adequacy criterion is satisfied. The advantage of interleaving FCAs over adaptive FCAs is that the information about MFCSs can be directly used to influence test input selection. This way, fewer test inputs are required because identified MFCSs can be directly excluded from further test inputs.

IPM-based Classification of Test Input	Test Oracle	Conclusion
Relevant	Pass	Conforming
Relevant	Fail	Non-Conforming (C^{ex} is under-constrained)
Irrelevant	Pass	Non-Conforming (C^{ex} is over-constrained)
Irrelevant	Fail	Conforming

Table 10.3.: Conclusions for Conformance between a IPM and a SUT

To determine whether a test input is passing or failing and to identify MFCSs, a test oracle is required to reveal failures. The way how a test oracle labels test inputs after execution determines the FC and the results of FCAs. Besides using FCAs in combination with conventional test oracles that reveal failures of SUTs to determine MFCSs, the idea of FC has already been used to identify conformance faults between IPMs and SUTs.

As discussed in related work (Chapter 4, Page 57 and following), GARGANTINI et al. [GPR17] use FC to automatically identify conformance faults and embedd FC into a process to automatically repair IPMs. They assume that the other system is correct and check if the exclusion-constraints of the IPM are correctly aligned, i.e. if the IPM conforms to the other system.

While a conventional test oracle checks if a SUT behaves as expected for a given test input, the test oracle used by GARGANTINI et al. checks if the given test input is a truly-relevant test input according to another system. A conformance fault occurs if a test input is relevant according to the IPM while the same test input is not relevant according to the other system and test oracle. Or if a test input is irrelevant according to the IPM while the same test input is relevant according to the other system and test oracle. In both cases, the IPM does not conform to the other system. In the first case, the IPM and its exclusion-constraints are under-constrained. In the latter case, the IPM and its exclusion-constraints are over-constrained. Table 10.3 (Page 192) summarizes the four different conclusions when checking for conformance between an IPM and another system.

GARGANTINI et al. use the example of a compiler as a test oracle that checks for conformance of a IPM that represents configurations of a highly-configurable system. Each test input represents a configuration of the highly-configurable system and each test input for which the compiler successfully builds the configuration highly-configurable system is considered as truly-relevant. Each test input for which the compiler fails to build is considered as truly-irrelevant.

When a test oracle that checks for conformance faults is used in combination with FC, non-conforming test inputs are further analyzed and FCAs search for MFCSs that represent minimal non-conforming schemata. After minimal non-conforming schemata are identified via FC, the exclusion-constraints can be repaired such that the IPM conforms to the SUT. For MFCSs that are caused by under-constrained exclusion-constraints, the exclusion-constraints are strengthened. For MFCSs that are caused by under-constrained exclusion-constraints, the exclusion-constraints are relaxed.

Here, we adopt the idea of finding conformance faults based on test execution to align RIPMs with other systems, e.g. the exception conditions of SUTs. Therefore, we assume that the EH of SUTs is correct. Furthermore, a test oracle is required that is able to distinguish standard and

```
Parameters:
  p₁ : Packages:  1, 3, 123
  p₂ : Country:   UK, USA, 123
  p₃ : Phone:     +44, +1, 123
  p₄ : Shipping:  Standard, Express
```

Figure 10.4.: Incomplete RIPM for the Ordering Web-Service Example

exception behavior of SUTs. Since only valid test inputs are selected, a test oracle is required to label truly-valid test inputs as passing and to label truly-invalid test inputs as failing. For instance, a test oracle could label test inputs as failing whenever SUTs propagate exception representations, respond with error-codes or error messages. Based on this information, FCAs can search for MFCSs that represent truly-minimal invalid schemata that are not yet modeled by error-constraints.

Running Example To illustrate the process for automatic generation of error-constraints and the identification of non-conforming schemata , Figure 10.4 (Page 193) depicts a RIPM for the ordering web-service example where the error-constraints are not yet modeled. For the sake of simplicity, the *known fault* (cf. Figure 6.1, Page 113) that is included in previous RIPMs is not considered here. The example implementation (Listing 2.1, Page 14) is considered to be correct. It is used as the other system to which the RIPM should conform. Therefore, test inputs are selected and the other system classifies them as truly-valid or truly-invalid according to the following validation rules.

```
1  if(isInvalidNumberOfPackages(packages)) return "INV_PACKAGES";
2  if(isInvalidCountry(country)) return "INV_COUNTRY";
3  if(isInvalidPhoneNumber(phone)) return "INV_PHONE";
4  if(isInvalidCountryCode(country, phone)) return "INV_CODE";
```

After executing the process for automatic generation of error-constraints, all strong invalid schemata shall be correctly identified. They are depicted below.

```
[Packages:123]
[Country:123]
[Phone:123]
[Country:UK, Phone:+1]
[Country:USA, Phone:+44]
```

For this example, the TRT (Tuple Relationship Tree) [Niu+13] FCA is used as an interleaving FCA in combination with the AETG [Coh+94]; [Coh+97] test selection strategy and a testing strength of $t = 1$.

The first selected test input is truly-valid and therefore passes the test. As a consequence, the coverage information is updated and AETG selects the next input. The second test input is classified as truly-invalid (INV_PACKAGES) by the other system. Both test

inputs are depicted below.

```
[Ph.:+1, Co.:USA, Sh.:Standard, Pa.:3] ↦ SUCCESS
[Ph.:+44, Co.:UK, Sh.:Express, Pa.:123] ↦ INV_PACKAGES
```

Consequently, the test fails and the FCA takes over control to identify the non-conforming schema to explain the failure. Since only two test inputs are executed until now, the available information is insufficient to identify a non-conforming schema. Therefore, the FCA selects additional test inputs and executes them until sufficient information is available.

```
[Ph.:+44, Co.:USA, Sh.:Express, Pa.:123] ↦ INV_PACKAGES
[Ph.:+44, Co.:123, Sh.:Express, Pa.:123] ↦ INV_PACKAGES
[Ph.:+44, Co.:USA, Sh.:Express, Pa.:3] ↦ INV_CODE
[Ph.:+44, Co.:123, Sh.:Express, Pa.:1] ↦ INV_COUNTRY
[Ph.:+44, Co.:UK, Sh.:Express, Pa.:1] ↦ SUCCESS
...
[Ph.:+44, Co.:UK, Sh.:Express, Pa.:3] ↦ SUCCESS
```

Once sufficient information is available, `[Packages:123]` is correctly identified as the non-conforming schema. Afterwards, the schema is stored internally to exclude it from further test selection and the conventional test selection with AETG resumes.

```
[Ph.:123, Co.:123, Sh.:Standard, Pa.:1] ↦ INV_COUNTRY
```

Since the very next test input is again non-conforming, the FCA again takes over control. This time, two non-conforming schemata are identified because the first non-conforming test input covers two, e.g. `[Phone:123]` and `[Country:123]`.

The schemata are again stored and the conventional test selection with AETG resumes. Since the previously selected and executed test inputs already satisfy t-wise valid coverage with $t = 1$, the identification of non-conforming schemata terminates at this point.

The two yet unidentified non-conforming schemata `[Country:UK, Phone:+1]` and `[Country:USA, Phone:+44]` never appeared in test inputs selected by AETG. To identify them, AETG must select test inputs that cover them. Therefore, a testing strength $t = 2$ can be used.

Alternatively, the three identified minimal non-conforming schemata can be added to the RIPM as error-constraints and the process for automated generation of error-constraints can be executed again with $t = 1$. The details how minimal non-conforming schemata can be transformed into error-constraints is discussed in the next section.

Although the appearance of `[Country:USA, Phone:+44]` or `[Country:UK, Phone:+1]` cannot be guaranteed when testing with $t = 1$, the probability may increase because of the generated error-constraints.

In this example, a test input selected by `AETG` covers `[Country:USA, Phone:+44]` which is then identified as non-conforming. After adding it to the RIPM as a fourth error-constraint, `[Country:UK, Phone:+1]` is identified in a third iteration of the process.

10.3. Generating New Error-Constraints

After a FCA identified one or more MFCSs that represent truly-minimal invalid schemata, the next step is to generate error-constraints in order to align the RIPM and the other system. The generated error-constraints can then be used to update the RIPM such that the RIPM classifies the truly-minimal invalid schemata as invalid. In this section, we discuss how identified MFCSs are grouped and translated into logical expressions to generate separate error-constraints. Updating the RIPM is a simple activity that does not require further discussion.

A simple strategy is to create a separate error-constraint for each identified MFCS. But then no a-wise combinatorics are possible and all proposed test adequacy criteria (Chapter 7, Page 119 and following) are equivalent. Another simple strategy is to create one error-constraint that contains all identified MFCSs. But since the parameters of all identified MFCSs are not necessarily identical, additional schemata may be locally-specified (See Chapter 6, Page 111 and following). This is because the computation relies on a Cartesian product that is build of the value domains of all parameters. As a consequence, test input selection can result in too many test inputs and some EH may be tested redundantly or it can result in too few test inputs and some EH may remain untested. A third simple strategy is to group MFCSs according to the parameters they cover. Thereby, the computation of additional schemata is reduced but some EH may remain untested when only a subset of schemata is selected for testing.

> **Running Example**   Continuing with the previous example and its five identified non-conforming schemata, the next step is to group them according to the aforementioned strategies. Therefore, we introduce keys and map the schemata to them.
>
> The first simple strategy to generate separate error-constraints uses a distinct key for each schema.
>
> $$1 \mapsto \texttt{[Packages:123]}$$
> $$2 \mapsto \texttt{[Country:123]}$$
> $$3 \mapsto \texttt{[Phone:123]}$$
> $$4 \mapsto \texttt{[Country:UK, Phone:+1]}$$
> $$5 \mapsto \texttt{[Country:USA, Phone:+44]}$$
>
> The second simple strategy to generate one error-constraint for all schemata relies on one key for all schemata.
>
> $$1 \mapsto \texttt{[Packages:123]}$$
> $$1 \mapsto \texttt{[Country:123]}$$
> $$1 \mapsto \texttt{[Phone:123]}$$
> $$1 \mapsto \texttt{[Country:UK, Phone:+1]}$$
> $$1 \mapsto \texttt{[Country:USA, Phone:+44]}$$
>
> The third simple strategy uses the same key for the last two schemata because they

share the same set of parameters.

```
1 ↦ [Packages:123]
2 ↦ [Country:123]
3 ↦ [Phone:123]
4 ↦ [Country:UK, Phone:+1]
4 ↦ [Country:USA, Phone:+44]
```

To ensure that each EH is tested, we rely on strategies that rely on the other system's response rather than structural properties of the identified MFCSs. In particular, we introduce the following two strategies.

The Maximum Count Strategy For each MFCS, the maximum count strategy iterates over all executed test inputs that cover the MFCS to identify the most frequent response, e.g. the most frequent error code, of the other system. After the most frequent response is identified for each MFCS, all MFCSs with the same most frequent response are grouped together.

The Isolation Strategy For each MFCS, the isolation strategy searches for a test input that covers only the MFCS but no other MFCSs. Then, the response of the other system is attached to the MFCS. After a response is attached to each MFCS, all MFCSs with the same attached response are grouped together. An isolating test input is a test input that covers exactly one MFCS. Since the MFCS represent non-conforming truly-minimal invalid schemata, an isolating test input represents a truly-strong invalid test input. To find isolating test inputs, additional test inputs may be necessary unless a FCA like `MixTgTe` [AGR19] is used that already searches for isolating test inputs as part of the identification of MFCSs.

Running Example When using either the maximum count strategy or the isolation strategy, the schemata are mapped to the error codes that each schema triggers in the other system.

```
INV_PACKAGES ↦ [Packages:123]
INV_COUNTRY ↦ [Country:123]
INV_PHONE ↦ [Phone:123]
INV_CODE ↦ [Country:UK, Phone:+1]
INV_CODE ↦ [Country:USA, Phone:+44]
```

After the identified MFCSs are grouped, error-constraints can be generated for one group at a time. Therefore, let G_i denote a group of MFCSs with index i. To generate error-constraint c_i, the `translate-schema(s)` function translates each MFCS $s \in G_i$ into a logical expression. First, each parameter-value pair $\pi = (p, v) \in s$ of the MFCS s is translated into a proposition $(p = v)$. Then, all propositions are combined via logical conjunction and the conjunction is

negated.

$$\texttt{translate-schema}(s) = \neg\Big(\bigwedge_{(p,v)\in s} (p = v)\Big)$$

Furthermore, all translated schemata of one group G_i are combined via logical conjunction as described by the $\texttt{translate-group}(G)$ function.

$$c_i = \texttt{translate-group}(G_i) = \bigwedge_{s\in G} \texttt{translate-schema}(s)$$

Thereby, the error-constraint c_i is not satisfied by each test input that covers at least one schemata s of the group G_i. But error-constraint c_i is satisfied by each test input that covers no schemata of group G_i.

> **Running Example**   For instance, [Country:UK, Phone:+1] is translated into the
> following error-constraint when using the first simple strategy.
>
> $$\neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = +1)$$
>
> When using either the maximum count strategy or the isolation strategy, [Country:UK, Phone:+1] and [Country:USA, Phone:+44] are grouped together
> and translated into the following error-constraint.
>
> $$\neg(\texttt{Country} = \texttt{UK} \wedge \texttt{Phone} = +1) \wedge \neg(\texttt{Country} = \texttt{USA} \wedge \texttt{Phone} = +44)$$

After all groups are translated, the RIPM can be updated to reflect the new conformance. We neglect further details of updating the RIPM because we consider it to be a simple activity. When updating the RIPM manually, the process execution terminates after all groups are translated. The tester then adds the generated error-constraints to the stored representation of the RIPM. Afterwards, the next iteration of the process for automatic generation of error-constraints is started and the updated RIPM is used.

When updating the RIPM automatically, the generated error-constraints are added to the in-memory representation of the RIPM and the next iteration of the process for automatic generation of error-constraints automatically uses the updated RIPM.

In this chapter, a process for automatic generation of error-constraints is introduced. Furthermore, the details of identifying non-conforming schemata and the details of generating error-constraints are discussed. With this information, the idealized process for automatic generation of error-constraints can be operationalized and executed. Thereby, the effort of modeling error-constraints is reduced and becomes comparable to the effort of CT.

In the corresponding evaluation (Chapter 15, Page 259 and following), the general feasibility of the process for automatic generation of error-constraints is demonstrated. Further on, the suitability of FC and FCAs is further analyzed.

Chapter 11.

A Framework for Automated Combinatorial Robustness Testing

Contents

The advantages of CRT only become apparent through a high level of automation. Therefore, the test selection strategies and supportive techniques presented in this work must be operationalized by appropriate tools and automation. This is not only important for the acceptance and adoption of CRT in practice but also for the evaluation of this work.

In this chapter, we, therefore, present a process, an architecture and an implementation of a tool that automates the test selection strategies and supportive techniques presented in this work. The tool is designed as a reusable and extensible framework. First, the process of the framework is illustrated. Afterwards, an architecture and an implementation are presented.

11.1. Orchestrated Process

The effort to model a RIPM is relatively low when compared to the manual design of test inputs. Moreover, techniques like FC, the detection and repair of over-constrained RIPMs, or the automatic generation of error-constraints can further support the test process and reduce the effort of modeling RIPMs.

If, on the other hand, test selection strategies and supportive techniques have to be carried out manually, the advantages of CRT are greatly reduced or do not even come into play as manually following the algorithmic instructions is very tedious and cumbersome. Although the number of test inputs is reduced by combinatorics, the number of test inputs is often still too high to be executed manually. Therefore, it is important to not only automate the test selection strategies and supportive techniques but also to automate the test execution and to integrate all activities into a fully automated test process.

As already mentioned, the tool is designed as a reusable and extensible framework. In general, frameworks define the control-flow and mostly act as orchestrators of configurable *extension points*, for which a user can either choose between a selection of framework-provided options or custom options [JF88].

Figure 11.1.: Process of the Test Automation Framework

Frameworks play an important role in test automation. Because the orchestration is similar from test project to test project, test selection strategies and supportive techniques can be extracted into a framework. The testers can then register their RIPMs, test scripts, and further configurations via defined extension points and the framework orchestrates test selection, test execution, and all other activities.

Figure 11.1 (Page 200) depicts the process that the framework implements. In the following, the activities of the process are discussed. The first three activities are manual activities which must be performed by the tester. They are highlighted in the figure by a gray background.

Model RIPM   To automate test selection strategies and supportive techniques, it is necessary to provide a meta-model structure that enables modeling of RIPMs with parameters, values, exclusion--constraints, and error-constraints. The RIPMs must be available in later activities.

Define Parameterized Test Script   To integrate all activities into a fully automated process, it is of vital importance to automate the test execution. A test script encodes the sequences of steps used to stimulate the SUT and to observe its behavior [UL07]. In the context of CRT, we expect the same sequence of steps to be executed again and again with varying test inputs. Therefore, we rely on *parameterized* test scripts which are test scripts with placeholders (variables) to represent variation points.

Besides observing the behavior of a SUT for a test input, it is important to determine whether the behavior is as expected or a failure. We assume the presence of an automated test oracle which labels a test input as passing or failing. The test scripts must be available and executable in later activities.

Configure the Process   An important of this framework is its reusability and extensibility. Although, processes are similar from test project to test project, there are subtle differences. Hence, it is necessary to configure the framework for each test project and a corresponding meta-model structure is required to describe configurations. Examples of configurations are different test selection strategies shall be used during test input selection, the testing strength that test suites shall satisfy, or the implementation of constraint handling that shall be used.

With these three manual activities, the framework can be configured and tailored to specific tst projects and processes. Next, the activities that must be automated and integrated are discussed.

Detect and Repair of Over-Constrained RIPMs   When enabled, the first automated activity is to detect, explain, and repair conflicts among error-constraints. Therefore, the algorithms for detection and semi-automatic repair of over-constrained RIPMs must be automated. The input required is a RIPM as well as the configuration that defines which strategies for conflict detection and conflict diagnosis should be used. The results of this activity are identified MISs, conflict sets, and minimal diagnosis hitting sets.

Depending on the configuration, the process can either be cancelled when MISs are detected, or the process can continue with the information being used by the constraint solver (soft-constraint handling strategies) during test input selection.

Select Test Inputs   To automate the selection of test inputs, an automated test selection strategy is necessary. The input required is a RIPM as well as the configuration that defines which test selection strategy to use and which testing strength to satisfy are required. Furthermore, the configuration defines the constraint solver that should be used. In the case of soft-constraint handling strategies, the information computed by the previous activity is also used. After executing the automated test selection strategy, the output of this activity is a set of test inputs.

Execute Parameterized Test Script   To automate the test execution, the parameterized test script and the test suite of the previous activity are required. The parameterized test script is *instantiated* for each test input of the test suite and executed. As the result of this activity, each test input is labeled with `pass` or `fail`.

Characterize Faults   When one or more test inputs fail, FC supports the tester in diagnosing to identify the failure causes. Often, not the test inputs but covered MFCSs are responsible for causing the failure. FC and its FCAs can automatically search for MFCSs.

Therefore, an automated FCA is necessary. As input, a FCA requires the IPM or RIPM, the selected test inputs, and its results. Then, either MFCSs are computed when the available information is sufficient. Otherwise, additional test inputs are selected to increase the informative

value. To label the additional test inputs with `pass` or `fail`, this activity must be able to trigger the automated test execution for them. The result of this activity is a set of MFCSs.

Generate Error-Constraints   As already discussed, CRT requires the additional effort of modeling error-constraints. To also support the initial modeling of error-constraints, an automatable technique is introduced in this work. The technique utilizes the automated test execution and the fault characterization to identify schemata that should be modeled by error-constraints Therefore, the test oracle in the parameterized test script is not used to reveal failures of the SUT. Instead, the SUT is considered to be correct and the test oracle is used to reveal nonconformity between the RIPM and the SUT. The identified minimal non-conforming schemata are then translated into error-constraints.

Due to the conceptual difference between testing with FC, where faults in the SUT are expected, and the error-constraint generation, where errors in the system are ruled out, the activities are mutually exclusive. Either a SUT is tested and FC supports the diagnosis of failures or error-constraints are generated and FC supports the identification of minimal non-conforming schemata. For this it is important that the configuration and the test oracle in the test script match either to test a SUT or to generate error-constraints.

It is important not only to automate the individual activities but also to automate the integration and orchestration of the entire process across individual activities. In the following chapter, we therefore show an architecture that implements and integrates these activities.

Figure 11.2.: Architecture Overview of the Test Automation Framework

11.2. Framework Architecture

An overview of the architecture is shown in Figure 11.2 (Page 203). The architecture is based on three layers. Thereby, the manual activities of modeling RIPMs, defining parameterized test scripts, and configuring the process are decoupled from those automated activities that are orchestrated by the framework. Further on, the details of the orchestration are decoupled from algorithmic details.

The top layer is said to be platform-dependent while the other two layers are said to be platform-independent. The top layer *platform-specific orchestration* is the interface to the tester. This layer is platform-specific, because decisions such as the format of the user interface (graphical, textual, etc.) and the type of software (standalone, web, embedded in another test automation framework, etc.) must be made at this level. The other two layers are platform-independent and can be reused with different user interfaces and in different types of software.

In the following, the three layers are discussed in more detail.

The lowest layer *platform-independent algorithmic* contains implementations of four separate automated activities for conflict detection and repair, test input selection, FC, and error-constraint generation. The fifth activity to instantiate a parameterized test script and to execute it for a test input is not included in this layer because it is very technology-dependent.

These four automated activities can be further customized by choosing different algorithms. Algorithms can be provided by the implementation of the framework or they can be added by the tester. The algorithms only need to implement the appropriate interface and be selected in the configuration. Then, the top layer will arrange the dependencies according to the configuration.

Configurations and the management of dependencies will be explained in more detail in the context of the top layer.

At the lowest layer, the four activities are realized. However, they are independent of each other and not integrated. Therefore, there is the middle layer which is responsible for the orchestration of the activities.

The *process orchestrator* is the central concept of the middle layer. A concrete variant of the orchestration represents a specific flow through the process which executes some of the four automated activities and exchanges information between them. For instance, the *sequential test process orchestrator* represents the *ordinary* test process to test a SUT with successive activities for test input selection, test input execution, and FC. In contrast, the *interleaving error-constraint generator* represents the process to generate error-constraints with the activities of test input selection and identification of non-conforming schemata being interleaved.

Further implementations of the process orchestration are also possible. This allows new process flows to be described and further activities to be integrated. For this purpose, only the interface for process orchestration must be implemented and the configuration must refer to the new implementation.

Besides orchestrating the activities of the lower layer, reporting and test input execution are two important activities of the middle layer. The reporter activity is about communicating the current status of the process such as test results, identified MFCSs or generated error constraints to the user. Test input execution refers to the activity of instantiating the parameterized test script for a test input, executing it, and returning a label `pass` or `fail` for the test input.

Since both reporting and execution are platform-specific, there are only interfaces for them on the middle layer. The top layer must implement these interfaces and transfer instances to the process orchestration via the management of dependencies.

The top layer represents the user interface to the tester, where testers provides their inputs for the test, e.g. modeling RIPMs, defining parameterized test scripts, and providing configurations for the processes. Furthermore, the tester is also informed about the current status of the process, e.g. the currently executed activity, results of test input executions, or identified MFCSs.

Furthermore, the top layer acts as a trigger for the automatic activities when the 3 manual activities are completed and the tester wants the other activities to start. After the trigger, the *platform-specific test engine* is executed, which first reads the RIPM, the parameterized test script and the configuration. Based on the configuration, a process orchestrator is then instantiated and further dependencies such as the parameterized test script, the reporter and specific algorithms for the four activities are arranged. Then, the process orchestrator is started, which executes the process and its activities.

By the division into three layers, the different concerns can be strictly separated and the individual concepts of each layer also remain interchangeable and reusable.

The process and architecture only describe the conceptual design of the framework. In order to actually use the framework in practice and for evaluation, it must also be implemented. This is described in the next section.

11.3. Implementation of coffee4j

The process and architecture concepts shown above represent only a part of what is implemented. With the support of 6 bachelor and master theses (cf. [Bon18]; [Kir19]; [Fri19]; [Ber19]; [Maß20]; [Bon20]), the framework is implemented using the Java[1] programming language and integrated into the JUnit[2] platform for test automation. The implementation is called `coffee4j` (combinatorial test and fault characterization framework for java) and published as open-source[3].

To introduce the `coffee4j` implementation, Listing 11.1 (Page 207) depicts a test for the ordering web service example (Listing 2.1, Page 14). The example contains all three inputs that are required from the tester: the RIPM (lines 3 to 32), the parameterized test script (lines 36 to 45), and the configuration (lines 34 and 35).

The configuration is the central link and the annotation `@CombinatorialTest` is mandatory for a combinatorial test to be recognised. Once the RIPM, the parameterized test script, and the configuration are ready, JUnit is started, which recognises the `@CombinatorialTest` annotation and thereby initiates our platform-specific test engine of `coffee4j`. As soon as the platform-specific test engine is started, further configurations are evaluated and the RIPM is retrieved. Depending on the configuration, the next step is to initialise a process orchestrator and start the execution of the process.

The `inputParameterModel = "model"` attribute (line 34) of the `@Combinatorial-Test` annotation provides a reference to the RIPM. Furthermore, `@EnableGeneration` is used to select the sequential test process orchestrator. If another process orchestrator is to be used, for instance the interleaving orchestrator, there are alternative annotations like `@En-ableInterleavingConstraintGeneration`. Attributes such as `algorithms = { Ipog.class, IpogNeg.class}` (line 35) can be used to further configure which algorithms should be used in the automated activities.

The configuration is always attached to a parameterized test script. Therefore, no additional reference is required. The parameterized test script is realized as a Java method with arguments linked to the parameters of the RIPM. The link is established by another annotation `@Input-Parameter` (lines 36 to 39). The realisation as Java methods allows describing the instructions of parameterized test scripts with Java expressions and statements. Furthermore, all Java-based objects and classes are supported by the arguments. Thereby, Java-based components can be tested directly without the need to implement additional adapters and translators.

The referenced RIPM describes the parameters in the same way, whereby the types of the parameters must match the types of the arguments. The RIPM is implemented as an object graph and consists of a set of parameters with assigned values as well as a set for exclusion-constraints and a set for error-constraints. To model a RIPM, there is a *fluent* API that matches the object graph and improves the readability. To model a RIPM, other approaches are also possible, such as reading files or interpreting other internal APIs, as long as this can be described within a static method.

---

[1]See https://java.com (Last access on 1st December 2020).
[2]See https://junit.org (Last access on 1st December 2020).
[3]See https://coffee4j.github.io (Last access on 1st December 2020).

As proposed by SHERWOOD [She14]; [She15]; [She16]; [She17], constraints are also realized as methods with arguments rather than as expressions based on propositional logic. Although, to work with the constraints in conflict detection or conflict diagnosis algorithms as well as in test selection strategies, they are translated into schemata or logical expressions internally. Here, the arguments of the methods are again linked to the parameters and the argument types must also the parameter match. Constraints also have an optional name to reference them later, e.g. as part of a test oracle.

The example shows the most important concepts of the `coffee4j` framework. Besides the already mentioned process orchestrators *sequential test process orchestrator* and *interleaving error-constraint generator*, the following algorithms are integrated into `coffee4j` for the following evaluation.

As test selection strategies, `AETG` [Coh+94]; [Coh+97], `IPOG-C` [Lei+07]; [Yu+13a]; [Yu+15], and `ROBUSTA` with the entire `IPOG-NEG`⟨∗,∗⟩ family of test selection strategies are implemented. All test selection strategies are based on the suggestions by KLEINE & SIMOS [KS18]. Furthermore, the `choco-solver`[4] library that provides a solver for CSPs and the `fastutil`[5] library that provides fast collections for Java are particularly noteworthy.

For the detection and repair of over-constrained RIPMs, `QuickXPlain` [Jun04] is used as a conflict detection algorithm and `HS-Tree` [Rei87]; [Fel+14] is used as a conflict diagnosis algorithm.

Furthermore, FC and the automatic generation of error-constraints are based on FCAs. Therefore, four FCAs are implemented: `IDD` [LNL12], `OFOT` [Niu+18], `TRT` [Niu+13], and `MixTgTe` [AGR19].

With this, the concept consisting of a fault model, the RIPM structure, test adequacy criteria, test selection strategies, and supporting techniques is completely operationalised and can be used in practice as well as in the subsequent evaluation.

---

[4]See https://choco-solver.org (Last access 1st December 2020).
[5]See http://fastutil.di.unimi.it (Last access 1st December 2020).

```java
public class OrderingWebServiceTest {

  private static InputParameterModel model() {
    return inputParameterModel("ordering-web-service-model")
      .parameters(
        parameter("Packages").values(1, 3, null),
        parameter("Country").values("UK", "USA", null),
        parameter("Phone").values("+44", "+1", null),
        parameter("Shipping").values(Standard, Express)
      ).errorConstraints(
        constrain("Packages")
          .withName("INV_PACKAGES")
          .by((Integer packages) -> packages != null),
        constrain("Country")
          .withName("INV_COUNTRY")
          .by((String country) -> country != null),
        constrain("Phone")
          .withName("INV_PHONE")
          .by((String phone) -> phone != null),
        constrain("Country", "Phone")
          .withName("INV_CODE")
          .by((String country, String phone) -> {
            if("UK".equals(country) && "+1".equals(phone))
              return false;
            if("USA".equals(country) && "+44".equals(phone))
              return false;

            return true;
          }
        )
      ).build();
  }

  @CombinatorialTest(inputParameterModel = "model")
  @EnableGeneration(algorithms = { Ipog.class, IpogNeg.class })
  void test(@InputParameter("Packages") Integer packages,
            @InputParameter("Country") String country,
            @InputParameter("Phone") String phone,
            @InputParameter("Shipping") ShippingType shipping) {
    final String result
      = processOrder(packages, country, phone, shipping;
    final String expectedResult = ...;

    assertEquals(expectedResult, result);
  }
}
```

Listing 11.1: Example of an Ordering Web Service Test

Part IV.

Evaluation and Conclusion

Chapter 12.

Overview of Evaluation

The research in this work is concerned with a design problem [Wie14], i.e. improving the way how test inputs are selected by introducing a new test method and additional supporting techniques. Design problems are in general a "call for a change in the real world and require an analysis of actual or hypothetical stakeholder goals" [Wie14]. In this work, the design problem is the invalid input masking effect caused by EH and the stakeholder goal is to minimize potential causes by avoiding the invalid input masking effect.

A solution to a design problem is a design [Wie14]. An important characteristic of design problems and their solutions is that many different solutions can exist. In this work, a newly introduced CRT test method is a design. Although, it is only one design besides already existing CRT test methods (See related work, Chapter 4, Page 55 and following). Since many designs can exist for a single design problem, there is not only one correct and best design [Wie14]. Instead, designs must be evaluated by their utility with respect to the stakeholder goals.

In the following, our research questions and our evaluation approaches to conclusively answer the research questions are discussed.

The first research question (Chapter 1, Page 7 and following) is a knowledge question about the context of a design problem [Wie14], i.e. SUTs with EH that causes invalid input masking.

Research Question 1 How effective is CT and what are its shortcomings in detecting faults when invalid input masking is present?

The answer to the knowledge question motivates the design problem as it explicates the problem of invalid input masking. The question is addressed by a controlled experiment (Section 5.1, Page 74 and following).

Evaluating Combinatorial Robustness Test Selection Strategies The remaining five research questions target the design problem. The first two research questions target the CRT test method directly while the remaining three research questions target supporting techniques.

Research Question 2 How can the t-factor fault model be refined such that it explicitly describes robustness faults and the inherent invalid input masking effect?

Research Question 3 How can a CRT test method capture and utilize the additional information for annotating invalid values and invalid value combinations to select test suites that activate faults in the presence of invalid input masking?

Research Question 3.1 How can the structure of IPMs be extended such that IPMs capture semantic information that annotate invalid values and invalid value combinations?

Research Question 3.2 How can test adequacy criteria be defined such that their satisfaction ensures the activation of faults as described by the refined t-factor fault model?

Research Question 3.3 How can test selection strategies be defined such that they select small test suites that satisfy the test adequacy criteria?

These research questions provide a fine-grained structure to design a CRT test method. Besides the design of our CRT test method which is discussed in the particular chapters, it is important to demonstrate and evaluate that our CRT test method is more useful than previously existing CRT and CT test methods.

In this regard, we evaluate our CRT test method as a whole by applying the developed test selection strategies and do not compare individual parts of it. This is because the fault model is based on a classification of robustness fault characteristics which is already assessed by the case study on MFCSs. The developed test adequacy criteria are argumentatively distinguished from existing test adequacy criteria and the relationships of test adequacy criteria among each other are already discussed using the subsumption relation. Moreover, test adequacy criteria are indirectly assessed through the test selection strategies that are designed such that they satisfy test adequacy criteria while minimizing the number of required test inputs.

Since our CRT test method is already compared to existing CRT test methods in the related work (Chapter 4, Page 55 and following), we focus on comparing our CRT test method to CT test methods. Therefore, we first provide a theoretical discussion of worst-case space and time complexity that allows comparing the developed test selection strategies with already existing and commonly used test selection strategies.

The development of the CRT test method is motivated by an initial controlled experiment that assesses weaknesses of the CT test method in the presence of invalid input masking. Consequently, the question is raised whether the developed CRT test method performs better than the existing CT test method. Therefore, we use another controlled experiment with comparable settings to evaluate and compare the performance of CRT with the performance of CT in the presence of invalid input masking. This controlled experiment requires an implementation of the RIPM structure and of a test selection strategy which are then applied to select test inputs according to one of the test adequacy criteria. Therefore, the controlled experiment evaluates not only test selection strategies but the CRT test method as a whole.

A case study with an industry partner further allows evaluating the practical applicability of the CRT test method and provides another way to compare the CRT test method with the CT test method.

Evaluating Detection and Repair of Over-constrained RIPMs   The first of the three remaining research questions is concerned with over-constrained RIPMs which we experienced while applying our CRT test method.

Research Question 4 How can inconsistent annotations of invalid values and invalid value combinations be detected, identified, and resolved?

The CRT test method requires additional effort in modeling error-constraints. We further acknowledge that it is easy to create over-constraints RIPMs resulting in incorrect partitions of valid and strong invalid schemata. To support the repair of over-constrained RIPMs, techniques to detect and repair conflicts are proposed. Further on, alternative and fully-automatable SCH strategies are proposed to postpone the repair of over-constrained RIPMs.

Since no comparable approaches for the CRT test method exist yet, we cannot compare our techniques with them. Therefore, we use benchmark RIPMs to conduct further experiments to evaluate the general feasibility of our techniques and to make comparisons between them.

Evaluating Automatic Generation of Error-Constraints   The automatic generation of error-constraint is another approach to further reduce the effort of modeling error-constraints.

Research Question 5 How can values and value combinations that trigger EH of a SUT be identified during testing and annotated as invalid?

As a constructive solution to this research question, we proposed a process that is based on FC and we further proposed the usage of FCAs to identify schemata that trigger EH. Since again no comparable approaches for the CRT test method exist yet, we cannot compare our techniques with them. Instead, we evaluate the feasibility of the approach in general and the feasibility of FCAs in terms of preciseness and completeness.

All research questions and all of the evaluations have now been discussed. The evaluations are now being carried out in the following.

Chapter 13.

Evaluating Combinatorial Robustness Test Selection Strategies

Contents

To compare the CRT test method with the CT test method, test selection strategies of each method are applied and compared.

We first provide a theoretical evaluation regarding the worst-case time and space complexity. Afterwards, we motivate and conduct a controlled experiment to assess differences in effectiveness and efficiency. Finally, we conduct a case study and apply CRT as well as CT to real SUTs from one of our industry partners to gather more information regarding the practicality of CRT.

13.1. Theoretical Evaluation

When directly comparing the `ROBUSTA` test selection strategy with other CT test selection strategies, it becomes apparent that `ROBUSTA` has an additional indirection, i.e. the `while`-loop, which separates the selections of test inputs for individual error-constraints and delegates the selections to test selection strategies of the `IPOG-NEG⟨∗,∗⟩` family. Since `IPOG-NEG⟨∗,∗⟩` test selection strategies are based on `IPOG-C`, we can compare the complexity of `ROBUSTA` and the `IPOG-NEG⟨∗,∗⟩` test selection strategies with `IPOG-C`.

Therefore, let t refer to the testing strength, let v refer to the largest domain size $|V_i| \in V$, and let n refer to the number of parameters $|\tilde{P}|$.

The space complexity of IPOG-C is dominated by the storage of t-sized schemata that must be covered by the final test suite [Lei+07]; [Lei+08]. Every time the initial test suite is extended by another parameter, at most $\binom{n-1}{t-1}$ interactions among parameters exist which is in $O(n^{t-1})$ [Lei+07]; [Lei+08]. Further on, each interaction of parameters relates to at most v^t schemata. Therefore, the space complexity is in $O(v^t \times n^{t-1})$.

Since the IPOG-NEG$\langle *, * \rangle$ test selection strategies are invoked sequentially and the schema storage is not changed, the space complexity of IPOG-NEG$\langle *, * \rangle$ is also in $O(v^t \times n^{t-1})$.

Because of the additional indirection, the worst-case time complexity of ROBUSTA is approximately the worst-case time complexity of IPOG-C and $|C^{err}|$-times the worst-case time complexity of IPOG-NEG$\langle *, * \rangle$.

To classify the time complexity more precisely, we first discuss the worst-case time complexity IPOG-C. Afterwards, we transfer it to ROBUSTA and the IPOG-NEG$\langle *, * \rangle$ family of test selection strategies.

As discussed by LEI et al. [Lei+07]; [Lei+08], the time complexity of IPOG (which does not support constraint checking) is in $O(v^{t+1} \times n^{t-1} \times \log n)$. It is argued that the time complexity of IPOG is dominated by the horizontal extension and the derivation is based on four pillars. First, it takes $O(1)$ to determine whether a t-sized schema is already covered by an existing schema. Second, it takes $O(n^{t-1})$ to determine the number of combinations covered by an existing schema. Third, for v values, it takes $O(v \times n^{t-1})$ to determine the number of yet uncovered t-sized schemata that are covered by an existing schema extended with a new value. Fourth, the number of test inputs selected by algorithms like IPOG is in $O(v^t \times \log n)$ [Coh+97]; [KKL13]. Therefore, the overall time complexity is in $O(v \times n^{t-1} \times v^t \times \log n)$ which can be shortened to $O(v^{t+1} \times n^{t-1} \times \log n)$.

In other words, the main operation this analysis is based on is the operation to find the new value which covers most of the yet uncovered t-sized schemata when combining the new value with an existing schema, i.e. $O(v \times n^{t-1})$.

The time complexity of IPOG-C comprises the time complexity of IPOG plus the time complexity of the additional constraint checking. With IPOG-C, constraint checking is involved every time a schema is extended with a new value to count the number of newly covered t-sized schemata. CSPs are commonly solved by *searching* for a solution, for instance via *backtracking* [RN10]. In the worst-case, backtracking has to check each possible schema in the solution space and the number of checks is in $O(v^n)$. Assuming each check can be performed in constant time, the time complexity of the main operation is in $O(v^n \times v \times n^{t-1})$.

The resulting time complexity of IPOG-C is in $O(v^n \times v \times n^{t-1} \times v^t \times \log n)$ which can be shortened as follows.

$$O(v^{t+1+n} \times n^{t-1} \times \log n)$$

In the following, we discuss how the worst-case time complexity of IPOG-C can be transferred to the IPOG-NEG$\langle *, * \rangle$ family of test selection strategies. Since the IPOG-NEG$\langle *, * \rangle$ test selection strategies are based on IPOC-C, they only differ in the way how the interactions of the initial test suite are computed and how the follow-up interactions between the initial test suite and the remaining parameters are computed.

Therefore, we discuss how the time complexity with the focus on horizontal extension changes for the different IPOG-NEG$\langle *, * \rangle$ test selection strategies.

The value of the first variable a of IPOG-NEG$\langle a, * \rangle$ determines the size of the initial test suite. For IPOG-NEG$\langle \forall, * \rangle$, the value of variable a is equal to the number of parameters of error-constraint c_i, i.e. $a = \texttt{Indices}(c_i)$. For IPOG-NEG$\langle \exists, * \rangle$, the initial test suite consists of exactly one schema.

The value of the second variable b of IPOG-NEG$\langle *, b \rangle$ determines the complexity of the horizontal extension because the existing schemata are extended to cover all b-sized schemata computed for the new parameter.. For $b = 0$, the horizontal extension as well as the horizontal extension can be skipped. Although for technical reasons, the horizontal extension adds "do not care" values to extend the initial schemata to cover n parameters. But the main operation of counting yet uncovered schemata is not required.

For the three test selection strategies IPOG-NEG$\langle \forall, b \rangle$, $\langle a, b \rangle$, and $\langle \exists, b \rangle$ with b-wise robustness interactions ($b > 0$), the time complexity is similar to the time complexity of IPOG-C. Although, the size of schemata that must be covered and consequently the size of the final test suite are determined by b instead of t.

$$O(v^{b+1+n} \times n^{b-1} \times \log n)$$

For the three test selection strategies IPOG-NEG$\langle \forall, 0 \rangle$, $\langle a, 0 \rangle$, and $\langle \exists, 0 \rangle$ with no robustness interactions ($b = 0$), the time complexity of horizontal extension would be linear without constraint checking. The time complexity growths with the number of parameters n instead of $v \times n^{t-1}$ because during horizontal extension only "do not care" values are stored. But because of the constraint checking, the time complexity is not linear, i.e. $v^n \times n$.

Further on, the size of the test suite is not in $O(v^t \times \log n)$ but in $O(v^a)$ since it is determined by the initial test suite. For IPOG-NEG$\langle \exists, 0 \rangle$, the test suite size is more precisely one and in $O(1)$. Therefore, the time complexity of IPOG-NEG$\langle \forall, 0 \rangle$ and IPOG-NEG$\langle a, 0 \rangle$ is as follows.

$$O(v^n \times n \times v^a)$$

While for IPOG-NEG$\langle \exists, 0 \rangle$, the time complexity is in $O(v^n \times n \times 1)$.

The complexity of IPOG-NEG$\langle *, * \rangle$ test selection strategies are individual for one particular error-constraint. Therefore, IPOG-C should also be compared with ROBUSTA that selects valid test inputs and strong invalid test inputs for all error-constraints.

To take the additional indirection of ROBUSTA as the dominant factor into account, we first argue that the worst-case time complexities of all IPOG-NEG$\langle *, * \rangle$ test selection strategies are in $O(v^{b+1+n} \times n^{b-1} \times \log n)$ because the IPOG-NEG$\langle *, * \rangle$ test selection strategies with robustness interactions clearly subsume the ones with no robustness interactions. This results in the following time complexity of ROBUSTA.

$$O(|C^{err}| \times v^{b+1+n} \times n^{b-1} \times \log n)$$

Again, the time complexity of IPOG-C is as follows.

$$O(v^{t+1+n} \times n^{t-1} \times \log n)$$

A comparison of IPOG-C and ROBUSTA is difficult because different values of testing strength t and robustness interaction degree b are used to solve the same test problem. In general, it can

be seen that both test selection strategies grow exponentially in terms of t and b. Although RO-BUSTA relies on error-constraints, increasing the number of error-constraints results only in a linear growth of the time complexity.

While a faster selection of test inputs is preferable, longer times are usually acceptable since the test inputs only have to be selected once. Therefore we empirically compare the strategies in the following and focus on the number of executed test inputs and the number of detected faults.

13.2. Experimental Evaluation

13.2.1. Motivation

To assess whether a CRT test method was necessary in the first place, we conducted a controlled experiment (Section 5.1, Page 74 and following) to evaluate how effective CT is in the presence of invalid input masking.

To briefly summarize the findings, invalid input masking negatively affects the effectiveness of the CT test methods and the effectiveness deteriorates with an increasing number of parameters involved in EH. In order to mitigate the negative effects of invalid input masking, higher testing strengths and larger test suites are required. Although, increasing the testing strengths is not always sufficient because testing strengths beyond $t \geq 6$ are generally considered impractical.

Based on these findings, the CRT test method is motivated and designed to avoid invalid input masking based on separating valid and strong invalid test inputs.

Consequently, the objective of this evaluation is to compare the proposed CRT test method with the CT test method. In particular, we aim to determine scenarios in which CRT and CT perform equally and to determine scenarios when the negative effects of invalid input masking cause CRT to perform better than CT.

Therefore, we conduct another controlled experiment where we select test inputs using RO-BUSTA with the IPOG-NEG$\langle \forall, b \rangle$ test selection strategy. Further on, we use the IPOG-C test selection strategy as a representative test selection strategy of the CT test method and compare its results with the results of ROBUSTA.

The IPOG-C test selection strategy is chosen over the NIST Covering Array Tables that we used in the first controlled experiment because it is more flexible in terms of number of parameters and values.

The IPOG-NEG$\langle \forall, b \rangle$ test selection strategy is chosen because it subsumes the IPOG-NEG$\langle *, * \rangle$ family of test selection strategies. Thereby, it promises the highest effectiveness and also requires the largest test suites among the IPOG-NEG$\langle \forall, b \rangle$ family of test selection strategies. When IPOG-C is more effective than IPOG-NEG$\langle \forall, b \rangle$, we know that IPOG-C is also more effective than the other IPOG-NEG$\langle *, * \rangle$ test selection strategies. In contrast, when IPOG-C is less efficient than IPOG-NEG$\langle \forall, b \rangle$, we know that IPOG-C is also less efficient than the other IPOG-NEG$\langle *, * \rangle$ test selection strategies.

13.2.2. Experiment Design

A common way for evaluating and comparing different test methods is to measure the number of detected faults [Gri+06]. We measure it by means of fault detection effectiveness (FDE) [Pet15] which allows us to compare test methods in terms of their effectiveness and efficiency.

$$\text{FDE}(T, SC) = \frac{\text{no. of faults detected by } T}{\text{no. of all faults in } TS}$$

Please note in contrast to the first controlled experiment, we place several faults into one test scenario. Therefore, the FDE is defined slightly different as the ratio of the number of faults detected by test suite T and the number of faults in test scenario SC.

Further on, we consider the same SUT characteristics that affect the FDE of test suites which are again discussed below.

Number of Parameters　　The more parameters a SUT has, the larger the exhaustive input space is. Thus, a smaller proportion of the input space is covered by a test suite that satisfies t-wise coverage. For n parameters with v values, the exhaustive input space is of size v^n while the size of a t-wise test suite is roughly $v^t \log n$ [Coh+97]; [KKL13].

Number of Values　　With an increasing number of values, the exhaustive input space of a SUT increases as well and a smaller proportion is covered by a t-wise test suite.

Degree of Invalid Input Masking　　The degree of invalid input masking is determined by the number of parameters involved in an exception condition and by the number of values per parameter that are involved in an exception condition.

For instance, a SUT with three parameters A, B and C each with two values has an exhaustive input space of size $2^3 = 8$. An exception condition $A = 1$ would initiate EH for four out of eight possible inputs. Therefore, a 1-factor ordinary fault $C = 2$ covered by four possible inputs would only be reached by two test inputs that do not contain $A = 1$.

If the three parameters consisted of three values each, the fault would be covered by nine possible test inputs and only three of them would lead to EH. Thus, the reachability increases from $2/4$ to $6/9$. If the exception condition of the original example with two values was extended from $A = 1$ to $A = 1 \land B = 1$, EH would only be initiated for two inputs and the reachability would increase from $2/4$ to $3/4$.

If the exception condition $A = 1 \land B = 1$ and the fault $A = 2 \land B = 2$ share the same parameters, EH and fault detection are mutually exclusive. Then, increasing the number of values does not change reachability.

Size of Faults　　The more parameter values are required to activate a fault, the harder it is to detect it. The 1-factor fault $C = 2$ is covered by four out of eight possible test inputs. In contrast, a 2-factor fault $B = 2 \land C = 2$ is only covered by two out of eight possible test inputs.

13.2.3. Experiment Setup

There are three problems with experimentation to evaluate testing approaches [Off+01]: Choosing a representative set of SUTs, choosing a representative set of faults, and choosing a representative set of tests.

To reach the faults, it is important for the IPM or RIPM to contain all necessary parameters and values. The reached faults must propagate to failures and the test oracle must reveal the failures [LO17]. Otherwise, the comparison is less reliable.

For an experiment, it is important to control the influencing factors. Therefore, we create artificial test scenarios to (1) completely control the size of the input space, (2) completely control the completeness and correctness of the IPMs and RIPMs, (3) completely control the number

and characteristics of faults, and (4) completely control reachability, infection and propagation of faults to failures.

The test scenarios are described in terms of the aforementioned characteristics. Irrelevant schemata are not considered in this experiment. Since they were excluded by both approaches, we assume that it would only complicate the experiment but not affect the results.

For a given test scenario, one test suite is selected by `IPOG-C` using an IPM and additional test suites are selected by `ROBUSTA` with different `IPOG-NEG`⟨∗,∗⟩ test selection strategies using a RIPM. The `run` method is executed for all test inputs and the FDE is based on the faults detected by each test suite.

```
 1  void run(int a1, ..., a5, boolean b1, ..., b5){
 2    if(a1 == 1) { // exit without failure
 3    } else if(a2 == 1) { // exit without failure
 4    } else if(a3 == 1) { // exit without failure
 5    } else if(a4 == 1) { fail("f1");
 6    } else if(a5 == 1) { if(b1) { fail("f2"); }
 7    } else {
 8      if(a3 == 2 && a4 == 2) {
 9        fail("f3");
10      }
11    }
12  }
```

Listing 13.1: Illustration of a Test Scenario Implementation

For further illustration, Listing 13.1 (Page 221) depicts an implementation of a test scenario. Suppose we have five input parameters with five values each implemented as integer variables, five binary configuration parameters implemented as boolean variables, and five exception conditions consisting of one value each. The separation of normal and exception control-flows is realized using `if` statements. The exception control-flows and different exception conditions are implemented by `if` and `else if` statements whereas the normal control-flow resides in the `else` statement. A fault is activated when the SUT is stimulated with the corresponding schema and when the location of the fault is reached which is modeled by `fail()`.

The illustration contains a 2-factor fault in the normal control-flow `f3`, a $(1, 0)$-factor configuration-independent robustness fault `f1` and a $(1, 1)$-factor configuration-dependent robustness fault `f2` which is only reachable when configuration `b1` is enabled.

The error-constraints are ignored by `IPOG-C` but considered by `ROBUSTA` to separate valid from strong invalid test inputs. A corresponding IPM is depicted in Listing 13.1 and a corresponding RIPM is depicted in Listing 13.2 (Page 222).

13.2.4. Experiment Scenarios

In this experiment, we use two *base* scenarios. To observe their impact on the FDE, they are extended according to the aforementioned characteristics.

```
                                        Parameters:
                                          p₁ : a₁        V₁ = {1, 2, 3, 4, 5}
                                          p₂ : a₂        V₂ = {1, 2, 3, 4, 5}
                                          p₄ : a₂        V₃ = {1, 2, 3, 4, 5}
                                          p₃ : a₂        V₄ = {1, 2, 3, 4, 5}
                                          p₅ : a₅        V₅ = {1, 2, 3, 4, 5}
          Parameters:                     p₆ : b₁        V₆ = {1, 2}
            p₁ : a₁        V₁ = {1, 2, 3, 4, 5}         p₇ : b₂        V₇ = {1, 2}
            p₂ : a₂        V₂ = {1, 2, 3, 4, 5}         p₈ : b₃        V₈ = {1, 2}
            p₄ : a₂        V₃ = {1, 2, 3, 4, 5}         p₉ : b₄        V₉ = {1, 2}
            p₃ : a₂        V₄ = {1, 2, 3, 4, 5}         p₁₀ : b₅       V₁₀ = {1, 2}
            p₅ : a₅        V₅ = {1, 2, 3, 4, 5}       Error-Constraints:
            p₆ : b₁        V₆ = {1, 2}                 c₁ : a₁ ≠ 1
            p₇ : b₂        V₇ = {1, 2}                 c₂ : a₂ ≠ 1
            p₈ : b₃        V₈ = {1, 2}                 c₃ : a₃ ≠ 1
            p₉ : b₄        V₉ = {1, 2}                 c₄ : a₄ ≠ 1
            p₁₀ : b₅       V₁₀ = {1, 2}                c₅ : a₅ ≠ 1
```

Left block — **Parameters:**

$p_1 : a_1 \quad V_1 = \{1, 2, 3, 4, 5\}$
$p_2 : a_2 \quad V_2 = \{1, 2, 3, 4, 5\}$
$p_4 : a_2 \quad V_3 = \{1, 2, 3, 4, 5\}$
$p_3 : a_2 \quad V_4 = \{1, 2, 3, 4, 5\}$
$p_5 : a_5 \quad V_5 = \{1, 2, 3, 4, 5\}$
$p_6 : b_1 \quad V_6 = \{1, 2\}$
$p_7 : b_2 \quad V_7 = \{1, 2\}$
$p_8 : b_3 \quad V_8 = \{1, 2\}$
$p_9 : b_4 \quad V_9 = \{1, 2\}$
$p_{10} : b_5 \quad V_{10} = \{1, 2\}$

Right block — **Parameters:**

$p_1 : a_1 \quad V_1 = \{1, 2, 3, 4, 5\}$
$p_2 : a_2 \quad V_2 = \{1, 2, 3, 4, 5\}$
$p_4 : a_2 \quad V_3 = \{1, 2, 3, 4, 5\}$
$p_3 : a_2 \quad V_4 = \{1, 2, 3, 4, 5\}$
$p_5 : a_5 \quad V_5 = \{1, 2, 3, 4, 5\}$
$p_6 : b_1 \quad V_6 = \{1, 2\}$
$p_7 : b_2 \quad V_7 = \{1, 2\}$
$p_8 : b_3 \quad V_8 = \{1, 2\}$
$p_9 : b_4 \quad V_9 = \{1, 2\}$
$p_{10} : b_5 \quad V_{10} = \{1, 2\}$

Error-Constraints:

$c_1 : a_1 \neq 1$
$c_2 : a_2 \neq 1$
$c_3 : a_3 \neq 1$
$c_4 : a_4 \neq 1$
$c_5 : a_5 \neq 1$

Figure 13.1.: IPM for the Test Scenario   Figure 13.2.: RIPM for the Test Scenario

Base scenario 1 consists of five input parameters and five configuration parameters. It includes five exception conditions of which each condition is unary and checks a separate input parameter. The faults are activated by values of the same five input parameters. Thus, there is a mutual exclusion between exception detection and fault detection.

Base scenario 2 consists of ten input parameters and five configuration parameters. It also includes the same five exception conditions of which each condition is unary and checks a separate input parameter. However, the included faults are triggered by values of the other five input parameters that are not checked by exception conditions. Thereby, we can observe whether the relationship between parameters involved in exception detection and parameters involved in t-factor faults has an impact on the FDE.

Both scenarios have a fixed set of t-factor ordinary faults with factors of one to four because these are the most common factors identified in empirical studies [KKL16]. In addition, they also contain a fixed set of (a, b)-factor robustness faults. They have to be defined relative to the exception conditions. A $(a, 0)$-factor robustness fault is always reached when the corresponding exception condition is satisfied. In contrast, a $(a, 2)$-factor robustness fault is only reached when the exception condition is satisfied and the fault-activating configuration is activated. The robustness interaction degrees b range from zero to three.

The base scenarios are then extended such that (a) the number of values per parameter increases and (b) the number of parameters increases. Further scenarios are then derived from each extended scenario by increasing the number of parameters involved in the exception conditions. The scenarios start with five unary conditions of which each checks a separate parameter. The conditions are extended to check two values, e.g. $a_1 = 1$ is extended to $a_1 = 1 \land a_2 = 2$, three and four values, e.g. $a_1 = 1 \land a_2 = 2 \land a_3 = 3 \land a_4 = 4$.

Test Scenarios	Descriptions	Parameters & Values	t-factor Ordinary Faults	(a,b)-factor Robustness Faults	Exception Conditions
1-1-1	**Base Scenario 1**			$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
1-1-2	* Increase Size of Conditions	$2^5 5^5$	$1^5 2^5 3^5 4^5$	$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
1-1-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
1-1-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
1-2-1	**Increase No. of Values**			$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
1-2-2	* Increase Size of Conditions	$3^5 7^5$	$1^5 2^5 3^5 4^5$	$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
1-2-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
1-2-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
1-3-1	**Increase No. of Parameters**			$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
1-3-2	* Increase Size of Conditions	$2^{10} 5^{10}$	$1^5 2^5 3^5 4^5$	$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
1-3-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
1-3-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
1-4-1	**Increase No. of Param. & Values**			$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
1-4-2	* Increase Size of Conditions	$3^{10} 7^{10}$	$1^5 2^5 3^5 4^5$	$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
1-4-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
1-4-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
2-1-1	**Base Scenario 2**			$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
2-1-2	* Increase Size of Conditions	$2^{10} 5^{10}$	$1^5 2^5 3^5 4^5$	$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
2-1-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
2-1-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5
2-2-1	**Increase No. of Values**			$(1,0)^5(1,1)^5(1,2)^5(1,3)^5$	1^5
2-2-2	* Increase Size of Conditions	$3^{10} 7^{10}$	$1^5 2^5 3^5 4^5$	$(2,0)^5(2,1)^5(2,2)^5(2,3)^5$	2^5
2-2-3	* Increase Size of Conditions			$(3,0)^5(3,1)^5(3,2)^5(3,3)^5$	3^5
2-2-4	* Increase Size of Conditions			$(4,0)^5(4,1)^5(4,2)^5(4,3)^5$	4^5

Table 13.3.: Test Scenarios used in the Evaluation

To reduce the effect of accidentally discovered faults, e.g. a 3-factor fault is detected by a 2-wise test suite, and to break the symmetry of the SUT and IPM or RIPM, i.e. the faults and the parameter values are ordered the same way, the parameters and values are randomly shuffled and the execution is repeated ten times.

Table 13.3 (Page 223) depicts all test scenarios used in the evaluation. It uses an exponential notation where x^y refers to y parameters consisting of x values, y t-factor ordinary faults of factor x, and y exception conditions with x parameters. For $(a, b)^y$-factor robustness faults, a refers to the number of involved input parameters that form an invalid schema which is always similar to the parameters involved in exception conditions. b refers to the number of configuration parameters necessary to make the fault reachable and y depicts the number of faults.

13.2.5. Results and Discussion

To compare `IPOG-C` and `ROBUSTA`, test inputs are selected with different testing strengths. For `IPOG-C`, the testing strengths range from one to five. For `ROBUSTA`, the testing strengths range from `1-0` to `4-3` where the first number represents the testing strength t used to select valid test inputs and the second number represents the robustness interaction degree b.

All 24 scenarios are evaluated by `IPOG-C` with five different testing strengths and by `RO-BUSTA` with 16 different testing strengths. Each triple of test scenario, test selection strategy and testing strength is computed ten times with randomly shuffled parameters and values. A subset of the resulting 5040 computations is listed in Table 13.4 (Page 224). All data is available in

Strategy	IPOG-C						ROBUSTA											
Strength	3		4		5		3-2		3-3		4-0		4-1		4-2		4-3	
Scenario	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.
1-1-1	199	0.88	923	0.99	3434	1.00	220	0.93	544	0.95	402	0.74	417	0.86	513	0.98	837	1
1-1-2	199	0.84	923	0.99	3433	1.00	344	0.94	814	0.95	845	0.74	865	0.86	989	0.99	1460	1
1-1-3	199	0.74	923	0.92	3433	0.99	335	0.93	508	0.95	916	0.74	936	0.86	1043	0.98	1216	1
1-1-4	199	0.55	923	0.76	3433	0.90	255	0.93	375	0.95	923	0.74	943	0.86	973	0.98	1092	1
1-2-1	552	0.86	3619	0.99	19428	1.00	603	0.89	1873	0.93	2041	0.69	2066	0.82	2278	0.96	3548	1
1-2-2	552	0.78	3619	0.97	19437	1.00	827	0.88	2326	0.92	3462	0.69	3493	0.82	3742	0.96	5242	1
1-2-3	552	0.63	3619	0.85	19437	0.97	809	0.88	1505	0.92	3617	0.69	3647	0.82	3867	0.97	4563	1
1-2-4	552	0.45	3619	0.69	19440	0.83	676	0.88	1044	0.92	3615	0.69	3645	0.82	3735	0.97	4103	1
1-3-1	324	0.91	1953	1.00	10737	1.00	473	0.96	1493	0.97	1367	0.72	1387	0.84	1580	0.99	2600	1
1-3-2	324	0.88	1949	1.00	10738	1.00	559	0.96	1757	0.96	1950	0.72	1970	0.84	2179	0.99	3378	1
1-3-3	324	0.77	1952	0.96	10740	1.00	547	0.95	1651	0.96	1962	0.72	1982	0.84	2180	0.99	3284	1
1-3-4	324	0.58	1954	0.84	10740	0.94	533	0.93	1515	0.95	1957	0.72	1977	0.84	2162	0.98	3144	1
1-4-1	881	0.90	7572	1.00	58759	1.00	1162	0.93	4379	0.96	5895	0.69	5925	0.81	6308	0.97	9525	1
1-4-2	881	0.82	7553	0.99	58723	1.00	1319	0.93	4839	0.95	7536	0.69	7567	0.81	7969	0.98	11490	1
1-4-3	881	0.67	7571	0.89	58724	0.99	1300	0.92	4579	0.94	7589	0.69	7619	0.81	8004	0.97	11283	1
1-4-4	881	0.50	7565	0.74	58715	0.90	1279	0.92	4272	0.94	7578	0.69	7608	0.81	7971	0.98	10964	1
2-1-1	324	0.85	1949	0.98	10741	1.00	473	0.94	1493	0.95	1369	0.72	1389	0.84	1582	0.99	2602	1
2-1-2	324	0.87	1950	1.00	10747	1.00	559	0.96	1757	0.96	1945	0.72	1965	0.84	2174	0.99	3373	1
2-1-3	324	0.78	1952	0.94	10747	0.99	547	0.94	1651	0.96	1966	0.72	1986	0.84	2184	0.99	3288	1
2-1-4	324	0.58	1949	0.82	10741	0.96	533	0.94	1515	0.96	1956	0.72	1976	0.84	2160	0.98	3143	1
2-2-1	881	0.85	7557	1.00	58725	1.00	1162	0.91	4378	0.94	5899	0.69	5929	0.81	6312	0.97	9528	1
2-2-2	881	0.80	7563	0.98	58728	1.00	1319	0.92	4839	0.93	7543	0.69	7573	0.81	7976	0.98	11497	1
2-2-3	882	0.66	7561	0.89	58768	0.99	1300	0.92	4579	0.94	7587	0.69	7617	0.81	8001	0.97	11280	1
2-2-4	882	0.49	7562	0.73	58715	0.91	1279	0.91	4272	0.93	7561	0.69	7591	0.81	7953	0.98	10946	1

Table 13.4.: Fault Detection Effectiveness and Test Suite Sizes for the Test Scenarios (Excerpt)

Appendix B (Page 335 and following). The presented sizes of test suites and computed degrees of FDE are average numbers of the ten randomized executions.

ROBUSTA with a testing strength of 4-3 activates all faults. The result can be expected because the seeded faults are at most 4-factor ordinary faults and $(4, 3)$-factor robustness faults. But it also demonstrates the correctness of the test adequacy criteria and test selection strategies. Lower testing strength are not sufficient. With a lower positive testing strength like 3-3, not all 4-factor ordinary faults in the normal control-flow are activated. With a lower negative testing strength like 4-2, not all $(4, 3)$-factor robustness faults are activated.

In comparison, IPOG-C with $t \leq 5$ does not activate all faults in all test scenarios. For the test scenarios with unary exception conditions (1-1-1, 1-2-1, 1-3-1, ...), IPOG-C with a testing strength of four activates almost all faults. The small deviations in FDE, e.g. 0.98 and 0.99 for test scenarios 1-1-1, 1-2-1, and 2-1-1, can be explained by invalid input masking. It performs as well as ROBUSTA if you compare the sizes of test suites. If you consider the additional effort to model error-constraints, IPOG-C is actually preferable because it requires less work than ROBUSTA.

Test scenarios with exception conditions that consist of two or more parameters require higher testing strengths. For a given testing strength, the FDE of IPOG-C decreases when the number of parameters involved in exception detection increases because a $(1, 3)$-factor robustness fault becomes a $(4, 3)$-factor robustness fault. As an example, the FDE decreases for a testing strength of four from an average of 0.993 (39.7 of 40 faults are activated) for scenarios with unary conditions to an average of 0.763 (30.5 of 40 faults are activated) for scenarios with 4-wise conditions. On average, the FDE decreases by 0.22 percentage points which equates to roughly 9 faults that

remain undetected.

This phenomenon can be explained by considering (a, b)-factor robustness faults as a special kind of t-factor faults (See Section 5.4, Page 109). Again, a (a, b)-factor robustness fault encompasses of a parameters that are involved in the exception sub-condition and b other parameters that are involved in the configuration sub-condition which is also necessary to activate the fault. When describing the same fault as a t-factor fault, the exception and configuration sub-conditions are summarized as the activation sub-condition. Since a t-factor fault does not consider any semantic information, an additional effective prevention sub-condition must be satisfied to prevent any invalid input masking. As a consequence, a $(4, 3)$-factor robustness fault translates into a t-factor fault of at least $t \geq 4 + 3$. Depending on the parameters involved in the effective prevention sub-condition, even more parameters may be involved. To reliably detect a $(4, 3)$-factor robustness fault with `IPOG-C`, a testing strength of at least $t = 7$ would be necessary.

Since the test scenarios with unary exception conditions (`1-1-1`, `1-2-1`, `1-3-1`, ...) contain at most 4-factor ordinary faults in the normal control-flow and at most $(1, 3)$-factor robustness faults, test suites that satisfy 4-wise coverage detect almost all of them.

In that sense, test suites that satisfy 5-wise coverage are at least required for test scenarios with binary exception conditions because $(2, 3)$-factor robustness faults must be activated. But even though a 5-wise test suite improves the FDE, the time required for selection as well as the size of the resulting test suite grow strongly. For every test scenario, `ROBUSTA` with `IPOG-NEG⟨∀, b⟩` produces a smaller test suite which is at least as effective as (or better than) the test suite selected by `IPOG-C` with a testing strength of $t = 5$.

This finding can be transferred to lower strengths as well. For instance, `IPOG-C` with a testing strength of $t = 3$ generates a test suite that detects almost all t-factor faults with $t \leq 3$ and almost all (a, b)-factor robustness faults with $a + b \leq 3$. However, `IPOG-C` with a higher testing strength is required for (a, b)-factor robustness faults with $a + b > 3$. Then, `ROBUSTA` produces a result with fewer test inputs which is at least as effective as (or better than) `IPOG-C`.

Therefore, increasing the testing strength of `IPOG-C` is not always the best option. The results indicate that `IPOG-C` with a testing strength of t is as effective as `ROBUSTA` as long as the (a, b)-factor robustness faults can be considered a special kind of t-factor faults. When (a, b)-factor robustness faults require a higher strength for `IPOG-C`, i.e. $a + b > t_1$, `ROBUSTA` requires fewer test inputs and is more effective in terms of fault detection.

To summarize this experiment, the results indicate that `IPOG-C` with a testing strength of t is effective in detecting - t_1-factor faults with $t_1 \leq t$ and all (a, b)-factor robustness faults with $a + b \leq t$. In contrast, `ROBUSTA` has the same or higher FDE and comparable or smaller test suite sizes but requires additional work to model error-constraints. But for higher testing strength, `ROBUSTA` is favorable because the test suites are much smaller and provide the same or higher FDE.

The CRT test method with `ROBUSTA` is preferable over the CT test method for SUTs with EH that involves many parameters and complex exception conditions. For less complex exception conditions, the CRT test method performs as well as the CT test method. But additional effort to model error-constraints is required.

In the following section, a case study is conducted to assess CRT and CT using real-world test scenarios. Thereby, we also demonstrate how to reuse error-constraints as a test oracle which

further relativizes the effort to model them.

13.2.6. Threats to Validity

We compared the FDE of ROBUSTA with IPOG-NEG$\langle \forall, b \rangle$ and IPOG-C. Since the comparison is based on actual test input selection and execution, our results might depend on the implementation of the test selection strategies as well as the implementation and design of the test scenarios.

To ensure unbiased implementations of the test selection strategies, we follow the suggestions for efficient implementations by KLEINE & SIMOS [KS18].

The test scenarios used for testing are artificial and do not necessarily represent real-world scenarios. However, we explicitly stated the considered characteristics and measured the implications in the controlled experiment. Therefore, the findings can be used to decide between the two alternatives in real-world scenarios.

To prevent faults being discovered by accident or because of symmetries between the IPM or RIPMs and the parameter values of the SUT, the scenarios are randomized and executed ten times. The presented numbers are average numbers.

To allow and support reproducibility of the experiment, all data is available in Appendix B (Page 335 and following). Further on, the source code of the test selection strategies is published as well (Section 11.3, Page 205 and following).

13.3. Case Study-based Evaluation

13.3.1. Motivation

In the previous section, a controlled experiment is conducted to assess the FDEs of CRT and CT. The results show CRT to be as good as CT for SUTs with little EH and CRT to be more effective and efficient than CT for SUTs with complex EH. However, the assessment is solely based on artificial test scenarios. Since artificial test scenarios allow to design test scenarios with desired properties, it remains unclear if these desired properties are equally important in real-world test scenarios.

Therefore, we further compare CRT with CT guided by the following two research objectives.

Research Objective 1 Is the CRT test method applicable in real-world test scenarios?

Research Objective 2 How does the CRT test method compare with CT in real-world test scenarios?

Furthermore, we hypothesize that error-constraints cannot only be used to guide the selection of test inputs and to separate valid from strong invalid test inputs. We hypothesize that error-constraints can further be used as a test oracle to decide whether or not a SUT correctly follows the standard or exception behavior. This raises the following additional research objective.

Research Objective 3 How effective are test oracles that are based on error-constraints?

To demonstrate the applicability of CRT in real-world test scenarios and to compare CRT with CT in real-world test scenarios, we conduct a case study. According to KITCHENHAM et al. [KPP95], a case study helps to evaluate the benefits of methods and tools in industrial settings and a case study provides a cost-effective way to ensure that process changes achieve the desired results. When applied to compare methods and tools, a case study is of explanatory nature "seeking an explanation of a situation or a problem" which "involves the testing of existing theories in confirmatory studies" [RH09].

As RUNESON & HÖST state, a case study "will never provide conclusions with statistical significance" [RH09]. However, it can provide sufficient information to help you judge if specific technologies will benefit your own organization or project" [KPP95]. Since a case study has by definition a higher degree of realism than a controlled experiment [RH09], a case study that compares CRT with CT can provide additional insights that complement and extend the findings of the previously conducted controlled experiment.

We again follow the guidelines for conducting and reporting case study research in software engineering [RH09]. The guidelines consist of five steps and suggest defining the objectives and planing the case study first. Second, procedures and protocols for data collection are defined. Third, the case study is *executed* and data is collected. Fourth, the collected data is analyzed and finally reported.

In the following subsection, we discuss the cases units under analysis as well as the data collection procedure. Afterwards, the results are presented and discussed.

13.3.2. Case Study Design

Case under Analysis

Because only a small number of cases are analyzed in a case study, the cases should be selected intentionally [RH09]. But unfortunately, when conducting case studies in industrial settings, cases cannot be chosen freely. In fact, cases are often selected based on availability and depends on the will of the industry partner and the availability of the industry partner's resources [RH09].

In this case study, we have access to a case which we can use to compare CRT with CT. The case is a software development project from an IT service provider of an insurance company. It is concerned with the validation of data according to a set of validation rules and with the storage or forwarding of data when it satisfies the validation rules. The case is from a project to develop a new software to manage the life-cycle of life insurance contracts. It is the same project which we analyzed in our previous case study (Section 5.3.2, Page 98 and following). While the previous case study had a project-wide scope, this case study only focuses on a sub-system that ensures applications for life insurances to be valid.

An application for a life insurance is passed to the sub-system in a hierarchical extensible markup language (XML) format and it is checked if the insurance application satisfies a set of validation rules. The order of the validation rules is predefined and the validation rules are iterated for each insurance application. Whenever a validation rule is not satisfied by an insurance application, a corresponding error code is returned and the remaining validation rules are skipped (*fail fast* principle). If all validation rules are satisfied, the sub-system returns SUC- CESS and the insurance application is further processed. Although, the further processing is out of scope for this case study.

The validation rules are specified in a domain-specific language (DSL). They are automatically transformed into source code that can be executed to verify an application. The automatic transformation does not require testing in this context as it is based on an off-the-shelf framework which is customized and extended. But to test the validation rules, no independent specification exists since the validation rules and the source code are semantically equivalent.

Therefore, we consider the current set of validation rules as correct and treat the current implementation as the specification. We use it as a specification to test earlier versions of the validation rules. By browsing the source code repository, we could identify 13 changes that have been made to the validation rules in order to correct them. Based on these 13 changes, 13 implementations can be reconstructed of which all deviate from the current set of validation rules and of which each reconstruction contains a fault. Each change is a previously-existing fault that is fixed in the current implementation. Since the software project is still under development, all 13 faults are fixed prior to release.

The case contains a set of 51 validation rules that are used to check incoming insurance applications. Each validation rule is specified in a DSL which requires the validation rule to be build as an implication that consists of two parts, i.e. `isApplicable(application)` $\Rightarrow$ `isValid(application)`. The first part, i.e. `isApplicable(application)`, determines whether the validation rule is applicable to the insurance application. Although, some validation rules are always applicable. If the validation rule is applicable, the insurance application must be valid, i.e. `isValid(application)`. If not, the validation rule is ignored.

Because details of the case are confidential, a generic example is given to provide further illustration of validation rules. The example depicts two validation rules to define maximum sums that can be insured depending on the permissions of the insurance agents. The first validation rule is applicable to all applications created by insurance agents with the highest level of permission. The second validation rule is applicable to all applications that are created by insurance agents with lower permission level.

The distinction between the two validation rules is made by the first part of the implication:

$$\textbf{Rule 1: } \texttt{isApplicable(application)}:$$
$$\texttt{application.agent.permission} = \texttt{highest\_level}$$
$$\textbf{Rule 2: } \texttt{isApplicable(application)}:$$
$$\texttt{application.agent.permission} \neq \texttt{highest\_level}$$

The second part of the implication is used to enforce the maximum insured sum. As an application may consist of several partial contracts, the individual insured sums of all partial contracts are collected first. Afterwards, it is checked whether the total sum exceeds the threshold. While the structure of both rule's $\texttt{isValid()}$ parts is the same, different values for the $\texttt{maximum\_insured\_sum}$ constant are used:

$$\texttt{isValid(application)}:$$
$$\texttt{total\_sum} = \sum_{\texttt{partial} \in \texttt{application}} \texttt{partial.insured\_sum}$$
$$\texttt{total\_sum} \leq \texttt{maximum\_insured\_sum}$$

This example shows that many parameters may be involved in a validation rule, that intermediate calculations may be required, and that intermediate calculations may be reused in different validation rules. Therefore, the validation rules should be tested thoroughly.

Since not all validation rules for all types of life insurances are modeled yet, we focus on risk-based life insurances which are already modeled. A subset of 31 validation rules is applicable for risk-based life insurances. The remaining subset of 20 validation rules are not applicable to risk-based life insurances and are therefore not further considered. All 13 faults can be detected by testing with applications for risk-based life insurances.

Because of the direct access to the validation rules, we cannot only test the case with its *fail fast* principle. But we can also iterate through all validation rules and collect all validation rules that are not satisfied by an insurance application (we refer to this as the *validate all* principle). Although this is a manipulation of the case, we use the *validate all* version besides the *fail fast* version to further assess the impact of invalid input masking.

Data Collection Procedure

In this case study, data collection refers to the measurement and calculation of metric values from modeled RIPMs and IPMs as well as from executed tests. The resources available from the software development project are not directly analyzed and compared. Instead, they are used

to model RIPMs and IPMs that represent variations of insurance applications and to reconstruct implementations for test execution.

In order to not consume too many resources of the industry partner, the RIPMs and IPMs are solely developed by us. The tests are also executed solely by us. However, the models and results are presented to the industry partner to receive feedback.

Based on RIPMs and IPMs, test inputs are selected using CT and CRT. Then, the test inputs are executed to assess the effectiveness of identifying faults in the 13 reconstructed implementations. To reduce the effect of accidental fault detection that is caused by ordering, the parameters and values of a test suite are randomly reordered and 20 different variants of each test suite are generated.

Below, the definition of FDE is repeated [GOA05]; [Pet15]. A test suite T is denoted as *failing* for a test scenario SC if at least one of the test inputs $\tau \in T$ detects the fault in SC and the test suite consequently fails.

$$\texttt{failing}(T, SC) = \left\{ \begin{array}{l} 1 \text{ if } \exists \tau \in T \text{ that fails for } SC \\ 0 \text{ otherwise} \end{array} \right.$$

Using the $\texttt{failing}$ function, FDE is defined as the ratio between the number of test suites $T \in T^*$ of a family that fail for a test scenario SC and the number of all test suites in a family T^* that is used to test SC. In this case study, a family of test suites contains 20 different variants. In other words, the FDE is based on 20 randomized variants of a test suite that all satisfy the same test adequacy criterion. They all are used to test the same test scenario.

$$\texttt{FDE}(T^*, SC) = \frac{\sum_{T \in T^*} \texttt{failing}(T, SC)}{|T^*|}$$

Further on, the AFDE denotes the average FDE over a family of test scenarios SC^*. Here, the family of test scenarios $SC^* = \{SC_1, ..., SC_{13}\}$ consists of the 13 reconstructed implementations. AFDE represents the average effectiveness of CRT and CT equally distributed over the 13 faults.

$$\texttt{AFDE}(T^*, SC^*) = \frac{\sum_{SC \in SC^*} \texttt{FDE}(T^*, SC)}{|SC^*|}$$

Based on the procedure and metrics described above, the case study is executed and its results are reported and discussed in the following.

13.3.3. Results and Discussion

Modeling of IPM and RIPM

The data collection procedure requires IPMs and RIPMs of the sub-system to produce varying insurance applications that can be used to exercise all 31 validation rules. Since the FDE and AFDE metrics highly depend on the quality of IPMs and RIPMs, a systematic modeling approach is necessary. Therefore, the IPM is modeled first and later extended with error-constraints to model a RIPM.

The IPM is modeled iteratively for one validation rule at a time. In each iteration, parameters and values are added to ensure that test inputs with the following three characteristics can be selected:

> (1) test inputs that are not applicable:
>
> $\neg$isApplicable(application)
>
> (2) test inputs that are applicable and valid:
>
> isApplicable(application) $\land$ isValid(application)
>
> (3) test inputs that are applicable but not valid:
>
> isApplicable(application) $\land \neg$isValid(application)

In addition, some exclusion-constraints are introduced to ensure syntactic correctness of selected test inputs. For instance, a test input should not cover information about a second insurance product when the insurance application only contains one sub-part. This cannot be described in the SUT's XML format and is therefore also not validated by the validation rules.

In testing, code coverage is an important metric to measure and describe the degree to which source code is *covered* by a test suite (cf. [CMH16]). The three cases of test inputs with different characteristics relate to function coverage, i.e. each validation rule has been called, and to branch coverage, i.e. each branch of each validation rule has been called. By ensuring that a test suite satisfies function and branch coverage of all validation rules, it can be assured that test inputs for all three cases are selectable.

The IPM is considered as complete once the IPM contains all parameters and values necessary to satisfy function and branch coverage. This is checked in every iteration by selection test inputs for the IPM with IPOG-C and by stimulating the SUT following the *validate all* principle.

For the RIPM, error-constraints are required for an additional separation of valid and strong invalid test inputs. Since we consider the current implementation of the sub-system as correct, SUCCESS is returned for each truly-valid test input while an error code is returned for each truly-strong invalid test input. The error-constraints are modeled iteratively and we add new or update existing error-constraints until the separation of valid and strong invalid test inputs is equivalent to the separation of truly-valid and truly-strong invalid test inputs. Therefore, we select valid and strong invalid test inputs using ROBUSTA and again test the sub-system following the *validate all* principle. The error-constraints are considered complete when (1) test suites selected by ROBUSTA satisfy function and branch coverage, (2) the separation of valid and strong invalid test inputs is correct, and (3) the RIPM is consistent and not over-constrained.

The complete IPM and RIPM are described below in exponential notation. For parameters and values, x^y refers to y parameters with x values. For exclusion- and error-constraints, x^y refers to y constraints with x parameters.

$$\text{Parameters \& Values: } 9^1 6^2 5^1 4^8 3^8 2^{12}$$

$$\text{Exclusion-Constraints: } 2^3$$

$$\text{Error-Constraints: } 5^2 3^6 2^8 1^{15}$$

Selecting and Executing Test Inputs

After modeling the IPM and RIPM, both models are used to select sets of test inputs. Since we compare CRT with CT, two different test selection strategies are used. ROBUSTA is used to select test inputs for the RIPM and IPOG-C is used to select test inputs for the IPM.

To compare the FDE and AFDE of CRT with CT, test suites that satisfy different test adequacy criteria are used. Therefore, we apply IPOG-C to select test suites that satisfy t-wise relevant coverage for $t \in \{1, ..., 5\}$. Furthermore, we apply ROBUSTA to select test suites that satisfy t-wise valid coverage with $t \in \{1, ..., 3\}$ and that satisfy variations of the $\langle *, * \rangle$-wise strong invalid coverage. All six variants of the $\langle *, * \rangle$-wise strong invalid coverage are used to select test inputs that cover all strong invalid schemata ($\forall$), some strong invalid schemata ($\exists$), and a-wise strong invalid schemata with $a \in \{1, 2, 3\}$. Since we do not expect configuration-dependent robustness faults, zero robustness interactions and a low degree of robustness interactions $b \in \{0, 1\}$ are considered.

To reduce the effect of accidental fault detection caused by ordering, the order of parameters and values is randomly reordered and 20 different variants are used to select test suites for each test adequacy criteria. Table 13.5 (Page 233) depicts the average sizes of test suites that satisfy the different test adequacy criteria. Since ROBUSTA encompasses two test adequacy criteria (t-wise valid coverage and $\langle *, * \rangle$-wise strong invalid coverage), the test suites are listed separately and combined.

The largest test suite is selected by IPOG-C which is required to satisfy t-wise relevant coverage with $t = 5$ (15023.70 test inputs). The second-largest test suite is also selected by IPOG-C to satisfy t-wise relevant coverage with $t = 4$ (2813.45 test inputs). The third-largest test suite is selected by ROBUSTA and satisfies t-wise valid coverage with $t = 3$ and $\langle \forall, 1 \rangle$-wise strong invalid coverage (2224.30 test inputs).

When comparing the test suite sizes of t-wise relevant coverage of IPOG-C with t-wise valid coverage of ROBUSTA, it can be seen that the error-constraints drastically reduce the number of valid schemata and consequently also the number of test inputs. Furthermore, it can be seen that the different options of all ($\forall$), some ($\exists$), or a-wise strong invalid schemata have a much smaller effect than the option for robustness interactions ($b = 0$ or $b = 1$). This is because strong invalid coverage with the $\langle \forall, * \rangle$ option is equivalent to the $\langle \exists, * \rangle$ option for all error-constraints that only specify one strong invalid schema. In addition, the $\langle a, * \rangle$ option is also equivalent to the $\langle \forall, * \rangle$ option for all error-constraints with a or fewer parameters.

After test input selection, the test suites are used to stimulate the SUT in 13 different versions. The current implementation is tested and no failing tests are expected to ensure no false positives. In addition, the current implementation is tested following the *validate all* principle to further illustrate the impact of invalid input masking. Moreover, the 13 reconstructed implementations of which each contains one injected faults are tested to determine which test suites are able to detect which faults.

Robustness Fault Characteristics

Now, the IPM and the RIPM are modeled, test suites that satisfy different test adequacy criteria are selected, and different versions of the SUT are tested. Before analyzing the results of the

Test Adequacy Criteria	t	a	b	Size	Test Adequacy Criteria	t	a	b	Size
t-wise relevant	1	-	-	9.00	t-wise valid coverage	1	∃	0	36.00
coverage	2	-	-	68.10	and $\langle *, * \rangle$-wise	1	∃	1	208.30
	3	-	-	480.10	strong invalid coverage	2	∃	0	77.30
	4	-	-	2813.45	(continued)	2	∃	1	249.60
	5	-	-	15023.70		3	∃	0	296.95
t-wise valid	1	-	-	7.00		3	∃	1	469.25
coverage	2	-	-	48.30		1	1	0	74.60
	3	-	-	267.95		1	1	1	446.10
$\langle *, * \rangle$-wise	-	∀	0	301.00		2	1	0	115.90
strong invalid coverage	-	∀	1	1956.35		2	1	1	487.40
	-	∃	0	29.00		3	1	0	335.55
	-	∃	1	201.30		3	1	1	707.05
	-	1	0	67.60		1	2	0	130.50
	-	1	1	439.10		1	2	1	800.80
	-	2	0	123.50		2	2	0	171.80
	-	2	1	793.80		2	2	1	842.10
	-	3	0	177.45		3	2	0	391.45
	-	3	1	1121.65		3	2	1	1061.75
t-wise valid coverage	1	∀	0	308.00		1	3	0	184.45
and $\langle *, * \rangle$-wise	1	∀	1	1963.35		1	3	1	1128.65
strong invalid coverage	2	∀	0	349.30		2	3	0	225.75
	2	∀	1	2004.65		2	3	1	1169.95
	3	∀	0	568.95		3	3	0	445.40
	3	∀	1	2224.30		3	3	1	1389.60

Table 13.5.: Sizes of Test Suites that satisfy different Test Adequacy Criteria

testing, i.e. the FDE values per test suite family, the 13 faults are discussed first.

All 13 faults can be detected by test suites that are based on the IPM or RIPM. Five faults can only be detected by invalid test inputs, while eight faults can be detected by both valid and invalid test inputs.

The faults can also be classified according to our classification of robustness fault characteristics (Section 5.2, Page 91 and following). We classify the five faults that can only be detected by invalid test inputs first. Two faults can be classified as *faults in exception propagation*. In order to reveal them, invalid test inputs must trigger EH which responds with an incorrect error code. Three faults can be classified as *faults in exception detection*. The exception conditions are too weak and do not detect invalid test inputs. Hence, the SUT does not reject the invalid test inputs and incorrectly proceeds with their processing.

The remaining eight faults that can all be detected by both valid and invalid test inputs. They are *faults in exception detection*. Four faults have exception conditions that are too strong and therefore incorrectly detect exception occurrences for valid test inputs. The other four faults have characteristics of being too weak and too strict at the same time because wrong parameters with similar characteristics are used in the exception condition. As a consequence, a minimal invalid schema may not violate the exception condition (too weak) while a valid schema may not satisfy the exception condition (too strong).

> **Running Example** Since concrete details of the specification and sub-system of the industry partner may not be published, we use the running example for illustration. Therefore, let's consider a validation rule of the example implementation (Listing 2.1, Page 14).
>
> ```
> 1    if(isInvalidNumberOfPackages(packages)) return "INV_PACKAGES";
> ```
>
> Let's assume this validation rule uses a wrong parameter in its exception condition, e.g. `Country` instead of `Packages`.
>
> ```
> 1    if(isInvalidNumberOfPackages(country)) return "INV_PACKAGES";
> ```
>
> This may cause minimal invalid schemata that cover an incorrect number of packages, e.g. `[Packages:123]`, not to be detected because the value for `Packages` is not checked. Furthermore, this may cause valid schemata that cover a correct country code, e.g. `[Country:UK]`, to be incorrectly detected. Therefore, this fault can be detected by both valid and invalid test inputs.
>
>   Although, there is another scenario in which this fault cannot be detected when both schemata are accidentally combined, e.g. `[Packages:123]` and `[Country:UK]` always appear together in the same invalid test inputs. Then, the exception behavior that is expected for the minimal invalid schema is accidentally triggered by the combined valid schema.

To reliably detect this type of fault with valid test inputs, it is necessary to simply test with valid schemata that violate the incorrect exception condition. To reliably detect this type of fault with invalid test inputs, it is important to combine the minimal invalid schema with different valid schemata to prevent an accidental combination of the minimal invalid schema that should trigger EH and the valid schema that incorrectly triggers the EH.

The remaining classes of robustness fault characteristics, i.e. fault in exception handler selection, fault in exception recovery, and fault in exception continuation, cannot appear in this case study for technical reasons.

Fault Detection Effectiveness

Table 13.6 (Page 243) lists the FDE values for test suites that satisfy different test adequacy criteria. For better readability, + is used to indicate an FDE value of 1.00. The faults Nos. 1 to 8 are the faults that can all be detected by both valid and invalid test inputs. The faults Nos. 9 to 13 are the faults that can only be detected by invalid test inputs.

Again, the FDE value is an average value for one test suite family with 20 different test suites that are created by randomizing the order of parameters and values before selecting test inputs. As an example, in the first row for fault no. 3, an FDE value of 0.75 means that 15 out of 20 test suites detected the fault at least once per test suite.

The t-wise relevant coverage is listed twice. To provide a reference, the test suites are additionally executed on SUT versions that follow the *validate all* principle. The *validate all* principle allows discussing validation rules and their FDE values individually without considering the invalid input masking effect. The FDE values show that $t = 3$ is sufficient to detect all

faults individually if no prior validation rule causes invalid input masking. It further shows that all parameters and values necessary to detect all faults are modeled in the IPM and RIPM.

With a testing strength of $t = 2$, t-wise relevant coverage is almost sufficient to detect all faults. Only fault no. 5 has a FDE value of 0.95 which means that 19 out of 20 test suites are able to detect the fault.

This result is in line with another observation. Overall, fault no. 5 is not reliably detected by 18 test suite families which is the highest count among all faults. Although fault no. 12 is also not reliably detected by 18 test suite families, the comparison is unfair because fault no. 12 cannot be detected by test suites for t-wise valid coverage. Therefore, fault no. 5 can be considered as the fault that is the toughest to detect.

The strong impact of the invalid input masking effect becomes obvious when comparing the results of t-wise relevant coverage for the *validate all* principle with t-wise relevant coverage for the *fail fast* principle. Following the *fail fast* principle, t-wise relevant coverage with $t = 3$ is no longer able to detect all faults reliably. Instead only for three faults, a FDE value of 1.00 is computed and two faults are never detected (FDE value of 0.00). In general, the FDE values increase when the testing strength t grows. But even with $t = 5$ (15023.70 test inputs), only 7 faults are detected reliably (FDE value of 1.00) and one fault is still never detected (FDE value of 0.00).

The CRT test adequacy criteria are characterized by avoiding the invalid input masking effect. Since all invalid schemata are excluded by t-wise valid coverage, the faults Nos. 9 to 13 cannot be detected. But for all other faults, t-wise valid coverage has higher FDE values for the same testing strength t when comparing it with t-wise relevant coverage. Because invalid input masking is avoided, a testing strength of $t = 2$ is sufficient to detect faults Nos. 1 to 8 reliably (FDE values of 1.00).

The family of $\langle *, * \rangle$-wise strong invalid coverage test adequacy criteria is able to detect faults Nos. 9 to 13 reliably since strong invalid schemata must be covered to satisfy them. It can be discussed in the two dimensions of robustness degree $(\forall, \exists, a)$ and robustness interaction degree $(b = 0, b > 0)$. Except for $\langle \exists, * \rangle$ and $\langle 1, 0 \rangle$, the faults Nos. 9 to 13 are detected by all test suites that satisfy $\langle *, * \rangle$-wise strong invalid coverage. In general, the $\langle \forall, * \rangle$ has the highest FDE values when comparing it with $\langle \exists, * \rangle$ and $\langle a, * \rangle$. With $\langle \forall, 0 \rangle$, 11 out of 13 faults can already be detected reliably and the two remaining faults have high FDE values of 0.9 and 0.8. The option $\langle \forall, * \rangle$ is directly followed by $\langle a, * \rangle$ with $a = 3$ and the FDE values decrease with decreasing a values. $\langle \exists, * \rangle$ performs the worst in this comparison

The effect of robustness interactions $(b > 0)$ is even higher than different robustness degree options. Except for $\langle \exists, 1 \rangle$ and $\langle 1, 1 \rangle$, test suites that cover robustness interactions detect almost all faults reliably. Three faults cannot be detected reliably with $\langle \exists, 1 \rangle$, while only 1 fault cannot be detected reliably with $\langle 1, 1 \rangle$.

Two faults (Nos. 7 and 8) require robustness interaction to be detected reliably. In fact, they are the only two faults that are never reliably detected without robustness interactions. These are two faults where the exception condition uses incorrect parameters. But also without robustness interactions, the FDE values are high (0.90, 0.80, and 0.75 in one case). In these cases, we could observe that the strong invalid schemata are accidentally combined with valid schemata that are able to detect the fault.

Moreover, four faults that have too strong exception conditions and that actually require valid test inputs to be detected are also reliably detected by $\langle \forall, * \rangle$, $\langle 2, 1 \rangle$, and $\langle 3, 1 \rangle$. We could observe that a strong invalid test input that is expected to violate the exception condition of the l-th validation rule is also expected to satisfy all prior validation rules from 1 to $l - 1$. Therefore, strong invalid test inputs can be considered as "partially-valid" test inputs that are able to accidentally detect faults that require valid schemata. This effect is strengthened by robustness interactions because more test inputs are selected and more interactions are covered by them.

Typically, t-wise valid coverage and $\langle *, * \rangle$-wise strong invalid coverage are combined by `ROBUSTA` and the FDE values show that test suites for both test adequacy criteria complement each other. Since valid and strong invalid test inputs are able to detect faults Nos. 1 to 8, the FDE values are complemented by the combination of both test suites. Although the FDE values of combined test suites are never worse than the FDE values of single test suites, the FDE values of combined test suites are also not the sum of the FDE values of single test suites. This is due to the fact that both test suites detect the same faults and the the FDE of both test suites overlaps.

For faults Nos. 9 to 13, the FDE values are not complemented by the combination of both test suites. This is because test suites that only satisfy t-wise valid coverage cannot detect these faults. Therefore, the FDE values of the combined test suites are the same as the FDE values of the test suites that satisfy $\langle *, * \rangle$-wise strong invalid coverage.

In order to detect all faults reliably, the $\langle *, * \rangle$-wise strong invalid coverage must be selected to detect faults Nos. 9 to 13 reliably. Further on, t-wise valid coverage and $\langle *, * \rangle$-wise strong invalid coverage must be selected to jointly detect faults Nos. 1 to 8 reliably.

Therefore, all combinations of $t = 2$ or $t = 3$ with $\langle \forall, * \rangle$, $\langle 1, 1 \rangle$, $\langle 2, * \rangle$, and $\langle 3, * \rangle$ detect all faults reliably. The options $\langle \exists, * \rangle$ and $\langle 1, 0 \rangle$ cannot be used because they do not reliably detect fault no. 12. Even though $t = 1$ is only sufficient to detect three of the first eight faults reliably, the combination with $\langle *, * \rangle$ improves the FDE. All faults can be detected reliably when combining $t = 1$ with $\langle \forall, * \rangle$, $\langle 2, 1 \rangle$, or $\langle 3, 1 \rangle$.

The discussion of the FDE shows which test adequacy criteria are appropriate to reliably detect different types of faults. In the following, we now discussion the average FDE over all 13 faults (AFDE).

Average Fault Detection Effectiveness

Because AFDE values are average values over a set of faults, it allows making general statements about the effectiveness of test adequacy criteria and it also allows making general statements about the efficiency of test adequacy criteria. First, we discuss the effectiveness in terms of AFDE values of different test adequacy criteria. Table 13.7 (Page 244) lists the AFDE values for test suites that satisfy different test adequacy criteria. Afterwards, we discuss the efficiency in terms of AFDE values in relation to test suite sizes as listed in Table 13.5 (Page 233).

The AFDE values reflect what we discussed before since they aggregate FDE values. Because of the invalid input masking effect, test suites that satisfy t-wise relevant coverage only reach an AFDE value of 0.62.

In direct comparison, test suites that satisfy t-wise valid coverage reach a maximum AFDE value of 0.62 as well. The same AFDE value can be reached because they prevent invalid input masking. However, the AFDE value cannot be further improved by increasing the testing

strength because faults Nos. 1 to 8 are already detected reliably and faults Nos. 9 to 13 cannot be detected by valid test inputs. Comparing the two test adequacy criteria for each testing strength individually shows that the AFDE value of t-wise valid coverage is always higher than the AFDE value of t-wise relevant coverage.

For different configurations of $\langle *, * \rangle$-wise strong invalid coverage, the lowest AFDE value is 0.68 which is always higher than the AFDE values of t-wise relevant and valid coverage. Furthermore, three (30%) different configurations have an AFDE value of 1 and therefore detect all faults reliably. Seven (70%) different configurations have an AFDE value of 0.92 or higher and therefore detect at least 12 of the 13 faults.

Overall, the combination of t-wise valid coverage and $\langle *, * \rangle$-wise strong invalid coverage performs the best. In the worst case, the AFDE value is 0.85 for the combination of $t = 1$ and $\langle \exists, 0 \rangle$. 18 different combinations have an AFDE value of 1.00 and detect all faults reliably. 27 different combinations have an AFDE value of 0.92 or higher and therefore detect at least 12 out of the 13 faults.

When putting the AFDE values in relation to test suite sizes, it can be noted that t-wise relevant coverage has the worst efficiency as it requires 15023.70 test inputs for an AFDE value of 0.62. In contrast, t-wise valid coverage only requires 48.30 test inputs for an AFDE value of 0.62. The best efficiency is offered by the combination of $t = 2$-wise valid coverage and $\langle 2, 0 \rangle$- wise strong invalid coverage which requires 171.80 test inputs for an AFDE value of 1.00. When using an AFDE value of 0.92 as a lower boundary (12 out of 13 faults), $\langle 2, 0 \rangle$-wise strong invalid coverage requires only 123.50 test inputs for an AFDE value of 0.94.

This discussion about efficiency is, of course, influenced by the characteristics of the 13 faults and cannot be generalized. But as more general statements, it can be observed that t-wise relevant coverage requires more test inputs to reach a similar AFDE value than t-wise valid coverage, $\langle *, * \rangle$-wise strong invalid coverage, or the combination of both. At the same time, the combination of t-wise valid coverage and $\langle *, * \rangle$-wise strong invalid coverage has an AFDE value of at least 0.85 while at most 2224.30 test inputs are used.

Therefore, we draw the conclusion that t-wise valid coverage, $\langle *, * \rangle$-wise strong invalid coverage, and the combination of both perform as well as or better than t-wise relevant coverage in terms of effectiveness and efficiency. Although the findings are only derived from one particular case. Therefore, we do not consider this to be true for all SUTs but for SUTs with many validation rules. This finding is also consistent with our experimental evaluation (Section 13.2, Page 219 and following).

Effectiveness of Error-Constraint-based Test Oracle

In addition to the IPM and RIPM, a test oracle is required to reveal failures that are caused by faults activated by the selected test inputs. Since the current implementation of the sub-system functions as the specification, it can also be used as a test oracle. Then, a failure is revealed whenever the output of a previous implementation deviates from the output of the current implementation. However, having an implementation that can function as a test oracle is not always the case in practice. Therefore, we construct a test oracle that is based on the error-constraints.

Error-constraints partition relevant inputs into valid and strong invalid test inputs. Valid inputs are expected to stimulate standard behavior while invalid inputs are expected to stimulate

exception behavior. From this, a test oracle can be derived to check that EH is not initiated for valid test inputs but is always initiated for invalid test inputs. For each relevant test input that satisfies all error-constraints, some standard behavior is expected. For each relevant test input for which at least one error-constraint remains unsatisfied, some exception behavior is expected.

For invalid test inputs, the exception behavior can be specified more precisely by distinguishing different exception behaviors for different error-constraints. For instance, a label that encodes an exception representation can be assigned to each error-constraint. Then, a strong invalid test input that does not satisfy a particular error-constraint is expected to induce exception behavior that matches the assigned label.

> **Running Example**   Since concrete details of the specification and sub-system of the industry partner may not be published, we use the running example to illustrate the test oracle construction. The RIPM of Figure 6.1 (Page 113) contains four error-constraints to which the following labels can be attached.
>
> $$c_1 \mapsto \text{INV\_PACKAGES}$$
> $$c_2 \mapsto \text{INV\_COUNTRY}$$
> $$c_3 \mapsto \text{INV\_PHONE}$$
> $$c_4 \mapsto \text{INV\_CODE}$$
>
> The error-constraints correspond to the four validation rules of the example implementation (Listing 2.1, Page 14) and the attached labels have a one-to-one mapping to the error codes of the example implementation.
>
> ```
> 1   if(isInvalidNumberOfPackages(packages)) return "INV_PACKAGES";
> 2   if(isInvalidCountry(country)) return "INV_COUNTRY";
> 3   if(isInvalidPhoneNumber(phone)) return "INV_PHONE";
> 4   if(isInvalidCountryCode(country, phone)) return "INV_CODE";
> ```
>
> As an example, the strong invalid test input [Packages:123, Country:UK, Phone:+44, Shipping:Standard] does not satisfy error-constraint c_1. Therefore, the SUT is expected to return the error code INV_PACKAGES.

When selecting test inputs ROBUSTA, only valid and strong invalid test inputs are selected and the test oracle can be used as describe beforehand. In this case study, the test oracle is able to reveal all faults for all valid and strong invalid test inputs. The partitioning of error-constraints is as powerful as the current implementation. Especially because the output of the current implementation is either SUCCESS or an error code.

If the output of the current implementation was more detailed, the current implementation would be able to reveal more failures beyond the simple partitioning of standard and exception behavior. Hence, the test oracle based on error-constraints is especially useful when testing validation rules. But the test oracle based on error-constraints can also be useful as an additional test oracle. This is especially true because the construction does not require much effort when the corresponding RIPM and its error-constraints already exist.

If the RIPM and its error-constraints do not exist yet, error-constraints and the test oracle can

be modeled iteratively. Therefore, parameters and values are modeled such that the exception conditions can be satisfied and validated. Then, error-constraints are modeled one at a time and refined until test inputs that satisfy the error-constraint also satisfy the exception condition and until test inputs that do not satisfy the error-constraint also violate the exception condition. The modeling can be considered complete when the partitioning is correct for all validation rules and its exception conditions.

In this case study, the IPM and RIPM are modeled depending on an implementation that is considered to be correct rather than on a non-executable specification. Therefore, the completeness of the RIPM and the test oracle can also be checked by selecting test inputs with ROBUSTA and by measuring its code coverage.

When no explicit specification exists, the RIPM and its error-constraints can be modeled depending on the SUT. When the error-constraints are modeled such that they reflect the partitioning of the SUT, the tester can decide in each iteration if the partitioning of the SUT is consistent and correctly reflects the intended behavior.

We follow this approach when modeling the error-constraints for this case study and we could identify three inconsistent exception conditions. One inconsistency is concerned with incorrect positions of parentheses, e.g. an inconsistent formula $((a \Rightarrow b) \lor c)$ is used instead of $(a \Rightarrow (b \lor c))$. The other two inconsistencies are concerned with subparts of insurance applications. One insurance application may encompass one to several sub-applications of different insurance products. Two validation rules check whether one subpart satisfies a certain precondition and then enforce satisfaction of a second condition in all subparts. However, it seems more reasonable to enforce the satisfaction of the second condition only in those subparts that also satisfy the precondition. Let $sp \in SP$ denote a subpart, let $P(a)$ denote the precondition as a predicate, and let $Q(a)$ denote the second condition. Then, an inconsistent formula $(\exists sp \in SP, P(sp)) \Rightarrow (\forall sp \in SP, Q(sp))$ is used instead of $\forall sp \in SP, P(sp) \Rightarrow Q(sp)$.

Although, we did not receive detailed feedback on the three inconsistencies and cannot confirm that they are actual faults. But nevertheless the identification of inconsistencies demonstrates that faults can already be detected while modeling error-constraints.

At this point, one can conclude that test oracles based on error-constraints are effective during testing and also able to reveal faults during the modeling of error-constraints. However, two more observations are made while modeling the error-constraints in this case study.

First, when using the test oracle with ROBUSTA, all selected test inputs are valid or strong invalid which makes the expected error code unambiguous. Since ROBUSTA is compared with IPOG-C during this case study, the test oracle is also used for test inputs selected by IPOG-C. These test inputs are valid or invalid but not necessarily strong invalid.

This has implications for the test oracle because a test input that does not satisfy two or more error-constraints is still expected to return only one error code. But the error code depends on the particular ordering of the validation rules because the first violated validation rule determines the error code.

The test oracle can be adjusted in two ways. Either the order of validation rules can be encoded in the test oracle such that the first unsatisfied error-constraint determines the expected error code. With this adjustment, the test oracle is implementation-specific and a SUT passes the test only if the order of validation rules is equal to the order of error-constraints. Or the test

oracle can be extended such that it computes a set of expected error codes and a SUT passes the test if With this adjustment, the test oracle becomes imprecise because a SUT passes the test if the actual error code equals to one of several expected error codes.

Both adjustments are a consequence of the invalid input masking effect which can be mitigated by using strong invalid test inputs. Especially because this test oracle requires the existence of error-constraints and the previous discussion indicates that ROBUSTA is more efficient than IPOG-C. The advantage of IPOG-C not requiring the effort to model error-constraints is void when using a test oracle based on error-constraints. Therefore, we believe the combination of IPOG-C and a test oracle based on error-constraints is a special case that is only relevant for comparative studies and not relevant for practical use.

Running Example   A not-strong invalid test input like [Packages:1, Country:123, Phone:123, Shipping:Standard] for the RIPM of Figure 6.1 (Page 113) does not satisfy two error-constraints c_2 and c_3.

When considering the order of error-constraints, the expected error code INV_COUNTRY of error-constraint c_2 must be returned by the SUT. This is true for the example implementation (Listing 2.1, Page 14). But it is not true for a different order of validation rules as illustrated below. Here, the order of validation rules Nos. 2 and 3 is changed and the SUT would return INV_PHONE instead.

```
1    if(isInvalidNumberOfPackages(packages)) return "INV_PACKAGES";
2    if(isInvalidPhoneNumber(phone)) return "INV_PHONE";
3    if(isInvalidCountry(country)) return "INV_COUNTRY";
4    if(isInvalidCountryCode(country, phone)) return "INV_CODE";
```

When accepting some imprecision, the test oracle may accept one of several expected error codes. In this case, the SUT is expected to either return the error code INV_COUNTRY or INV_PHONE and both ordering are considered valid.

Second, validation rules may be overlapping in practice and promote over-constrained RIPMs. Although, the over-constrained RIPMs can still be used for a test oracle in combination with IPOG-C. Or the over-constrained RIPMs can be repaired to function for ROBUSTA and for a test oracle.

Running Example   Since concrete details of the specification and sub-system of the industry partner may not be published, we again use the running example for the illustration. Again, imagine a different order of validation rules for the example implementation (Listing 2.1, Page 14).

```
1    if(isInvalidCountryCode(country, phone)) return "INV_CODE";
2    if(isInvalidNumberOfPackages(packages)) return "INV_PACKAGES";
3    if(isInvalidCountry(country)) return "INV_COUNTRY";
4    if(isInvalidPhoneNumber(phone)) return "INV_PHONE";
```

In addition, imagine that isInvalidCountryCode(country, phone) not only returns false for invalid combinations like [Country:UK, Phone:+1] but also for the combination of two invalid values [Country:123, Phone:123].

Then, a not-strong invalid test input that is selected by `IPOG-C` and that covers `[Country:123, Phone:123]` would stimulate the SUT to return `INV_CODE` instead of either `INV_COUNTRY` or `INV_PHONE`. Consequently, the test would fail. Strong invalid test inputs that are selected by `ROBUSTA` would either cover `[Country:123]` or `[Phone:123]`. Consequently, `INV_CODE` would never be returned for the invalid values and the test would not fail. Although, this part of the validation rule would remain untested.

For not-strong invalid test inputs, the error-constraint c_4 could be extended to c_4' as follows. Because c_2 and c_3 already define `[Country:123]` and `[Phone:123]` as invalid, this extension results in an over-constrained RIPM with conflicts between c_4', c_2, and c_3.

c_2 :Country $\neq$ 123

c_3 :Phone $\neq$ 123

c_4 :$\neg$(Country = UK $\wedge$ Phone = +1) $\wedge$ $\neg$(Country = USA $\wedge$ Phone = +44)

c_4' :$\neg$(Country = UK $\wedge$ Phone = +1) $\wedge$ $\neg$(Country = USA $\wedge$ Phone = +44)
    $\wedge$ $\neg$(Country = 123 $\wedge$ Phone = 123)

Using an over-constrained RIPM as a test oracle is not a problem because for a test input that covers a schema like `[Country:123, Phone:123]`, either `INV_CODE`, `INV_COUNTRY`, or `INV_PHONE` are expected.

However, the over-constrained RIPM should not be used for test selection and as a test oracle because no strong invalid test inputs could be selected that cover `[Country:123, Phone:123]`. As a solution, the error-constraints c_2 and c_3 could be relaxed such that they do not conflict with c_4'. When using the semi-automated repair technique (Section 9.2, Page 169 and following), the relaxed error-constraints are as follows.

c_2' :Country $\neq$ 123 $\vee$ (Country = 123 $\wedge$ Phone = 123)

c_3' :Phone $\neq$ 123 $\vee$ (Country = 123 $\wedge$ Phone = 123)

They can also be rewritten to the following expressions which emphasize the invalid schemata they model.

c_2' :$\neg$(Country = 123 $\wedge$ Phone = +1) $\wedge$ $\neg$(Country = 123 $\wedge$ Phone = +44)

c_3' :$\neg$(Country = UK $\wedge$ Phone = 123) $\wedge$ $\neg$(Country = USA $\wedge$ Phone = 123)

Instead of defining `[Country:123]` and `[Phone:123]` as invalid, invalid schemata that cover `Country` and `Phone` are used to separate schemata for which `INV_COUNTRY` or `INV_PHONE` is expected from the schemata for which `INV_CODE` is expected.

To conclude this comparison of the CRT test method with the CT test method, t-wise valid

coverage, $\langle *, * \rangle$-wise strong invalid coverage and the combination of both performs as well as or better than t-wise relevant coverage in terms of effectiveness and efficiency.

The t-wise valid coverage avoids invalid input masking and performs better than t-wise relevant coverage for faults that an be detected by valid test inputs. The $\langle *, * \rangle$-wise strong invalid coverage performs better than t-wise relevant coverage and the combination of both has the best performance.

When considering different configurations of $\langle *, * \rangle$-wise strong invalid coverage, $\langle \forall, * \rangle$ and $\langle a, * \rangle$ with a with higher values should be considered as they detect all faults almost always. In contrast, $\langle \exists, * \rangle$ and $\langle a, * \rangle$ with $a = 1$ require the smallest test suites for their satisfaction but also provide the lowest AFDE values.

Accidental combinations of strong invalid schemata and valid schemata are able to detect faults that actually require robustness interactions. When selecting test inputs with robustness interaction, the effectiveness is further improved but substantially more test inputs are required. Moreover, strong invalid test inputs can be considered as "partially-valid" test inputs that can also detect faults similar to valid test inputs selected by t-wise valid coverage.

When a test oracle is already available, the CT test method has the advantage of less modeling effort because error-constraints are not needed. Then, the decision whether to choose CRT or CT is a trade-off between additional effort in modeling error-constraints or additional effort in execution more test inputs.

When a test oracle not yet available, it is demonstrated that error-constraints can be used to construct effective test oracles. Then, the CRT test method should be used because the effort of modeling error-constraints is distributed while still fewer test inputs are required when comparing it with the CT test method.

13.3.4. Threats to Validity

We compared the effectiveness of the CRT test method using an implementation of the RO-BUSTA test selection strategy with the CT test method using an implementation of the IPOG-C test selection strategy. Therefore, test inputs are selected to satisfy different test adequacy criteria. To ensure an unbiased implementation, both implementations follow the guidelines of KLEINE & SIMOS [KS18].

The effectiveness of CRT and CT highly depend upon the IPM and RIPM. To avoid any bias, both the IPM and RIPM are modeled systematically and share the same set of parameters and values. To prevent falsified results due to accidental fault triggering, the orders of parameters and values are randomized and 20 different variants are used in test input selection. All presented FDE values are average values.

Since this is a case study with only one case, it is difficult to generalize the findings [RH09]. Further on, it has to be noted that the archival data of this case study is only a snapshot and the ground truth, i.e. the existing and previously-existing faults, is unknown. Hence, the data can be biased towards simpler faults that are easier to detect. To prevent too far-reaching conclusions, we describe the characteristics of the SUT and also limit our conclusions to similar systems with many exception conditions.

Test Adequacy Criteria	t	a	b	Faults												
				1	2	3	4	5	6	7	8	9	10	11	12	13
t-wise relevant coverage and the *validate all* principle	1	-	-	+	+	0.75	0.75	0.20	0.15	+	0.75	0.90	0.45	+	+	0.85
	2	-	-	+	+	+	+	0.95	+	+	+	+	+	+	+	+
	3	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
	4	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
	5	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
t-wise relevant coverage and the *fail fast* principle	1	-	-	0.00	0.00	0.05	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.05	0.00
	2	-	-	0.10	0.10	0.45	0.20	0.10	0.00	0.00	0.00	0.00	0.00	0.65	0.20	0.00
	3	-	-	0.75	0.75	+	+	0.65	0.05	0.10	0.05	0.05	0.00	+	0.65	0.00
	4	-	-	+	+	+	+	+	0.15	0.10	0.05	0.00	0.00	+	+	0.00
	5	-	-	+	+	+	+	+	0.50	0.35	0.15	0.05	0.00	+	+	0.05
t-wise valid coverage	1	-	-	0.75	0.75	+	+	0.50	0.50	+	0.80	0.00	0.00	0.00	0.00	0.00
	2	-	-	+	+	+	+	+	+	+	+	0.00	0.00	0.00	0.00	0.00
	3	-	-	+	+	+	+	+	+	+	+	0.00	0.00	0.00	0.00	0.00
⟨∗, ∗⟩-wise strong invalid coverage	-	∀	0	+	+	+	+	+	+	0.90	0.80	+	+	+	+	+
	-	∀	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	-	∃	0	0.55	0.60	0.80	0.45	0.25	0.15	0.90	0.75	+	+	+	0.35	+
	-	∃	1	+	+	+	+	0.60	0.85	+	+	+	+	+	0.35	+
	-	1	0	0.85	0.90	+	0.60	0.30	0.35	0.90	0.80	+	+	+	0.80	+
	-	1	1	+	+	+	+	0.80	+	+	+	+	+	+	+	+
	-	2	0	+	+	+	0.95	0.55	+	0.90	0.80	+	+	+	+	+
	-	2	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	-	3	0	+	+	+	+	0.80	+	0.90	0.80	+	+	+	+	+
	-	3	1	+	+	+	+	+	+	+	+	+	+	+	+	+
t-wise valid coverage and ⟨∗, ∗⟩-wise strong invalid coverage	1	∀	0	+	+	+	+	+	+	+	+	+	+	+	+	+
	1	∀	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	2	∀	0	+	+	+	+	+	+	+	+	+	+	+	+	+
	2	∀	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	∀	0	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	∀	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	1	∃	0	0.85	0.85	+	+	0.50	0.50	+	+	+	+	+	0.35	+
	1	∃	1	+	+	+	+	0.60	0.85	+	+	+	+	+	0.35	+
	2	∃	0	+	+	+	+	+	+	+	+	+	+	+	0.35	+
	2	∃	1	+	+	+	+	+	+	+	+	+	+	+	0.35	+
	3	∃	0	+	+	+	+	+	+	+	+	+	+	+	0.35	+
	3	∃	1	+	+	+	+	+	+	+	+	+	+	+	0.35	+
	1	1	0	0.90	0.90	+	+	0.55	0.65	+	+	+	+	+	0.80	+
	1	1	1	+	+	+	+	0.80	+	+	+	+	+	+	+	+
	2	1	0	+	+	+	+	+	+	+	+	+	+	+	0.80	+
	2	1	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	1	0	+	+	+	+	+	+	+	+	+	+	+	0.80	+
	3	1	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	1	2	0	+	+	+	+	0.70	+	+	+	+	+	+	+	+
	1	2	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	2	2	0	+	+	+	+	+	+	+	+	+	+	+	+	+
	2	2	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	2	0	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	2	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	1	3	0	+	+	+	+	0.85	+	+	+	+	+	+	+	+
	1	3	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	2	3	0	+	+	+	+	+	+	+	+	+	+	+	+	+
	2	3	1	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	3	0	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	3	1	+	+	+	+	+	+	+	+	+	+	+	+	+

Table 13.6.: FDE Values for Different Test Adequacy Criteria

Test Adequacy Criteria	t	a	b	AFDE
t-wise relevant	1	-	-	0.03
coverage	2	-	-	0.14
	3	-	-	0.47
	4	-	-	0.56
	5	-	-	0.62
t-wise valid	1	-	-	0.48
coverage	2	-	-	0.62
	3	-	-	0.62
$\langle *,* \rangle$-wise	-	$\forall$	0	0.98
strong invalid coverage	-	$\forall$	1	+
	-	$\exists$	0	0.68
	-	$\exists$	1	0.91
	-	1	0	0.81
	-	1	1	0.98
	-	2	0	0.94
	-	2	1	+
	-	3	0	0.96
	-	3	1	+
t-wise valid coverage	1	$\forall$	0	+
and $\langle *,* \rangle$-wise	1	$\forall$	1	+
strong invalid coverage	2	$\forall$	0	+
	2	$\forall$	1	+
	3	$\forall$	0	+
	3	$\forall$	1	+

Test Adequacy Criteria	t	a	b	AFDE
t-wise valid coverage	1	$\exists$	0	0.85
and $\langle *,* \rangle$-wise	1	$\exists$	1	0.91
strong invalid coverage	2	$\exists$	0	0.95
(continued)	2	$\exists$	1	0.95
	3	$\exists$	0	0.95
	3	$\exists$	1	0.95
	1	1	0	0.91
	1	1	1	0.98
	2	1	0	0.98
	2	1	1	+
	3	1	0	0.98
	3	1	1	+
	1	2	0	0.98
	1	2	1	+
	2	2	0	+
	2	2	1	+
	3	2	0	+
	3	2	1	+
	1	3	0	0.99
	1	3	1	+
	2	3	0	+
	2	3	1	+
	3	3	0	+
	3	3	1	+

Table 13.7.: AFDE Values for Different Test Adequacy Criteria

Chapter 14.

Evaluating Detection and Repair of Over-constrained RIPMs

Contents

As can be seen in the previous chapter about evaluating the test selection strategies, error-constraints play a crucial role in distinguishing valid from strong invalid schemata. But error-constraints are not only used in test selection strategies but can also be used in test oracle.

Unfortunately, error-constraints can be modeled too strict resulting in over-constrained RIPMs. Therefore, we develop three techniques (1) to identify conflicts and manually repair over-constrained RIPMs, (2) to semi-automatically repair over-constrained RIPMs, and (3) to deal with conflicts automatically during test input selection. In this chapter, the applicability of all three techniques is evaluated.

14.1. Evaluating the Detection and Manual Repair of Over-Constrained RIPMs

14.1.1. Experiment Design and Setup

The techniques to detect and explain over-constrained RIPMs with MISs and minimal conflict sets is based on already existing conflict detection algorithms. Therefore, the objective of this evaluation is not to assess the correctness of the proposed techniques. Instead, the objective

Name	P & V	C^{err}	Invalid	Missing
Example-1	$3^3 2^1$	$2^2 1^3$	5	0
Example-2	$3^3 2^1$	$2^2 1^3$	7	3
Registration	$6^1 4^4 3^1 2^9$	$2^{11} 1^6$	41	11
Banking-1	$4^1 3^4$	5^1	112	0
Banking-1 +10	$4^1 3^4$	5^2	122	20
Banking-1 +50	$4^1 3^4$	5^2	162	100
Banking-2	$4^1 2^{14}$	2^1	3	0
Banking-2 +1	$4^1 2^{14}$	2^2	4	2
Banking-2 +1 +1	$4^1 2^{14}$	2^3	5	4
HealthCare-3	$6^1 5^1 4^5 3^6 2^{16}$	2^{10}	31	0
HealthCare-3 +3 +4 +5	$6^1 5^1 4^5 3^6 2^{16}$	2^{13}	43	24
HealthCare-4	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	2^7	22	0
HealthCare-4 +3 +4	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	2^9	29	7

Table 14.1.: Benchmark RIPMs used for the Experiments

of this evaluation is to assess the feasibility of the proposed conflict detection and explanation techniques.

We conduct another controlled experiment where we select test inputs and measure the overhead by comparing the test input selection with conflict explanation enabled versus test input selection with conflict explanation disabled. Therefore, the aforementioned prototype `coffee4j` also includes the techniques for detection and explanation of conflicts.

Table 14.1 lists the used benchmark RIPMs. `Example-1` and `Example-2` are the correct and over-constrained examples used when discussing the technique (Section 9.1.1, Page 160 and following). `Registration` is a real-world RIPM from one of our industry partners. The other RIPMs are modified IPMs that originate from SEGALL et al. [STF11] and are often used to compare test selection strategies. For these experiments, they are modified such that all constraints are error-constraints. The identifier `A +n` indicates that RIPM `A` is extended by an error-constraint that specifies n MISs. For instance, two error-constraints are added to `Banking-2 +1 +1`. Each one specifies one invalid value that is already specified for `Banking-2`. The `P & V` column describes the parameter values in exponential notation; x^y refers to y parameters with x values. C^{err} uses exponential notation to describe error-constraints. Here, x^y refers to y error-constraints for x parameters. The `Invalid` column states the number of locally-specified schemata of all error-constraints and `Missing` states the subset of MISs for which no strong invalid test inputs are selected.

The experiments are carried out with an Intel i5 2.20 Ghz CPU and 12GB of memory. The experiments are repeated for the $\langle \forall, 0 \rangle$-wise and $\langle \forall, 1 \rangle$-wise strong invalid coverage.

14.1.2. Results and Discussion

The results for the conflict detection and explanation are summarized in Table 14.2.

The b column refers to the robustness interaction degree and `Size` refers to the sizes of test

Name	b	Size	Enabled	Disabled	Diff.
Example-1	0	5	1.47	0.73	0.74
	1	10	2.97	2.77	0.20
Example-2	0	4	5.23	1.00	4.23
	1	8	6.20	3.17	3.03
Registration	0	30	201.63	105.87	95.76
	1	105	1068.63	971.10	97.53
Banking-1	0	112	372.57	229.83	142.74
	1	112	368.50	228.27	140.23
Banking-1 +10	0	102	3056.93	483.60	2573.33
	1	102	3053.23	484.67	2568.56
Banking-1 +50	0	62	20190.10	695.93	19494.17
	1	62	20149.87	691.07	19458.80
Banking-2	0	3	2.17	1.17	1.00
	1	6	7.33	7.03	0.30
Banking-2 +1	0	2	2.60	1.00	1.60
	1	4	11.53	10.07	1.46
Banking-2 +1 +1	0	1	4.50	1.50	3.00
	1	2	17.10	13.50	3.60
HealthCare-3	0	31	205.50	191.77	13.73
	1	214	2960.87	2945.93	14.94
HealthCare-3 +3 +4 +5	0	19	618.83	169.73	449.10
	1	139	3781.07	3332.67	448.40
HealthCare-4	0	22	133.00	128.10	4.90
	1	154	2184.80	2181.37	3.43
HealthCare-4 +3 +4 +5	0	15	275.63	113.40	162.23
	1	105	2536.97	2379.53	157.44

Table 14.2.: Results of the Experiments

suites. The columns `Enabled` and `Disabled` list the measured times in milliseconds for test input selection with conflict explanation disabled and enabled, respectively.

The experiments reveal a small overhead in time for test input selection which is caused by the additional constraint solving for identification and explanation of minimal conflict sets. On average, the measured overhead, i.e. the absolute difference between test input selection times with and without explanation, is 1,763 milliseconds. For experiments with correct RIPMs without any conflicts, the average overhead is only 32 milliseconds. In contrast, the average of all experiments with over-constrained RIPMs is 2,845 milliseconds.

Two experiments (`Banking-1 +10` and `Banking-1 +50`) are significantly slower than the other experiments with the highest measured overhead of 19,494 and 2,573 milliseconds, respectively. They are slower because more MISs (10 and 50) must be identified and explained. In addition, the constraints involve five parameters whereas all other scenarios only have constraints that involve two parameters. Therefore, more explanations must be computed and the

search space is much larger.

In relative terms, enabling identification and explanation increases the generation time by 40.53% on average. The two slow experiments have a relative overhead ranging from 84.13% up to 96.57%. Ignoring the slower experiments results in an overhead of only 31.87% for the remaining experiments.

In six cases, the differences between time measured for the generation with explanation and without explanation is less than 5 milliseconds. Differences within a few milliseconds can be caused by environmental noise because the execution cannot be completely controlled. Although, the small differences highlight that the additional overhead can be neglected in many cases.

The experiments further emphasize that the identification and explanation are independent from the specified testing strength. Increasing the testing strength from one to two increases the generation time on average by 854 milliseconds. But practically no difference between enabled and disabled identification could be measured. On average, the difference is -3 milliseconds which can be explained by noise.

The experiments indicate that the additional computations for identification of MISs and explanation with minimal conflict sets is feasible and applicable to real world scenarios. In most cases, the measured overhead is very small. But even for cases with larger overhead, we believe that the manual tasks benefit from knowing why invalid schemata are missing and the additional time is worth it.

14.1.3. Threats to Validity

We compared test input selection with conflict explanation enabled versus conflict explanation disabled. Since the comparison is based on actual test input selection, our results might depend on the implementation of the algorithms and on the design of the test scenarios. To ensure an unbiased implementation, we follow the suggestions for an efficient implementation of test selection strategies by KLEINE & SIMOS [KS18]. Further on, the source code of the test selection strategies is published[1].

The test scenarios are based on existing benchmark IPMs. During execution of experiments, resource consumption of other applications may have distorted the results. Therefore, the measurements are based on 30 repetitions.

---

[1]See Section 11.3 (Page 205 and following) for more information.

Name	Correct RIPMs		Modifications	
	P & V	**Inv. Schemata**	**Added Inv. Schemata**	**MISs**
Example-1	$3^3 2^1$	$2^2 1^3$	2^2	3
Registration	$6^1 4^4 3^1 2^9$	$2^{17} 1^9$	2^{12}	11
Banking-1	$4^1 3^4$	$5^1 1^2$	$3^2 2^1$	28
Banking-2	$4^1 2^{14}$	2^3	$3^2 1^1$	4
HealthCare-1	$6^1 5^1 3^2 2^6$	$3^{18} 2^3$	1^2	7
HealthCare-2	$4^1 3^6 2^5$	$5^{18} 3^6 2^1$	$4^6 3^1$	4
HealthCare-3	$6^1 5^1 4^5 3^6 2^{16}$	2^{31}	$3^2 1^1$	12
HealthCare-4	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	2^{22}	$2^1 1^1$	13

Table 14.3.: Benchmark RIPMs used for the Experiments

14.2. Evaluating Semi-Automatic Repair of Over-Constrained RIPMs

14.2.1. Experiment Design and Setup

In Section 9.2 (Page 169 and following), we propose a semi-automatic technique to repair over-constrained RIPMs. Since no comparable work exists yet, the objective of this experiment is to evaluate the general feasibility and applicability of this technique. Therefore, this technique is implemented in our prototype `coffee4j`[2] and applied it to different over-constrained RIPMs.

The semi-automatic repair technique is evaluated in two dimensions. We measured the computational overhead when applying the technique and we compared the computed minimal diagnosis hitting sets and evaluated the resulting test inputs.

The experiments are based on 8 benchmark RIPM. Again, `Example-1` is the correct running example used when discussing the technique (Section 9.1.1, Page 160 and following). `Registration` is a real-world RIPM from one of our industry partners. The other RIPMs are modified IPMs that originate from SEGALL et al. [STF11] and are often used to compare test selection strategies. For these experiments, they are modified such that all constraints are error-constraints. All test models are listed in Table 14.3 (Page 249). The first two columns describe the correct RIPM. The `P & V` column describes the parameter values in exponential notation where v^p refers to p parameters with v values. `Inv. Schemata` describes the invalid schemata as specified by the error-constraints. Here, x^y refers to y invalid schemata for x parameters. As the correct RIPMs are not over-constrained, we added additional error-constraints to artificially create conflicts. Column 3 describes the additional invalid schemata and the fourth column describes the number of MISs.

14.2.2. Results and Discussion

Table 14.4 (Page 250) depicts the execution times for the RIPMs in milliseconds where the repair technique is applied to both correct and modified versions. The execution times show that

---

[2]See Section 11.3 (Page 205 and following) for more information.

Name	Min. Diagnosis Hitting Sets	Execution Times	
		Correct RIPMs	Modified RIPMs
Example-1	2	0.00 ms	1.02 ms
Registration	32	6.90 ms	62.14 ms
Banking-1	16	236.66 ms	6166.74 ms
Banking-2	4	0.00 ms	2.04 ms
HealthCare-1	4	4.68 ms	29.04 ms
HealthCare-2	8	11.44 ms	54.24 ms
HealthCare-3	10	12.2 ms	127.94 ms
HealthCare-4	16	6.1 ms	206.54 ms

Table 14.4.: Overview of Results

the repair technique is a feasible extension with no noteworthy computation overhead, i.e. on average the overhead is 34.75 ms. for the correct RIPMs and 831.21 ms for the modified RIPMs.

`Banking-1` caused the longest execution times because it specifies 112 invalid schemata that cover all five parameters whereas all other RIPMs specify fewer invalid schemata that cover fewer parameters. The noticeable differences in execution times between correct and modified RIPMs is caused by the additional computations of conflict and diagnosis sets which is exclusive to over-constrained RIPMs.

To evaluate the computed 92 minimal diagnosis hitting sets, they are applied to the modified RIPMs. Then, the resulting repaired RIPMs are used to generate test inputs which are compared to test inputs from the correct RIPMs. To make sure that all invalid schemata should appear in some strong invalid test input, the selected test suites are selected to satisfy $\langle \forall, 0 \rangle$-wise strong invalid coverage. The results are illustrated in Table 14.5 (Page 251) which shows a subset of minimal diagnosis hitting sets for `HealthCare-3`. All results are available in Appendix C (Page 341 and following). Each row of the table relates to another minimal diagnosis hitting set and to the generated test inputs of the repaired RIPMs.

The metric not present invalid schemata (NIS) counts the number of invalid schemata that are not generated by the repaired RIPM but are specified by the correct RIPM. For instance, applying $\triangle_{HS} = \{c_{4,2}, c_{5,2}\}$ to the over-constrained example RIPM (Figure 9.1, Page 161) removes the invalid value combinations `[Country:USA, Phone:123]` and `[Country:UK, Phone:123]` while the value `[Phone:123]` remains invalid. Then, the resulting test suite likely contains only one strong invalid test input with `[Phone:123]`. When the correct RIPM specifies both invalid value combinations, then at least one is not present in the test suite.

The metric duplicate invalid schemata (DIS) measures the number of invalid schemata that appear more than once. For instance, applying $\triangle_{HS} = \{c_{3,1}\}$ to the example RIPM removes `[Phone:123]`. The resulting test suite likely contains two strong invalid test inputs with `[Country:USA, Phone:123]` and `[Country:UK, Phone:123]`. If the correct RIPM specifies only one invalid value, then at least one invalid value combination is redundant.

The metric not valid test input (NVTI) counts the number of valid test inputs generated from the repaired RIPM that contain invalid schemata as specified by the correct RIPM.

The metric not strong invalid test input (NSITI) counts the invalid test inputs generated from

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	36	0	1	6	2
2	**37**	**0**	**0**	**0**	**0**
3	36	0	1	6	2
4	28	8	1	2	0
5	28	8	1	2	0
6	30	6	2	4	1
...	...	...	...	...	...

Table 14.5.: Results for HealthCare-3 RIPM (Excerpt)

the repaired RIPM that contain more than one invalid schema as specified by the correct RIPMs.

The applications of minimal diagnosis hitting sets show that they always result in repaired RIPMs which are not over-constrained anymore. However, not all repaired RIPMs are equivalent to the correct RIPMs since the generated test inputs do not cover invalid schemata and do not exclude invalid schemata as specified by the correct RIPMs. For each modified RIPM, exactly one minimal diagnosis hitting set is found that leads to a RIPM equivalent to the correct RIPM. Except for HealthCare-2 where four out of eight minimal diagnosis hitting sets lead to equivalent test models.

The differences between the minimal diagnosis hitting sets can be explained by a relation between the constraint that belongs to the correct RIPM and the constraint that is used in the repaired RIPM. For instance, error-constraint c_3 of the example model locally-specifies one invalid schema, i.e. $L_{c_3} = \{[\text{Phone:123}]\}$. At the same time, it forbids the appearance of one invalid schema in all other valid and strong invalid test inputs. In comparison, c_4 with $L_{c_4} = \{s_1 = [\text{Country:UK, Phone:+1}], s_2 = [\text{Country:UK, Phone:123}]\}$ locally-specifies two invalid schemata. It forbids the appearance of two invalid schemata in all other valid and invalid test inputs. Using c_3 when c_4 is correct can lead to invalid schemata that are not present. Using c_4 when c_3 is correct can lead to duplicate invalid schemata as well as to not valid and not strong invalid test inputs. Having another error-constraint c_6 equal to c_3 leads to a conflict where [Phone:123] is specified twice but cannot appear in any invalid test input. The relaxation of either c_3 or c_6 is equivalent and both diagnosis hitting sets result in a repaired RIPM equivalent to the correct RIPM.

Since the correct RIPM is unknown, an *oracle* is required to decide which minimal diagnosis hitting sets result in an equivalent RIPM. As long as no automated oracle exists, some human effort is still required for the selection.

14.2.3. Threats to Validity

The results of the comparison might depend on the implementation of the algorithms. To ensure an unbiased implementation of the test input selection, we followed the suggestions for an efficient implementation by KLEINE & SIMOS [KS18]. For the conflict detection and diagnosis, we explicitly named the used algorithms (QuickXplain[Jun04] and HS-Tree [Rei87]). Further

on, the source code of the test selection strategies is published[3].

The artificial RIPMs do not represent real-world scenarios. However, we based our evaluation on existing benchmark RIPMs, explicitly stated the characteristics and modifications to measure the implications in a controlled environment.

The experiments are carried out with an Intel i5 2.20 Ghz CPU and 12 GB of memory. During the execution, resource consumption of other applications may have distorted the results. Therefore, time measurements are based on 50 repetitions.

---

[3]See Section 11.3 (Page 205) for more information.

Name	Correct RIPMs		Modifications	
	P & V	Inv. Schemata	Added Inv. Schemata	MISs
Example-1	$3^3 2^1$	$2^2 1^3$	2^2	3
Registration	$6^1 4^4 3^1 2^9$	$2^{17} 1^9$	2^{12}	11
Banking-1	$4^1 3^4$	5^{112}	$3^2 2^1$	28
Banking-2	$4^1 2^{14}$	2^3	$3^2 1^1$	4
HealthCare-1	$6^1 5^1 3^2 2^6$	$3^{18} 2^3$	1^2	7
HealthCare-2	$4^1 3^6 2^5$	$5^{18} 3^6 2^1$	$4^6 3^1$	4
HealthCare-3	$6^1 5^1 4^5 3^6 2^{16}$	2^{31}	$3^2 1^1$	12
HealthCare-4	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	2^{22}	$2^1 1^1$	13

Table 14.6.: Benchmark RIPMs used for the Experiments

14.3. Evaluating Strong Invalid Test Input Selection from Over-Constrained RIPMs

14.3.1. Experiment Design and Setup

In Section 9.3 (Page 178 and following), we propose alternative constraint handling strategies to use in CRT with over-constrained RIPMs when manual or semi-automatic repair is rejected. The objective of this experiment is to compare the alternative strategies with the default strategy HCH and the semi-automatic repair techniques of Section 9.2 (Page 169 and following). The constraint handling strategies are implemented in our prototype `coffee4j`[4] and applied it to different over-constrained RIPMs.

We evaluate the strategies in two dimensions. First, we measure the computational overhead for test input generation and compared the default HCH strategy with B-SCH and D-SCH. Second, we analyze characteristics of test suites selected by HCH, B-SCH, and D-SCH and also compare them with the semi-automatic repair technique.

Eight benchmark RIPMs are used for the experiments. They are the same RIPMs as used in the last evaluation (Section 14.2, Page 249 and following). `Example-1` is the running example used throughout this paper. `Registration` is a real-world RIPM from an industry partner. The other RIPMs originate from SEGALL et al. [STF11] and are often used to compare test selection strategies. For these experiments, they are modified such that all constraints are error-constraints. All RIPMs are listed in Table 14.6 (Page 253). The first two columns describe the original test models. The `P & V` column describes the parameter values in exponential notation where v^p refers to p parameters with v values. `Invalid Schemata` describes the invalid schemata as specified by the error-constraints. Here, x^y refers to y invalid schemata for x parameters.

As the original RIPMs are not over-constrained, additional error-constraints are added to artificially create conflicts. Column 3 describes the additional invalid schemata and the fourth column describes the number of missing invalid schemata.

---

[4]See Section 11.3 (Page 205) for more information.

Name	Constraint Handling Strategies		
	HCH	**B-SCH**	**D-SCH**
Example-1	6.84	9.46	11.94
Registration	1002.90	1638.72	1372.02
Banking-1	615.90	6361.56	6948.76
Banking-2	26.14	38.80	42.40
HealthCare-1	201.60	300.62	489.44
HealthCare-2	3882.10	3487.20	3583.56
HealthCare-3	3432.92	4540.62	4402.14
HealthCare-4	1851.46	3575.70	3522.18

Table 14.7.: Times for Test Input Selection

14.3.2. Results and Discussion

Computational Overhead

Table 14.7 (Page 254) depicts the measured times for test input selection in milliseconds using the different constraint handling strategies. The RIPMs include the modifications to make them over-constrained. Because otherwise, the soft-constraint handling strategies are almost equal to HCH. Overall, the test input selection times show that the alternative constraint handling strategies are feasible extensions with acceptable overhead. HCH is the fastest constraint handling strategy in seven out of eight times. The absolute difference between the slowest and fastest strategy is on average 1313.09 milliseconds and the relative difference is on average 188.49%.

`Banking-1` performs significantly worse than all other RIPMs. The difference between the slowest and fastest strategy is 6332.86 milliseconds (1028.23%). It requires long test input selection times because it specifies 112 invalid schemata that cover all five parameters of the RIPM. Using B-SCH or D-SCH involves a check of all 112 invalid schemata for MISs. Without `Banking-1`, the absolute difference is 595.98 milliseconds (68.52%). Since the measured times range from milliseconds to a few seconds, a slowdown by 68.52% or even 188.49% on average makes the strategies still feasible for application. This is especially true when compared to the manual work of repairing RIPMs.

There is only one test model (`HealthCare-2`) for which HCH is slower than B-SCH and D-SCH. But the differences are only 394.90 milliseconds (11.32 %) which is the lowest difference measured among all RIPMs. Therefore, we assume this is caused by distortions from other applications or operating system services.

When comparing the computation overhead of B-SCH with D-SCH, the difference is the additional computation of minimal diagnosis sets in the case of D-SCH. In total, the differences between them are very small being 183.38 milliseconds (18.98%) on average. `Banking-1` is the RIPM that performed the worst with an overhead of 587.20 milliseconds (62.81%) for D-SCH. Without `Banking-1`, the differences are even smaller (107.14 milliseconds, 17.88%).

Name	Test Suite Size	# Correct $\triangle_{HS}$	# All $\triangle_{HS}$
Example-1	5	1	2
Registration	26	1	32
Banking-1	112	1	16
Banking-2	3	1	4
HealthCare-1	21	1	4
HealthCare-2	25	4	8
HealthCare-3	31	1	10
HealthCare-4	22	1	16

Table 14.8.: Characteristics of Repaired RIPMs

Characteristics of Generated Test Inputs

Selecting test inputs from over-constrained RIPMs with the default HCH strategy results in MISs. When repairing an over-constrained RIPM, the repaired RIPM can be used to select test inputs. But the repaired RIPM is not necessarily equivalent to the hypothetical correct RIPM. Some MISs may still not appear, some may appear more than once and some may even appear in one test input simultaneously. Therefore, we use the following metrics to analyze and compare test inputs which result from either the over-constrained or repaired RIPMs. We compare them with test inputs which result from the hypothetical correct RIPM.

The metric not present invalid schemata (NIS) counts the number of invalid schemata that do not appear in any test input despite being modeled by the correct RIPM.

This metric should not be confused with the concept of MISs. A MIS is a schema $s \in L_{c_i}$ that is locally-specified by an error-constraint c_i of a RIPM. But due to some conflicts with other constraints, the MIS is not a strong invalid schema, i.e. $s \notin I_{c_i}$.

In contrast, a NIS is a strong invalid schema that is specified by an error-constraint c_i of the correct RIPM. But the schema is not a strong invalid schema of another modified (incorrect) RIPM. NISs describe relations between correct and incorrect RIPMs.

This is also the reason why the metric values for MIS and NIS differ. For an incorrect RIPM, a schema that is not present accounts for one NIS in relation to the correct RIPM. Although, the schema is not present for a reason because it contradicts with some other schemata and all these schemata account for MISs. For instance, the over-constrained example RIPM (Figure 9.1, Page 161) specifies the three schemata [Country:USA, Phone:123], [Country:UK, Phone:123], and [Phone:123]. While the correct example RIPM only specifies [Phone:123].

For the over-constrained example RIPM, there is one NIS, i.e. [Phone:123] but there are three MIS, i.e. [Country:USA, Phone:123], [Country:UK, Phone:123], and [Phone:123].

The metric duplicate invalid schemata (DIS) counts the number of invalid schemata that do appear in at least two test inputs.

While the NIS metric is concerned with the effectiveness that may deteriorate because of

Name	Constraint Handling Strategy	Test Suite Size	# NIS	# DIS	# NSITI
Example-1	HCH	4	1	0	0
	B-SCH	7	0	1.67 (1-3)	0.97 (0-3)
	D-SCH	7	0	1.37 (1-2)	0.37 (0-1)
	Repair	5.50 (5-6)	0	0.50 (0-1)	0.17 (0-1)
Registration	HCH	30	0	0	0
	B-SCH	41	0	6.73 (6-9)	6.37 (1-7)
	D-SCH	41	0	5	0
	Repair	32.97 (30-36)	0.02 (0-1)	2.01 (0-4)	14 (0-32)
Banking-1	HCH	88	28	0	0
	B-SCH	116	0	0	0
	D-SCH	116	0	0	0
	Repair	100 (88-112)	13.14 (0-28)	0	0
Banking-2	HCH	2	2	0	0
	B-SCH	6	0	1	0
	D-SCH	6	0	1	0
	Repair	3.66 (3-4)	0.19 (0-2)	0.66 (0-1)	0
HealthCare-1	HCH	16	7	0	0
	B-SCH	23	0	0.5 (0-1)	6 (0-12)
	D-SCH	23	0	0	0
	Repair	18.50 (16-21)	3.10 (0-7)	0	0
HealthCare-2	HCH	28	2	0	0
	B-SCH	32	0	2.93 (0-2)	0
	D-SCH	32	0	2	0
	Repair	28 (25-29)	0.88 (0-2)	0.03 (0-1)	0.03 (0-1)
HealthCare-3	HCH	22	10	0	0
	B-SCH	34	0	6.27 (4-9)	11.17 (5-13)
	D-SCH	34	0	3	3.60 (2-5)
	Repair	25.50 (22-31)	4.35 (0-9)	1.14 (0-3)	5.84 (0-24)
HealthCare-4	HCH	11	11	0	0
	B-SCH	24	0	3.07 (1-5)	8.77 (3-12)
	D-SCH	24	0	2.23 (0-3)	5.27 (0-6)
	Repair	18.88 (12-22)	2.50 (0-10)	0.92 (0-2)	6.19 (0-11)

Table 14.9.: Characteristics of Selected Test Inputs

untested schemata, the DIS metric is rather concerned with the efficiency.

The metric not strong invalid test input (NSITI) (Not Strong Invalid Test Input) counts the number of test inputs that contain more than one invalid schema.

The not strong invalid test input (NSITI) metric is again concerned with the effectiveness that may deteriorate because of potential invalid input masking.

Table 14.8 depicts characteristics of repaired RIPMs, i.e. the test suite size of the correct RIPM, the number of diagnosis hitting sets that lead to a correct RIPM and the total number of computed diagnosis hitting sets. The characteristics derived from test input selection are shown in Table 14.9 (Page 256). For each metric, three numbers are presented as a (b - c) where a is the average number calculated from 30 repetitions with randomly shuffled RIPMs, b denotes the minimum and c denotes the maximum number if they differ from the average number.

For the Example-1 RIPM, four test inputs are selected using the HCH strategy and one

invalid schema is not present. B-SCH and D-SCH result in seven test inputs that include all invalid schemata as specified by the correct RIPM. However, both SCH strategies lead to some duplicate invalid schemata and to some not-strong invalid test inputs. Although, D-SCH has lower DIS and NSITI values. When comparing the SCH strategies with the semi-automatic repair strategy, two diagnosis hitting sets are computed of which one must be selected (See Table 14.8, Page 255). One of the two diagnosis hitting sets leads to a correct RIPM with which five test inputs are selected. In comparison with B-SCH and D-SCH, both repaired RIPMs require fewer test inputs. Both have no NIS, fewer DISs and also on average fewer NSITIs.

Although, `Example-1` is the only RIPM for which all metric values are equal or better than the metric values for D-SCH. We believe this is because in total only two minimal diagnosis hitting sets are computed. Hence, the chance that a repair results in an equivalent RIPM is 50%.

Except for `Example-1`, these characteristics can be observed among all RIPMs. While the semi-automatic repair technique leads to smaller test suite sizes on average and equal or lower DIS values, it requires the selection of the correct RIPM to ensure zero NIS and zero NSITI. Otherwise, some invalid schemata may remain missing. In contrast, B-SCH and D-SCH have larger test suite sizes and DIS values. But the test suites have always zero NIS values which tackles the problem of over-constrained RIPMs and MISs. In direct comparison, D-SCH leads to fewer DISs and fewer NSITIs than B-SCH which also increases the chance of detecting faults.

For the HCH strategy, the test suite size is always smaller compared to the two SCH strategies and also compared to the repair technique. This is simply because some invalid schemata are not present.

Except for `Registration`, this is true for all other RIPMs. In the case of `Registration`, the RIPM is over-constrained but all MISs are not specified by the correct RIPMs. Therefore, the conflict prevents some schemata to appear but it has no impact on the FDE because all originally-specified strong invalid schemata are present.

To conclude these experiments to evaluate the feasibility of the detection and repair of over-constrained RIPMs, it can be stated that all three approaches, i.e. the manual and semi-automatic repair as well as the alternative constraint handling strategies, are feasible when applying them to benchmark RIPMs. Conflicts can be identified, explained and based on that information, minimal diagnosis hitting sets can be computed to repair over-constrained RIPMs.

The experiments further indicate that repair is desirable to prevent NISs. Although the experiments also show the limits of the semi-automatic repair technique since an oracle is required to select the right minimal diagnosis hitting set.

When an oracle is not available, the D-SCH strategy is the best choice among the three strategies HCH, B-SCH, and D-SCH. Compared with B-SCH, D-SCH has lower DIS and NSITI values which improves efficiency and reduces effects of the invalid input masking effect. Compared with HCH, D-SCH has fewer NISs which improves the efficiency. Using SCH strategies reduces the efficiency compared to the HCH strategy because the CSPs for the SCH strategies are more complex as they require more variables and more constraints. Further on, test suites selected with SCH strategies contain more test inputs than test suites selected with the HCH strategy.

But as these experiments show, the SCH strategies are still feasible when using them with benchmark RIPMs. In contrast to the HCH strategy, no schemata are NIS and fewer test inputs

are not-strong invalid. Therefore, they reduce the main concern of over-constrained RIPMs without the need of immediately repairing them.

14.3.3. Threats to Validity

B-SCH and D-SCH are compared with HCH and with the semi-automatic repair technique. The measured times depend on the implementation of the techniques. Therefore, the constraint handling strategies are integrated into the `coffee4j` prototype which also includes HCH strategy and the semi-automatic repair technique. The source code of the test selection strategies is published[5].

The RIPMs do not represent real-world scenarios. But they are derived from existing benchmark RIPMs and the characteristics as well as modifications are public.

The experiments are carried out with an Intel i5 2.20 Ghz CPU and 12 GB of memory. Resource consumption of other applications may have distorted the results. To prevent results that are caused by symmetries between RIPM and test selection strategy, we repeated each experiment 30 times with randomly shuffled RIPMs.

---

[5]See Section 11.3 (Page 205 and following) for more information.

Chapter 15.

Evaluating Automatic Generation of Error-Constraints

Contents

The additional effort of modeling error-constraints can be perceived as an obstacle when adopting our CRT test method. To overcome this obstacle, we introduce a process for automatic generation of error-constraints that is based on FC and utilizes existing FCAs.

In this chapter, the general feasibility of the process for automatic generation of error-constraints is demonstrated. Further on, we conduct another controlled experiment to analyze the suitability of FC and FCAs.

15.1. Experiment Design and Setup

Since no comparable approaches for the CRT test method exist yet, we cannot compare our techniques with them. Instead, we evaluate the feasibility of the approach and the feasibility of FCAs in terms of preciseness and completeness. Therefore, we conduct a controlled experiment to answer the following two research objectives.

Research Objective 1 Is the process for automatic generation of error-constraints feasible?

Research Objective 2 Is FC a suitable technique to identify minimal non-conforming schemata?

We execute the process to generate error-constraints for different test scenarios. The `cof-fee4j` prototype also includes the process for automatic generation of error-constraints, FCAs to identify minimal non-conforming schemata, and strategies that group schemata and generate error-constraints. The implementations and experiments are developed in the context of a master thesis (cf. [Maß20]) which I defined and supervised. In the following, the details of the controlled experiment are introduced. Afterwards, the results are presented and discussed.

15.1.1. Fault Characterization Algorithms

As already mentioned, FCAs only compute approximate solutions because an exact solution is often infeasible. Therefore, a variety of FCAs exist. Each FCA has its own advantages and its computation of approximate solutions is based on assumptions. For instance, some FCAs assume that no constraints exist and all parameters are independent (cf. [Zhe+16]). Some FCAs require the presence of *safe values* and assume that at least one test input passes (cf. [Niu+18]). When using FCAs to identify truly-minimal invalid schemata, it is likely that numerous assumptions do not hold. For instance, constraints exist in CRT. Otherwise, CRT would not be necessary. Safe values also exist for each parameter. Otherwise, the RIPM cannot be conflict-free and strong invalid test inputs cannot be selected. The selection of FCAs that can be used in this context is restricted and our evaluation is concerned with the suitability of FCAs.

Four different FCAs are used in this experiment. `IDD` (Improved Delta Debugging) [LNL12] is a sequential FCA which requires a test selection strategy like `IPOG-C` to provide the initial test suite. `IDD` is chosen because previous results indicate very good performance in comparison to other sequential FCAs[1]. The other three FCAs are interleaving FCAs: `OFOT` (One Factor One Time) [Niu+18], `TRT` (Tuple Relationship Tree) [Niu+13], and `MixTgTe` [AGR19]. They are chosen because interleaving FCAs promise to be advantageous over sequential FCAs since identified MFCSs can be excluded from further test selection and execution. They do not require initial test suites but they require a test selection strategy that selects "one test input at a time" [Niu+18]. Therefore, the test selection strategy `AETG` [Coh+94]; [Coh+97] is used in combination with `OFOT` and `TRT`. In contrast, `MixTgTe` does not require an additional test selection strategy because it already integrates test selection.

`IDD` and `OFOT` are expected to be fast because they are have a simple design that is based on two assumptions: First, they assume that each failing test input contains only one MFCS. Second, they assume that additional test inputs for the identification of the MFCS do not fail because of another MFCS. In contrast, `TRT` and `MixTgTe` are expected to be slower but to provide a higher precision. Their design is more complex because they are able to identify multiple MFCSs that are covered by one test input and they are also able to identify MFCSs that overlapping, i.e. MFCSs that share common parameter values.

15.1.2. Scenarios

A scenario consists of a RIPM without error-constraints since they shall be generated during the experiment. In addition, another system represents the authoritative specification and partitions

---

[1]See our publication by Friedrichs, Fögen, and Lichter [FFL20] for a comparison of FCAs.

Name	Parameters & Values	Strong Invalid Schemata
Example-1	$3^3 2^1$	$2^2 1^3$
Delivery	$9^1 6^1 5^1 3^1 2^1$	2^{26}
BibTeX	$15^1 4^6 3^6$	$3^3 2^{44}$
Banking-1	$4^1 3^4$	5^{112}
Banking-2	$4^1 2^{14}$	2^3
HealthCare-1	$6^1 5^1 3^2 2^6$	$3^{18} 2^3$
HealthCare-2	$4^1 3^6 2^5$	$5^{18} 3^6 2^1$
HealthCare-3	$6^1 5^1 4^5 3^6 2^{16}$	2^{31}

Table 15.1.: RIPMs used for the Experiments

test inputs into truly-valid and truly-invalid.

Table 15.1 (Page 261) lists the benchmark RIPMs. For each RIPM, the column `Parameters & Values` describes number of parameters and values in exponential notation where x^y refers to y parameters with x values. The column `Strong Invalid Schemata` describes the number of strong invalid schemata that should be identified for the RIPM in exponential notation where x^y refers to y strong invalid schemata with x parameter-value pairs. `Example-1` refers to the incomplete example (Figure 10.4, Page 193) with no error-constraints modeled yet as discussed in Chapter 10 (Page 187 and following). The RIPMs `Delivery` and `BibTeX` are based on the pricelist of a parcel delivery company and BibTeX entry types. The other RIPMs originate from SEGALL et al. [STF11] and are often used to compare test selection strategies.

For these experiments, they are modified such that all constraints are considered as *expected* error-constraints. The initial RIPMs have an empty set of error-constraints. But the purpose of this controlled experiment is to identify and generate the expected error-constraints. Therefore, the partition of the other system is congruent with the partition of the expected error-|constraints.

15.1.3. Experiments

The scenarios are executed using the four different FCAs to search for minimal non-conforming schemata and to generate error-constraints as illustrated by the example in Chapter 10 (Page 187 and following). They are further structured into four experiments that are defined to evaluate the performance of FCAs in different contexts.

Experiment 1 In the first experiment, we execute the process for automatic generation of error-constraints with all four FCAs on all eight scenarios. Therefore, we use a testing strength of $t = 2$ and stop after the first iteration.

Experiment 2 In the next experiment, we again execute the process for automatic generation of error-constraints with all four FCAs on all eight scenarios. This time, we use the isolating strategy to group non-conforming schemata. Since additional search is required to isolate schemata, we expected an increase in precision.

Experiment 3 In the next experiment, we again use the isolating strategy to group non-conforming schemata. This time, the testing strength is increased to $t = 3$. We expect an increase in test suite size and test execution time. But we also expect increasing precision and recall.

Experiment 4 Since the `TRT` FCA provides the best performance in the previous experiments, we use `TRT` to execute several iterations of the process for automatic generation of error-constraints. The goal is to generate all expected error-constraints. Since this experiment is labour-intensive, we focus on the two scenarios with the worst performance of `TRT`.

15.1.4. Data Collection Procedure

In this controlled experiment, the FCAs are executed in the context of the process for automatic generation of error-constraints. Therefore, eight different scenarios are used in five different experiments. To systematically measure the performance of FCAs and to make comparisons among them, the following metrics are introduced.

The precision and recall metrics are used to compare the effectiveness of the FCAs. Therefore, the identified non-conforming schemata are compared to the non-conforming schemata that are specified by the test scenario.

Precision When using a FCA to identify minimal non-conforming schemata, it is important that the identification is correct. Therefore, precision is defined as the ratio between the correctly identified non-conforming schemata and the total number of identified non-conforming schemata.

$$\texttt{precision} = \frac{|\texttt{identified} \cap \texttt{specified}|}{|\texttt{identified}|}$$

Recall In addition, it is important that a FCA is able to identify many if not all existing non-conforming schemata. In that sense, this metric is a notion of completeness. Recall is defined as the ratio between the correctly identified non-conforming schemata and the total number of all specified non-conforming schemata.

$$\texttt{recall} = \frac{|\texttt{identified} \cap \texttt{specified}|}{|\texttt{specified}|}$$

Further on, the metrics **test suite size** and the **test execution time** are used to compare the efficiency of the FCAs. We view test suite size as a good measure to compare the efficiency among FCAs. In contrast, the test execution time is a good indicator to demonstrate the general feasibility. However, the test execution time is less suitable for comparisons because implementation detail can have a huge impact. For instance, the implementation of the `IPOG-C` test selection strategy relies on performance optimizations (cf. [KS18]) whereas no comparable performance optimizations exist for `AETG`.

15.2. Results and Discussion

In the following, the results of the experiments are presented and discussed for one type of experiment at a time. At the end, a summary concludes the controlled experiment.

15.2.1. Experiment 1

Algorithm	Test Suite Size	Execution Time	Precision	Recall
OFOT	194.900	3.389	0.471	0.194
TRT	13735.125	1336.871	0.932	0.705
IDD	42.425	0.007	0.161	0.117
MixTgTe	126.075	538.566	0.520	0.341

Table 15.2.: Aggregated Average Values for Experiment 1

For experiment 1, each combination of scenario and FCA is executed ten times. The average metric values of all four FCAs for all scenarios are depicted in Table D.1 (Page 346). Table 15.2 (Page 263) depicts the aggregated average values of the four FCAs among all scenarios. The execution times are depicted in seconds.

Considered individually, using `IDD` results in the lowest test suite sizes and the lowest execution times. But at the same time, using `IDD` results in the lowest values for precision and recall. `IDD` never achieves a precision or recall value of 1.

In contrast, using `TRT` produces the largest test suite sizes and requires the longest execution times. But it achieves the highest precision and recall values. `TRT` is the only FCA that has a precision value of 0.475 and a recall value of 0.873 for `HealthCare-3` while the precision and recall values of the other three FCAs are zero.

`MixTgTe` performs second best. It requires fewer test inputs and less execution time compared to `TRT` and has the second best values for precision and recall. `OFOT` has slightly worse values for precision and recall but requires more test inputs than `MixTgTe`.

Apart from `Banking-1` and `HealthCare-2`, `MixTgTe` has higher precision and recall values than `OFOT`. Those two scenarios are difficult for `MixTgTe` because `MixTgTe` guarantees the detection of non-conforming schemata with degrees smaller or equal to the testing strength [AGR19] which is $t = 2$ in this experiment. But for the two scenarios, many non-conforming schemata have a degree of 5 and cannot be detected by `MixTgTe`.

Overall, `IDD` and `OFOT` have the fastest execution times and the small test suite sizes because their approach is simpler and relies on more assumptions than `TRT` and `MixTgTe`. We believe the assumptions are responsible for the low precision and recall values of `IDD` and `OFOT`. In these scenarios, non-conforming test inputs may contain multiple non-conforming schemata which may even be overlapping. Further on, each scenario contains more than one non-conforming schema. Therefore, additional non-conforming schemata may be discovered during the identification of one non-conforming schema. Thereby, the precision and recall values of `IDD` and `OFOT` may fluctuate as a random factor is introduced by the violated assumptions.

15.2.2. Experiment 2

Algorithm	Test Suite Size	Execution Time	Precision	Recall
OFOT	200.850	4.216	0.594	0.201
TRT	13953.675	1493.343	0.996	0.620
IDD	42.975	0.057	0.186	0.118
MixTgTe	128.425	1082.440	0.594	0.378

Table 15.3.: Aggregated Average Values for Experiment 2

For the second experiment, the isolating strategy is used to group non-conforming schemata. Each combination of scenario and FCA is again executed ten times. The average metric values of all four FCAs for all scenarios are depicted in Table D.2 (Page 347). Table 15.3 (Page 264) depicts the aggregated average values of the four FCAs among all scenarios. The execution times are again depicted in seconds.

As expected, the test suite sizes and execution times increase for all FCAs in comparison to experiment 1. In general, the test suite sizes and execution times only increase slightly indicating a minor overhead of the isolating strategy.

But there are exceptions for the `HealthCare-2` and `HealthCare-3` scenarios. Both scenarios contain non-conforming schemata that are overlapping, which make the identification and classification of truly-minimal invalid schemata difficult. As a result, there are increases in execution times in three cases – both in absolute and in relative terms. The execution times of `MixTgTe` for the `HealthCare-3` scenario doubled (+4369 seconds). The execution times of `TRT` almost quadrupled for the `HealthCare-2` scenario (+103 seconds) and almost increases six-fold for the `HealthCare-3` scenario (+955 seconds).

The precision values improve because of the isolating strategy. But two exceptions exist. The `TRT` FCA has a decreased precision value of 0.971 instead of 0.98 for `Delivery`. The raw data reveals that the lowest precision value among the ten executions is increased from 0.905 to 0.929. This indicates that a slightly higher precision value can be expected when using the isolating strategy. But it also indicates that a precision value of 1.0 is achieved less often. Further on, the `OFOT` FCA has a decreased precision value of 0.072 instead of 0.094 for `BibTeX`. In this case, the lowest precision value among the ten executions also decreased from 0.333 to 0.25. Although we believe this is mainly caused by the random effects of violated assumptions.

The recall value is worse in seven out of 32 cases. Although, the deviations are rather small. The biggest change can be observed for `HealthCare-3` and the `TRT` FCA. The recall value for this combination decreases from 0.873 to 0.420. Although at the same time, the precision value increases from 0.475 to 1.0. We suspect the higher precision is responsible for the lower recall. After a non-conforming schema is identified, this non-conforming schema is excluded from further test input selection. Due to the the higher precision values, a better conformance is established which causes some non-conforming schemata to not be triggered by conventional test inputs. Consequently, `TRT` does not attempt to identify them.

Overall, the isolating strategy improves the precision. To improve the recall as well, we attempt to increase the testing strength in the next experiment.

15.2.3. Experiment 3

Algorithm	Test Suite Size	Execution Time	Precision	Recall
OFOT	272.475	95.739	0.550	0.254
TRT	15442.138	2111.939	1.000	0.716
IDD	149.938	2.911	0.100	0.077
MixTgTe	313.257	1037.827	0.804	0.633

Table 15.4.: Aggregated Average Values for Experiment 3

The third experiment uses a testing strength of $t = 3$ instead of $t = 2$. The isolating strategy is used again. In the last experiment, MixTgTe was already the slowest FCA for the HealthCare-3 scenario. With $t = 3$, executing MixTgTe on HealthCare-3 did not terminate within six hours. Therefore, we consider it as timed out. As a consequence, the aggregated values for MixTgTe are not comparable to the first experiment.

The average metric values of all four FCAs for all scenarios are depicted in Table D.3 (Page 348). Table 15.4 (Page 265) depicts the aggregated average values of the four FCAs among all scenarios. The execution times are again depicted in seconds.

As expected, the test suite sizes and execution times increase in all cases. We also expected improved recall values. While this is true for TRT and MixTgTe, it is not true for IDD and OFOT. In three out of 31 cases, the recall values of IDD and OFOT decrease in comparison to the second experiment. For instance, the recall value of OFOT decreases from 0.162 to 0.115 for the Delivery scenario. We again suspect that this is caused by random effects due to violations of assumptions. Especially because the involved recall values are already rather low for experiment no. 2.

The recall values for TRT improve because more test inputs are required to satisfy t-wise valid coverage with $t = 3$. Consequently, more non-conforming test inputs are executed. The recall values for MixTgTe improve for the same reason. But in addition, MixTgTe can identify non-conforming schemata of degree $d = 3$ which explains the improvement for the HealthCare-2 scenario. Since the Banking-1 scenario would require a testing strength of $t = 5$, the recall value of MixTgTe is still zero.

The precision values also improved for TRT and MixTgTe but worsened for IDD and OFOT. In six out of 31 cases, the precision values of IDD and OFOT worsened.

With $t = 3$, TRT achieves a precision value of 1 for all scenarios. For MixTgTe, the precision value improved from 0.678 to 0.804 which again can be explained by the guarantee of detecting non-conforming schemata that involve t or fewer parameters.

15.2.4. Experiment 4

In the previous experiments, the TRT FCA always achieves the highest values for precision and recall. While a precision value of 1 can be achieved with a testing strength of $t = 3$, the recall value often remains on a rather low level. In particular, the recall value of TRT is the lowest for Banking-1 (0.032) and HealthCare-2 (0.098).

Iteration	Non-conforming Schemata	Execution Time
1	8	1.225
2	11	1.330
3	15	1.416
4	11	1.279
5	12	1.370
6	8	1.239
7	7	1.292
8	7	1.655
9	6	1.274
10	4	1.142
11	3	1.759
12	5	1.186
13	3	1.656
14	4	2.330
15	2	2.180
16	2	2.323
17	1	0.933
18	2	2.744
19	1	2.441
SUM	112	30.783

Table 15.5.: Iterations for `TRT` and `Banking-1`

In this experiment, several iterations of the process for automatic generation of error-constraints are executed to check whether the iterative approach can improve the recall. Therefore, the process for automatic generation of error-constraints is executed with `TRT` for `Banking-1` and `HealthCare-2`. After each iteration, the RIPM is updated and the identified minimal non-conforming schemata are added as error-constraints.

Table 15.5 (Page 266) depicts the iterations for `Banking-1`. The column `Non-conforming Schemata` depicts the number of non-conforming schemata that are identified in one iteration. The column `Execution Time` depicts the execution time in seconds.

In total, 19 iterations with a testing strength of $t = 2$ are necessary to detect all minimal non-conforming schemata. Although, the term minimal is misleading in this context because `Banking-1` is a special case where all non-conforming schemata are exhaustive, i.e. the schemata encompass five parameters and the RIPM also encompasses five parameters.

Test inputs are selected to satisfy t-wise valid coverage and some selected test inputs are non-conforming. After identifying the non-conforming schemata and updating the error-constraints, a different set of test inputs is selected to satisfy the t-wise valid coverage in the next iteration. Again, some test inputs are non-conforming and the RIPM is again updated. This iterative approach works very well until all 112 schemata are identified after 19 iterations. We suspect the iterative approach works very well for `Banking-1` because of the exhaustive nature of the non-conforming schemata.

Iteration	Non-conforming Schemata	Execution Time
1	15	52.913
2	5	25.223
3	2	13.519
4	1	9.958
5	1	89.000
SUM	24	190.613

Table 15.6.: Iterations for `TRT` and `HealthCare-2`

For the second scenario, the non-conforming schemata are not exhaustive. Table 15.6 (Page 267) depicts the iterations for `HealthCare-2`. Since only 25 non-conforming schemata can be identified for `HealthCare-2`, fewer iterations are required. The first four iterations are again based on t-wise valid coverage with a testing strength of $t = 2$. But after the four iterations, all selected test inputs conform to the other system and no further non-conforming schemata can be identified. Therefore, we executed iteration five with a testing strength of $t = 3$ to identify another non-conforming schema. Then, again all selected test inputs conform to the other system. Although one non-conforming schema is still not identified. Either testing with a higher testing strength or more randomness in the selection of test inputs is required to select a test input that is non-conforming.

The second scenario highlights an inherent problem of the process for automatic generation of error-constraints. The iterative approach is able to establish conformance between a RIPM and another system. However, the conformance should not be confused with complete conformance for every test input of the exhaustive test suite. The conformance should rather be understood as an equilibrium between a RIPM, its test selection strategy and another system.

As long as the RIPM and its test selection strategy select remain unchanged and deterministically select test inputs, the established conformance can be used in regular testing. But if the selected test inputs change because the test selection strategy or the RIPM are changed, the process for automatic generation of error-constraints must be executed again to restore the equilibrium. This is because a complete conformance can only be achieved by checking all test inputs of the exhaustive test input which is usually infeasible.

15.2.5. Summary

To summarize the four experiments, we briefly answer the two research objectives.

Research Objective 1 Is the process for automatic generation of error-constraints feasible?

The four experiments indicate that the process for automatic generation of error-constraints is feasible in general. The test suite sizes that are required for the identification as well as the execution times for test input selection and test execution are within reasonable limits. Further on, the precision value is 1 when an appropriate FCA is selected for the identification of non-conforming schemata.

The recall value is on a relatively low level after one iteration of the process. But as the fourth experiment shows, an iterative approach allows finding more non-conforming schemata and it allows finding an equilibrium between the RIPM, the test selection strategy and the other system. The equilibrium describes a state where the current set of error-constraints is sufficient to select a set of test inputs of which all test inputs are truly-valid. The level of conformance can be further increased when additional test inputs are checked for conformance – either because the test selection is randomized or the testing strength is increased. But to achieve complete conformance, it would be necessary to check the exhaustive test suite which is usually impractical.

Nevertheless, as long as the the RIPM and the test selection strategy do not change, the test suite conforms to the other system and can be used for further testing activities. If the RIPM or the test selection strategy, the process for automatic generation of error-constraints can be used to find another equilibrium.

Research Objective 2 Is FC a suitable technique to identify minimal non-conforming schemata?

While the four experiments demonstrate a general feasibility of the process for automatic generation of error-constraints, they are not sufficient for a detailed analysis of individual FCAs.

However, the four experiments emphasize the importance of choosing an appropriate FCA. When FCAs are used to identify minimal non-conforming schemata, many assumptions that are typically considered by FCAs do not hold. It is important to acknowledge that more than one non-conforming schema exists. Further on, a non-conforming test input may cover two or more non-conforming schemata that can even be overlapping.

It is shown that FCAs like `IDD` or `OFOT` are rather simple but also efficient because they rely on assumptions. However, since these assumptions do not hold in the four experiments, the precision and recall values are on a relatively low level. At the same time, test suite sizes and execution times highlight their efficiency.

In contrast, `MixTgTe` and `TRT` rely on fewer assumptions but require a more complicated design. As a consequence, they are less efficient than `IDD` or `OFOT` in terms of test suite sizes and test execution times. At the same time, the precision and recall values are on a high level. Especially after several iterations of the process for automatic generation of error-constraints.

Among the considered FCAs, `TRT` performs best in terms of precision and recall because it relies on fewest assumptions and is also able to identify non-conforming schemata that are larger than the testing strength.

15.3. Threats to Validity

We compared precision and recall values for different FCAs and different scenarios. The scenarios and its RIPMs do not represent real-world scenarios. Although, they are derived from existing benchmark RIPMs and the characteristics as well as modifications are explicit. The selection of scenarios may have an impact on the results. Therefore, only general conclusions are drawn from the experiments.

Implementations details of the test selection strategies or FCAs may have an impact on the results as well. To allow and support reproducibility of the experiments, the source code of the test selection strategies and FCAs is published (Section 11.3, Page 205).

The experiments are carried out with an Intel i5 3.5 Ghz CPU running Manjaro 19.0.2. Resource consumption of other applications may have distorted the results. To reduce the potential impact of distortions, each experiment is executed 10 times.

Chapter 16.
Conclusion

At this point, all the research questions are answered by our developed solutions and their evaluation. We conclude this work with a summary and look ahead to future work.

16.1. Summary

Robustness failures can cause many problems such as unavailable programs, operating system crashes, or security breaches [Koo+97]; [Avi+04]; [BE11]. To prevent robustness failures, defensively developed programs mistrust all influences that are external to the program's boundary [Mey92]; [BF04]; [MSB11]. External faults are anticipated and programs contain computer instructions to detect errors in the program's state and further computer instructions to recover the state by eliminating the error.

Exceptions represent errors in the program's state and EH is used to detect exceptions and to recover from them [DG94]; [MT97]; [LS98]; [Gar+01]. Although, EH is intended to support the development of robust programs, it is itself a significant source of robustness faults and robustness failures [MT97]; [Mar11]; [SAW12]. Consequently, it is not only important to check that a program realizes its standard functions. It is also important to check the behavior of a program in exceptional situations.

CT is a specification-based test method and various studies demonstrate its effectiveness [Gri+06]. Unfortunately, the effectiveness of CT deteriorates in the presence of irrelevant schemata. The input masking effect causes schemata that only appear in an irrelevant test input to remain untested and faults may remain undetected [Yil+14a]; [Wu+19b]. As a countermeasure, constraint handling and exclusion-constraints are introduced to exclude irrelevant schemata and to only select relevant test inputs.

The phenomenon of input masking is not limited to irrelevant schemata and EH can also cause input masking. Test inputs with invalid schemata are expected to trigger EH. Instead of the normal control flow path, an exceptional control flow path for recovery is exercised through the SUT. As a consequence, schemata may remain untested and faults may remain undetected for schemata only appear in an invalid test input. Avoidance is not an option because invalid schemata are necessary to check the behavior in exceptional situations.

This difficulty is already acknowledged in some research (cf. [She94]; [Coh+94]; [Coh+97]; [GOA05]; [Cze06]). It led to extensions of CT that we called CRT which separates valid and strong invalid test inputs such that all schemata can be tested as expected.

In this work, we further expanded the idea of CRT. We started by establishing a taxonomy with

a precise description and clear distinction of robustness and related concepts and by discussing shortcomings of related work.

Moreover, we argued that CRT is still insufficiently researched. It was not clear if the existing CRT test methods are necessary at all or if CT test methods are already appropriate. Prior to our work, only qualitative argumentation and examples but no quantitative assessment existed.

For this, we asked the following research question.

Research Question 1 How effective is CT and what are its shortcomings in detecting faults when invalid input masking is present?

To answer the research question, we conducted a controlled experiment in which we assessed the FDE of CT. The controlled experiment shows that CT can be effective in detecting robustness faults. Although, not all robustness faults can be detected reliably. Depending on the number of EH and the distribution of valid and invalid values, a high testing strength is necessary which requires the execution of large test suites.

In conclusion, extensions like CRT are useful additions worth investigating. Further on, the controlled experiment indicated that the generality of the t-factor fault model can be a disadvantage as it does not allow further information to be modeled. As a step forward, we identified a refinement of the t-factor fault model that incorporates robustness faults and the inherent invalid input masking effect. Therefore, the second research questions was as follows.

Research Question 2 How can the t-factor fault model be refined such that it explicitly describes robustness faults and the inherent invalid input masking effect?

To answer this research question, we identified issues related to EH and derived a classification of robustness fault characteristics. Afterwards, we used this information to create a refined t-factor fault model.

With the refined t-factor fault model, CRT could be adapted so that the additional information is taken into account. This was expressed by the following research question.

Research Question 3 How can a CRT test method capture and utilize the additional information for annotating invalid values and invalid value combinations to select test suites that activate faults in the presence of invalid input masking?

To answer the research question, a new CRT test method was developed, which consists of an extended input structure (the RIPM with error-constraints), new test adequacy criteria (t-wise valid coverage and the family of $\langle *, * \rangle$-wise strong invalid coverage), and new test selection strategies (ROBUSTA and IPOG-NEG$\langle *, * \rangle$).

The corresponding evaluation consists of a theoretical examination, a controlled experiment and an industrial case study. Even though the theoretical examination revealed disadvantages in the time complexity of ROBUSTA compared to IPOG-C, the selection of test inputs is often a non-recurring activity. Therefore, we consider the FDE to be more important as long as the runtime of ROBUSTA remains within reasonable limits. When comparing the number of executed test inputs and the number of detected faults, CRT has been shown to be at least as good as CT

or better. With CRT, fewer test inputs are needed than with CT to achieve the same or a greater FDE. The industrial case study also showed that larger RIPMs can also be modeled and used for testing with CRT.

The error-constraints allow a more advantageous selection of test inputs by CRT. But error-constraints are also the main disadvantage of CRT, because they have to be modeled in addition to parameters, values, and exclusion-constraints. To mitigate this disadvantage, we developed two supporting techniques to partially automate the activity of error-constraint modeling.

First, we acknowledged that the activity of error-constraint modeling is prone to errors and it is likely to create over-constrained RIPMs. This was expressed by the following research question.

Research Question 4 How can inconsistent annotations of invalid values and invalid value combinations be detected, identified, and resolved?

To answer this research question, we utilized ideas from the research on CSP to detect and repair inconsistencies among error-constraints. Further on, we utilized the ideas to create new SCH strategies that allow postponing the repair of inconsistent error-constraints.

The corresponding evaluation showed that over-constrained RIPMs can be detected reliably. Although, a complete automation of the repair could not be realized because an oracle is required to select a minimal diagnosis hitting set. When an oracle is not available, SCH is a feasible alternative to select test inputs despite the existence of inconsistent error-constraints.

Second, we proposed a technique for the initial modeling of error-constraints. This was expressed by the following research question.

Research Question 5 How can values and value combinations that trigger EH of a SUT be identified during testing and annotated as invalid?

To answer this research question, we designed a process for the automatic generation of error-constraints that utilized ideas of FC to search for test inputs that trigger EH and to identify minimal non-conforming schemata.

The corresponding evaluation showed the designed solution works in general. However, its effectiveness and efficiency depends largely on the chosen FCA.

Last but not least, the objective of this work was to operationalize the conceptual solutions so that not only can the evaluation be carried out, but the solutions can also be applied in practice. For this purpose we designed a framework with an associated process and architecture that includes all previously discussed concepts and algorithms. Further on, we implemented the framework based on the Java programming language and the JUnit 5 test automation framework.

With this, our work is now finally complete. Nevertheless, our work has raised further questions, which should also be addressed in order to further pursue the idea of the CRT. These are discussed subsequently.

16.2. Future Work

The idea of CRT can be further pursued in different areas. Some possible extensions to this work are discussed below. Since error-constraints are introduced as a new concept in this work, there is unexploited potential to support and ensure the completeness and correctness of error-constraints.

Develop Visual Aid for Modeling Error-Constraints   As already mentioned, error-constraints require additional effort in modeling. To reduce this effort, further support in modeling is imaginable. For instance, SEGALL et al. [STF11] introduce a Cartesian product view of IPMs. Users select any set of parameters and the Cartesian product view displays a list of all schemata among the value domains of the selected parameters. The view utilizes different colors to indicate whether a schema is irrelevant, relevant, or partially-relevant. By reviewing the listed schemata, users may identify schemata that have been mistakenly classified as relevant or irrelevant. Furthermore, users may update exclusion-constraints and immediately review the updated classification. According to SEGALL et al., the Cartesian product view is "very effective in helping the user to avoid omissions. The user is faced with the Cartesian product of a set of parameters, thus reducing the risk of value combinations being left unnoticed."

This idea could also be transferred and extended to CRT and RIPMs with error-constraints where not only relevance but also validness would be displayed. Users could distinguish between strong invalid schemata, invalid schemata and valid schemata. Further on, they could update error-constraints whenever they identify incorrect classifications.

Further on, this idea could also be beneficial to visualize conflicts among error-constraints in over-constrained RIPMs and to support their explanation and repair.

Improve Detection and Repair of Over-Constrained RIPMs   The manual or semi-automatic repair activities of over-constrained RIPMs cannot only be improved by the aforementioned visual aid.

The satisfiability function $\texttt{SAT} : C^* \times A \rightarrow Bool$ has already been recognised as a performance bottleneck within test selection strategies (cf. [Wu+19b]; [Wu+19a]). Instead of explicitly enumerating all schemata that should be excluded or transforming the RIPM into a CSP, YU et al. [Yu+15] introduced an approach to deal with constraints based on the notion of "minimal forbidden tuples", i.e. minimal irrelevant schemata, which delivers a better performance in most cases. This approach can be directly transferred to error-constraints and the selection of strong invalid test inputs. In fact, our *coffee4j* framework contains interchangeable constraint checking implementations that are based on CSPs and minimal irrelevant schemata.

It is to be expected that the performance of conflict detection techniques, conflict diagnosis techniques, and soft constraint handling strategies would also improve when an approach based on minimal irrelevant schemata rather than CSPs were used.

Another potential improvement is concerned with the automatic relaxation of conflicting constraints. In Section 9.2.2 (Page 174 and following), our $\texttt{Exclude}(c_i, s_j)$ function denotes a simple strategy to relax constraint c_i by excluding s_j via logical disjunction. Unfortunately, repeated automatic relaxation may result in an error-constraint $c_i' = c_i \vee s_{j_1} \vee s_{j_2} \vee s_{j_3} \vee \ldots$ that

is difficult to comprehend. More sophisticated functions to relax constraints may improve the comprehensibility.

Extend Automatic Generation of Error-Constraints   Our technique to automatically generate error-constraints starts with an empty set of error-constraints and assumes that the RIPMs may be under-constrained but never over-constrained.

This assumption could be relaxed. Similar to GARGANTINI et al. [Gar+16]; [GPR17], our technique could be extended to not only identify and generate error-constraints for non-conforming schemata that indicate under-constrained error-constraints. The identification could be expanded to search for non-conforming schemata that indicate over-constrained error-constraints. Furthermore, the generation could be expanded to relax over-constrained error-constraints.

This extension would require a change in the search strategy, similar to the search strategies proposed by GARGANTINI et al. [Gar+16]. Instead of only selecting valid test inputs and checking whether they are truly-valid, as it is currently the case, strong invalid test inputs must also be selected and checked if they are truly-strong invalid.

Besides the completeness and correctness of error-constraints, further ideas are concerned with the utilization of error-constraints. They are discussed below.

Increase Collateral Strong Invalid Coverage   Testing with t-wise coverage involves so-called *collateral coverage*. The concept of collateral coverage describes the effect that not only schemata with degrees $d \leq t$ also some schemata with degrees $d > t$ can be detected by t-wise coverage [CZ11]; [Pet+15]. This is because a test suite that satisfies t-wise coverage does not only cover all schemata with degrees $d \leq t$ but also covers some schemata with degrees $d = (t + k)$ with $k > 0$ and $d \leq n$. Therefore, there is also a probability to detect faults that are represented by MFCSs with degrees $d > t$.

For CT and t-wise coverage already exists research to intentionally increase collateral coverage. For instance, CHEN & ZHANG [CZ11] propose to replace "do not care" values with values that cover as many yet uncovered schemata with degree $t + 1$. CHOI et al. [Cho+16] introduce a test selection strategy that increases collateral coverage by maximizing the distance between test inputs.

These ideas can be directly transferred to t-wise valid coverage. But transferring them to strong invalid coverage test adequacy criteria requires additional work. Since it can be deduced from the case study (Section 13.3, Page 227 and following) that collateral coverage can improve the FDE, we believe the transfer would be desirable. We could observe that strong invalid test inputs that violate the l-th exception condition behave as "partially valid" test inputs for the prior $l - 1$ validation rules. Although the robustness interaction ($b > 0$) is originally designed to detect configuration-dependent robustness faults, further faults can be detected by the increased number of test inputs and its collateral coverage. Without robustness interaction ($b = 0$), the effect cannot be achieved because the IPOG-NEG$\langle *, 0 \rangle$ test selection strategies assign replace the "do not care" values randomly.

But the "do not care" values are often not only replaced randomly but also similarly, as the same test selection strategy is iteratively applied to select strong invalid test inputs for several

error-constraints. Here an iteration-spanning optimization of the "do not care" values would be desirable in order to increase the FDE via collateral coverage without enlarging test suites.

Evaluate Test Oracles based on Error-Constraint   Selecting adequate test inputs is not the only challenge in testing. In addition, a correct and accurate test oracle is required to not only trigger failures but also to reveal failures [Wey82]; [AL85]; [Bar+15]; [LO17]. The existence of a test oracle is often assumed. Following WEYUKER [Wey82], "the belief that the tester is routinely able to determine whether or not the test output is correct is the oracle assumption". In practice, test oracles seldom exist just like that but must be developed in addition to test models and test scripts increasing the costs of testing (cf. [Bar+15]).

In the case study (Section 13.3, Page 227 and following), we could not only observe that the specified error-constraints improve the selection of test inputs. Additionally, we could observe that the partitioning of valid and strong invalid schemata as defined by the error-constraints can also be used to define test oracles. Therefore, error-constraints can be beneficial in both dimensions. The benefits are especially true for test projects where a test oracle is not yet available. To further pursue the idea of test oracles based on error-constraints, further case studies and further constructions of test oracles are necessary.

Part V.

Appendix

Appendix A.

Data of FDE in the Presence of Invalid Input Masking

This is the appendix to Section 5.1 (Page 74 and following) which contains all FDE values for all test scenario families used in the experiment.

l	t	6-1-1-6	12-1-1-6	18-1-1-6	24-1-1-6	30-1-1-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	57	51	51	44	57
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	32	28	26	26	29
3	2	67	75	84	85	86
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	15	8	10	15	12
4	2	42	45	51	49	60
4	3	74	83	90	89	87
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	10	8	8	0	10
5	2	14	28	24	32	31
5	3	32	46	62	63	73
5	4	78	96	95	96	97
5	5	100	100	100	100	100
6	1	2	5	5	3	4
6	2	9	12	15	13	14
6	3	25	31	33	32	46
6	4	42	69	69	73	78
6	5	70	87	98	100	99

Table A.1.: FDE for Test Scenario Families with six EHs, one valid value and one invalid value per parameter

l	t	6-1-2-6	12-1-2-6	18-1-2-6	24-1-2-6	30-1-2-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	68	63	64	65	69
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	19	18	23	20	29
3	2	76	94	88	91	96
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	9	8	8	8	6
4	2	28	48	41	44	49
4	3	77	88	96	95	96
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	2	3	1	5	3
5	2	14	14	17	17	19
5	3	40	46	60	55	69
5	4	86	95	98	99	100
5	5	100	100	100	100	100
6	1	1	1	1	2	0
6	2	5	8	7	7	6
6	3	12	15	22	16	36
6	4	34	60	67	69	80
6	5	76	96	100	99	99

Table A.2.: FDE for Test Scenario Families with six EHs, one valid value and two invalid values per parameter

l	t	6-1-3-6	12-1-3-6	18-1-3-6	24-1-3-6	30-1-3-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	83	73	79	77	79
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	15	13	16	14	14
3	2	83	87	94	96	93
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	8	3	4	6	6
4	2	26	39	36	41	37
4	3	90	96	95	97	99
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	3	0	1	1	0
5	2	8	15	11	8	11
5	3	36	38	54	51	56
5	4	83	94	99	99	100
5	5	100	100	100	100	100
6	1	0	0	0	0	0
6	2	1	3	1	4	2
6	3	14	12	15	17	19
6	4	36	49	66	67	61
6	5	78	99	100	100	100

Table A.3.: FDE for Test Scenario Families with six EHs, one valid value and three invalid values per parameter

l	t	6-1-4-6	12-1-4-6	18-1-4-6	24-1-4-6	30-1-4-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	85	81	82	80	76
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	18	13	15	16	23
3	2	92	93	95	97	93
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	1	3	0	4	3
4	2	24	30	31	34	38
4	3	90	95	98	98	99
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	1	1	0	1	1
5	2	4	4	12	12	10
5	3	30	45	42	49	47
5	4	90	100	100	100	99
5	5	100	100	100	100	100
6	1	0	0	0	0	0
6	2	0	1	2	1	2
6	3	5	10	4	7	16
6	4	22	46	53	49	49
6	5	84	96	100	100	100

Table A.4.: FDE for Test Scenario Families with six EHs, one valid value and four invalid values per parameter

l	t	6-1-5-6	12-1-5-6	18-1-5-6	24-1-5-6	30-1-5-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	86	86	89	76	82
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	16	15	10	16	10
3	2	92	93	93	97	96
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	1	2	5	4	1
4	2	21	33	23	31	38
4	3	97	93	99	97	94
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	0	1	0	0	1
5	2	1	9	4	5	5
5	3	24	30	38	43	37
5	4	91	97	100	100	100
5	5	100	100	100	100	100
6	1	0	0	0	0	0
6	2	0	1	1	0	1
6	3	3	5	6	10	4
6	4	27	36	43	51	58
6	5	90	100	100	100	100

Table A.5.: FDE for Test Scenario Families with six EHs, one valid value and five invalid values per parameter

l	t	6-2-2-6	12-2-2-6	18-2-2-6	24-2-2-6	30-2-2-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	86	86	78	84	91
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	51	49	36	57	44
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	26	27	32	20	22
4	2	86	93	96	97	93
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	16	11	10	12	14
5	2	62	65	70	76	76
5	3	100	99	99	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	6	6	7	4	4
6	2	29	49	50	42	40
6	3	88	97	98	98	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100

Table A.6.: FDE for Test Scenario Families with six EHs, two valid values and two invalid values per parameter

l	t	6-3-3-6	12-3-3-6	18-3-3-6	24-3-3-6	30-3-3-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	96	97	90	94	96
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	66	74	61	67	56
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	35	45	39	35	36
4	2	100	100	99	100	100
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	13	22	20	15	12
5	2	90	88	97	91	98
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	10	7	3	3	10
6	2	78	69	71	74	72
6	3	99	100	100	100	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100

Table A.7.: FDE for Test Scenario Families with six EHs, three valid values and three invalid values per parameter

l	t	6-2-1-6	12-2-1-6	18-2-1-6	24-2-1-6	30-2-1-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	63	74	65	66	70
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	43	34	38	35	52
3	2	98	99	100	99	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	30	35	33	29	31
4	2	90	96	95	95	99
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	22	22	20	19	27
5	2	72	82	84	85	91
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	12	14	11	13	18
6	2	61	61	69	63	69
6	3	95	99	100	100	99
6	4	100	100	100	100	100
6	5	100	100	100	100	100

Table A.8.: FDE for Test Scenario Families with six EHs, two valid values and one invalid value per parameter

l	t	6-3-1-6	12-3-1-6	18-3-1-6	24-3-1-6	30-3-1-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	71	72	76	70	72
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	59	60	57	57	53
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	34	51	52	44	43
4	2	100	100	100	100	100
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	30	27	37	29	34
5	2	97	100	98	100	98
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	24	25	20	22	23
6	2	85	94	87	90	95
6	3	100	100	100	100	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100

Table A.9.: FDE for Test Scenario Families with six EHs, three valid values and one invalid value per parameter

l	t	6-4-1-6	12-4-1-6	18-4-1-6	24-4-1-6	30-4-1-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	77	85	76	76	85
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	65	54	60	62	55
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	59	65	46	50	51
4	2	100	100	100	100	100
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	46	41	33	40	33
5	2	100	100	100	100	100
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	33	39	35	34	41
6	2	97	98	100	100	100
6	3	100	100	100	100	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100

Table A.10.: FDE for Test Scenario Families with six EHs, four valid values and one invalid value per parameter

l	t	6-5-1-6	12-5-1-6	18-5-1-6	24-5-1-6	30-5-1-6
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	80	83	86	86	89
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	61	74	61	70	73
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	63	62	60	53	52
4	2	100	100	100	100	100
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	46	47	47	46	56
5	2	100	100	100	100	100
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	45	39	41	44	45
6	2	99	100	100	100	100
6	3	100	100	100	100	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100

Table A.11.: FDE for Test Scenario Families with six EHs, five valid values and one invalid value per parameter

l	t	6-1-1-6	12-1-1-12	18-1-1-18	24-1-1-24	30-1-1-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	57	53	46	41	42
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	32	28	25	27	24
3	2	67	63	83	85	80
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	15	15	14	16	6
4	2	42	43	50	51	56
4	3	74	86	91	90	90
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	10	3	5	7	4
5	2	14	23	26	33	34
5	3	32	50	53	68	70
5	4	78	89	95	96	97
5	5	100	100	100	100	100
6	1	2	3	4	5	7
6	2	9	10	21	12	14
6	3	25	29	34	30	41
6	4	42	64	73	76	76
6	5	70	94	99	97	98
7	1		2	1	0	3
7	2		8	7	7	11
7	3		17	20	19	19
7	4		31	36	46	55
7	5		64	63	89	87
8	1		0	1	1	0
8	2		6	6	1	6
8	3		12	8	11	16
8	4		23	19	28	30
8	5		39	51	56	51

Table A.12.: FDE for Test Scenario Families with one valid value per parameter, one invalid value per parameter, and an equal number of EHs (1/4)

l	t	6-1-1-6	12-1-1-12	18-1-1-18	24-1-1-24	30-1-1-30
9	1		0	1	0	2
9	2		1	3	5	2
9	3		4	6	5	2
9	4		7	13	12	17
9	5		21	31	34	35
10	1		0	1	0	0
10	2		0	4	0	2
10	3		2	1	3	2
10	4		5	9	6	9
10	5		6	13	19	21
11	1		0	0	1	0
11	2		0	1	0	1
11	3		2	1	1	0
11	4		3	7	2	4
11	5		10	6	9	13
12	1		0	0	0	1
12	2		0	0	1	1
12	3		1	1	2	0
12	4		2	2	1	1
12	5		2	3	7	5
13	1			0	0	0
13	2			1	0	0
13	3			0	0	0
13	4			1	0	2
13	5			0	3	2
14	1			0	0	0
14	2			0	0	0
14	3			0	0	0
14	4			0	0	2
14	5			1	0	1
15	1			0	0	0
15	2			0	0	0
15	3			0	1	0
15	4			0	1	0
15	5			1	0	1
16	1			0	0	0
16	2			0	0	0
16	3			0	0	1
16	4			0	0	0
16	5			0	0	0

Table A.13.: FDE for Test Scenario Families with one valid value per parameter, one invalid value per parameter, and an equal number of EHs (2/4)

l	t	6-1-1-6	12-1-1-12	18-1-1-18	24-1-1-24	30-1-1-30
17	1			0	0	0
17	2			0	0	0
17	3			0	0	0
17	4			0	0	0
17	5			0	0	0
18	1			0	0	0
18	2			0	0	0
18	3			0	0	0
18	4			0	0	0
18	5			0	0	0
19	1				0	0
19	2				0	0
19	3				0	0
19	4				0	0
19	5				0	0
20	1				0	0
20	2				0	0
20	3				0	0
20	4				0	0
20	5				0	0
21	1				0	0
21	2				0	0
21	3				0	0
21	4				0	0
21	5				0	0
22	1				0	0
22	2				0	0
22	3				0	0
22	4				0	0
22	5				0	0
23	1				0	0
23	2				0	0
23	3				0	0
23	4				0	0
23	5				0	0
24	1				0	0
24	2				0	0
24	3				0	0
24	4				0	0
24	5				0	0

Table A.14.: FDE for Test Scenario Families with one valid value per parameter, one invalid value per parameter, and an equal number of EHs (3/4)

l	t	6-1-1-6	12-1-1-12	18-1-1-18	24-1-1-24	30-1-1-30
25	1				0	0
25	2					0
25	3					0
25	4					0
25	5					0
26	1					0
26	2					0
26	3					0
26	4					0
26	5					0
27	1					0
27	2					0
27	3					0
27	4					0
27	5					0
27	1					0
27	2					0
27	3					0
27	4					0
27	5					0
28	1					0
28	2					0
28	3					0
28	4					0
28	5					0
29	1					0
29	2					0
29	3					0
29	4					0
29	5					0
30	1					0
30	2					0
30	3					0
30	4					0
30	5					0

Table A.15.: FDE for Test Scenario Families with one valid value per parameter, one invalid value per parameter, and an equal number of EHs (4/4)

l	t	6-1-2-6	12-1-2-12	18-1-2-18	24-1-2-24	30-1-2-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	68	68	70	68	72
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	19	13	19	15	19
3	2	76	88	83	88	93
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	9	8	8	14	12
4	2	28	42	46	45	45
4	3	77	95	98	95	97
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	2	2	3	2	4
5	2	14	22	19	16	19
5	3	40	52	63	50	62
5	4	86	92	98	99	98
5	5	100	100	100	100	100
6	1	1	1	1	2	0
6	2	5	7	6	7	3
6	3	12	18	19	14	31
6	4	34	57	64	74	72
6	5	76	96	100	100	100
7	1		0	0	0	0
7	2		1	0	2	3
7	3		5	10	9	9
7	4		33	31	38	35
7	5		61	73	82	76
8	1		0	0	0	1
8	2		0	0	0	0
8	3		3	6	3	0
8	4		8	11	13	11
8	5		23	31	48	46

Table A.16.: FDE for Test Scenario Families with one valid value per parameter, two invalid values per parameter, and an equal number of EHs (1/4)

l	t	6-1-2-6	12-1-2-12	18-1-2-18	24-1-2-24	30-1-2-30
9	1		0	0	0	0
9	2		0	0	0	0
9	3		0	1	1	1
9	4		2	3	10	4
9	5		9	12	10	23
10	1		0	0	0	0
10	2		0	1	1	0
10	3		1	2	0	0
10	4		0	1	3	2
10	5		4	4	9	2
11	1		0	0	0	0
11	2		0	0	0	0
11	3		0	0	0	0
11	4		2	1	0	1
11	5		1	2	0	1
12	1		0	0	0	0
12	2		0	0	0	0
12	3		0	0	0	0
12	4		0	0	0	0
12	5		0	0	0	0
13	1			0	0	0
13	2			0	0	0
13	3			0	0	0
13	4			0	0	0
13	5			0	0	1
14	1			0	0	0
14	2			0	0	0
14	3			0	0	0
14	4			0	0	0
14	5			0	0	0
15	1			0	0	0
15	2			0	0	0
15	3			0	0	0
15	4			0	0	0
15	5			0	0	0
16	1			0	0	0
16	2			0	0	0
16	3			0	0	0
16	4			0	0	0
16	5			0	0	0

Table A.17.: FDE for Test Scenario Families with one valid value per parameter, two invalid values per parameter, and an equal number of EHs (2/4)

l	t	6-1-2-6	12-1-2-12	18-1-2-18	24-1-2-24	30-1-2-30
17	1			0	0	0
17	2			0	0	0
17	3			0	0	0
17	4			0	0	0
17	5			0	0	0
18	1			0	0	0
18	2			0	0	0
18	3			0	0	0
18	4			0	0	0
18	5			0	0	0
19	1				0	0
19	2				0	0
19	3				0	0
19	4				0	0
19	5				0	0
20	1				0	0
20	2				0	0
20	3				0	0
20	4				0	0
20	5				0	0
21	1				0	0
21	2				0	0
21	3				0	0
21	4				0	0
21	5				0	0
22	1				0	0
22	2				0	0
22	3				0	0
22	4				0	0
22	5				0	0
23	1				0	0
23	2				0	0
23	3				0	0
23	4				0	0
23	5				0	0
24	1				0	0
24	2				0	0
24	3				0	0
24	4				0	0
24	5				0	0

Table A.18.: FDE for Test Scenario Families with one valid value per parameter, two invalid values per parameter, and an equal number of EHs (3/4)

l	t	6-1-2-6	12-1-2-12	18-1-2-18	24-1-2-24	30-1-2-30
25	1				0	0
25	2					0
25	3					0
25	4					0
25	5					0
26	1					0
26	2					0
26	3					0
26	4					0
26	5					0
27	1					0
27	2					0
27	3					0
27	4					0
27	5					0
28	1					0
28	2					0
28	3					0
28	4					0
28	5					0
29	1					0
29	2					0
29	3					0
29	4					0
29	5					0
30	1					0
30	2					0
30	3					0
30	4					0
30	5					0

Table A.19.: FDE for Test Scenario Families with one valid value per parameter, two invalid values per parameter, and an equal number of EHs (4/4)

l	t	6-1-3-6	12-1-3-12	18-1-3-18	24-1-3-24	30-1-3-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	83	75	77	76	69
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	15	15	16	19	14
3	2	83	90	94	92	93
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	8	6	10	4	6
4	2	26	34	41	35	43
4	3	90	91	95	96	96
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	3	3	0	3	0
5	2	8	8	11	9	15
5	3	36	51	50	52	52
5	4	83	94	99	100	99
5	5	100	100	100	100	100
6	1	0	0	0	1	0
6	2	1	0	1	2	1
6	3	14	11	13	20	12
6	4	36	54	54	64	72
6	5	78	97	99	99	99
7	1		0	0	0	0
7	2		2	0	0	1
7	3		4	7	8	6
7	4		12	21	19	16
7	5		47	71	72	80
8	1		0	0	0	0
8	2		0	0	0	1
8	3		0	3	1	2
8	4		1	8	8	6
8	5		14	18	25	25

Table A.20.: FDE for Test Scenario Families one valid value per parameter, three invalid values per parameter, and an equal number of EHs (1/4)

l	t	6-1-3-6	12-1-3-12	18-1-3-18	24-1-3-24	30-1-3-30
9	1		0	0	0	0
9	2		0	0	0	0
9	3		0	0	0	0
9	4		1	2	1	0
9	5		4	5	4	5
10	1		0	0	0	0
10	2		0	0	0	0
10	3		0	0	0	0
10	4		0	0	1	0
10	5		2	3	0	1
11	1		0	0	0	0
11	2		0	0	0	0
11	3		0	0	0	0
11	4		0	0	0	0
11	5		0	1	0	0
12	1		0	0	0	0
12	2		0	0	0	0
12	3		0	0	0	0
12	4		0	0	0	0
12	5		0	0	0	0
13	1			0	0	0
13	2			0	0	0
13	3			0	0	0
13	4			0	0	0
13	5			0	1	0
14	1			0	0	0
14	2			0	0	0
14	3			0	0	0
14	4			0	0	0
14	5			0	0	0
15	1			0	0	0
15	2			0	0	0
15	3			0	0	0
15	4			0	0	0
15	5			0	0	0
16	1			0	0	0
16	2			0	0	0
16	3			0	0	0
16	4			0	0	0
16	5			0	0	0

Table A.21.: FDE for Test Scenario Families one valid value per parameter, three invalid values
        per parameter, and an equal number of EHs (2/4)

l	t	6-1-3-6	12-1-3-12	18-1-3-18	24-1-3-24	30-1-3-30
17	1			0	0	0
17	2			0	0	0
17	3			0	0	0
17	4			0	0	0
17	5			0	0	0
18	1			0	0	0
18	2			0	0	0
18	3			0	0	0
18	4			0	0	0
18	5			0	0	0
19	1				0	0
19	2				0	0
19	3				0	0
19	4				0	0
19	5				0	0
20	1				0	0
20	2				0	0
20	3				0	0
20	4				0	0
20	5				0	0
21	1				0	0
21	2				0	0
21	3				0	0
21	4				0	0
21	5				0	0
22	1				0	0
22	2				0	0
22	3				0	0
22	4				0	0
22	5				0	0
23	1				0	0
23	2				0	0
23	3				0	0
23	4				0	0
23	5				0	0
24	1				0	0
24	2				0	0
24	3				0	0
24	4				0	0
24	5				0	0

Table A.22.: FDE for Test Scenario Families one valid value per parameter, three invalid values per parameter, and an equal number of EHs (3/4)

l	t	6-1-3-6	12-1-3-12	18-1-3-18	24-1-3-24	30-1-3-30
25	1				0	0
25	2					0
25	3					0
25	4					0
25	5					0
26	1					0
26	2					0
26	3					0
26	4					0
26	5					0
27	1					0
27	2					0
27	3					0
27	4					0
27	5					0
28	1					0
28	2					0
28	3					0
28	4					0
28	5					0
29	1					0
29	2					0
29	3					0
29	4					0
29	5					0
30	1					0
30	2					0
30	3					0
30	4					0
30	5					0

Table A.23.: FDE for Test Scenario Families one valid value per parameter, three invalid values per parameter, and an equal number of EHs (4/4)

l	t	6-1-4-6	12-1-4-12	18-1-4-18	24-1-4-24	30-1-4-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	85	87	76	76	72
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	18	8	17	13	18
3	2	92	89	93	100	94
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	1	1	4	3	3
4	2	24	25	35	39	42
4	3	90	92	99	97	98
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	1	1	1	1	0
5	2	4	7	4	9	6
5	3	30	37	38	51	58
5	4	90	94	100	100	100
5	5	100	100	100	100	100
6	1	0	1	0	0	1
6	2	0	1	1	2	2
6	3	5	11	9	11	15
6	4	22	40	45	64	60
6	5	84	99	100	100	100
7	1		0	0	0	0
7	2		0	0	0	0
7	3		1	1	2	1
7	4		10	9	13	12
7	5		46	54	63	71
8	1		0	0	0	0
8	2		0	0	0	0
8	3		0	0	0	0
8	4		4	4	7	7
8	5		14	18	23	18

Table A.24.: FDE for Test Scenario Families with one valid value per parameter, four invalid values per parameter, and an equal number of EHs (1/4)

l	t	6-1-4-6	12-1-4-12	18-1-4-18	24-1-4-24	30-1-4-30
9	1		0	0	0	0
9	2		0	0	0	0
9	3		0	0	2	0
9	4		0	0	1	0
9	5		2	4	3	1
10	1		0	0	0	0
10	2		0	0	0	0
10	3		0	0	0	0
10	4		0	0	0	0
10	5		0	0	2	2
11	1		0	0	0	0
11	2		0	0	0	0
11	3		0	0	0	0
11	4		0	0	0	0
11	5		0	0	0	0
12	1		0	0	0	0
12	2		0	0	0	0
12	3		0	0	0	0
12	4		0	0	0	0
12	5		0	0	0	0
13	1			0	0	0
13	2			0	0	0
13	3			0	0	0
13	4			0	0	0
13	5			0	0	0
14	1			0	0	0
14	2			0	0	0
14	3			0	0	0
14	4			0	0	0
14	5			0	0	0
15	1			0	0	0
15	2			0	0	0
15	3			0	0	0
15	4			0	0	0
15	5			0	0	0
16	1			0	0	0
16	2			0	0	0
16	3			0	0	0
16	4			0	0	0
16	5			0	0	0

Table A.25.: FDE for Test Scenario Families with one valid value per parameter, four invalid values per parameter, and an equal number of EHs (2/4)

l	t	6-1-4-6	12-1-4-12	18-1-4-18	24-1-4-24	30-1-4-30
17	1			0	0	0
17	2			0	0	0
17	3			0	0	0
17	4			0	0	0
17	5			0	0	0
18	1			0	0	0
18	2			0	0	0
18	3			0	0	0
18	4			0	0	0
18	5			0	0	0
19	1				0	0
19	2				0	0
19	3				0	0
19	4				0	0
19	5				0	0
20	1				0	0
20	2				0	0
20	3				0	0
20	4				0	0
20	5				0	0
21	1				0	0
21	2				0	0
21	3				0	0
21	4				0	0
21	5				0	0
22	1				0	0
22	2				0	0
22	3				0	0
22	4				0	0
22	5				0	0
23	1				0	0
23	2				0	0
23	3				0	0
23	4				0	0
23	5				0	0
24	1				0	0
24	2				0	0
24	3				0	0
24	4				0	0
24	5				0	0

Table A.26.: FDE for Test Scenario Families with one valid value per parameter, four invalid values per parameter, and an equal number of EHs (3/4)

l	t	6-1-4-6	12-1-4-12	18-1-4-18	24-1-4-24	30-1-4-30
25	1				0	0
25	2					0
25	3					0
25	4					0
25	5					0
26	1					0
26	2					0
26	3					0
26	4					0
26	5					0
27	1					0
27	2					0
27	3					0
27	4					0
27	5					0
28	1					0
28	2					0
28	3					0
28	4					0
28	5					0
29	1					0
29	2					0
29	3					0
29	4					0
29	5					0
30	1					0
30	2					0
30	3					0
30	4					0
30	5					0

Table A.27.: FDE for Test Scenario Families with one valid value per parameter, four invalid values per parameter, and an equal number of EHs (4/4)

l	t	6-1-5-6	12-1-5-12	18-1-5-18	24-1-5-24	30-1-5-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	86	84	81	84	87
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	16	16	10	17	16
3	2	92	91	92	93	97
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	1	0	2	5	2
4	2	21	25	18	32	29
4	3	97	97	95	98	99
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	0	0	0	1	0
5	2	1	5	5	5	2
5	3	24	28	37	51	43
5	4	91	96	99	99	100
5	5	100	100	100	100	100
6	1	0	0	0	0	0
6	2	0	2	0	0	0
6	3	3	7	3	7	7
6	4	27	36	44	52	59
6	5	90	99	100	100	100
7	1		0	0	0	0
7	2		0	0	0	0
7	3		1	0	1	3
7	4		9	10	13	10
7	5		45	51	62	68
8	1		0	0	0	0
8	2		0	0	0	0
8	3		0	0	2	0
8	4		1	2	3	1
8	5		9	4	16	15

Table A.28.: FDE for Test Scenario Families with one valid value per parameter, five invalid values per parameter, and an equal number of EHs (1/4)

l	t	6-1-5-6	12-1-5-12	18-1-5-18	24-1-5-24	30-1-5-30
9	1	0	0	0	0
9	2	0	0	0	0
9	3	0	1	0	0
9	4	0	0	1	1
9	5	3	3	3	0
10	1	0	0	0	0
10	2	0	0	0	0
10	3	0	0	0	0
10	4	0	0	0	0
10	5	1	0	2	1
11	1	0	0	0	0
11	2	0	0	0	0
11	3	0	0	0	0
11	4	0	0	0	0
11	5	0	0	0	0
12	1	0	0	0	0
12	2	0	0	0	0
12	3	0	0	0	0
12	4	0	0	0	0
12	5	0	0	0	0
13	1		0	0	0
13	2		0	0	0
13	3		0	0	0
13	4		0	0	0
13	5		0	0	0
14	1		0	0	0
14	2		0	0	0
14	3		0	0	0
14	4		0	0	0
14	5		0	0	0
15	1		0	0	0
15	2		0	0	0
15	3		0	0	0
15	4		0	0	0
15	5		0	0	0
16	1		0	0	0
16	2		0	0	0
16	3		0	0	0
16	4		0	0	0
16	5		0	0	0

Table A.29.: FDE for Test Scenario Families with one valid value per parameter, five invalid values per parameter, and an equal number of EHs (2/4)

l	t	6-1-5-6	12-1-5-12	18-1-5-18	24-1-5-24	30-1-5-30
17	1			0	0	0
17	2			0	0	0
17	3			0	0	0
17	4			0	0	0
17	5			0	0	0
18	1			0	0	0
18	2			0	0	0
18	3			0	0	0
18	4			0	0	0
18	5		0		0	0
19	1				0	0
19	2				0	0
19	3				0	0
19	4				0	0
19	5				0	0
20	1				0	0
20	2				0	0
20	3				0	0
20	4				0	0
20	5				0	0
21	1				0	0
21	2				0	0
21	3				0	0
21	4				0	0
21	5				0	0
22	1				0	0
22	2				0	0
22	3				0	0
22	4				0	0
22	5				0	0
23	1				0	0
23	2				0	0
23	3				0	0
23	4				0	0
23	5				0	0
24	1				0	0
24	2				0	0
24	3				0	0
24	4				0	0
24	5				0	0

Table A.30.: FDE for Test Scenario Families with one valid value per parameter, five invalid values per parameter, and an equal number of EHs (3/4)

l	t	6-1-5-6	12-1-5-12	18-1-5-18	24-1-5-24	30-1-5-30
25	1				0	0
25	2					0
25	3					0
25	4					0
25	5					0
26	1					0
26	2					0
26	3					0
26	4					0
26	5					0
27	1					0
27	2					0
27	3					0
27	4					0
27	5					0
28	1					0
28	2					0
28	3					0
28	4					0
28	5					0
29	1					0
29	2					0
29	3					0
29	4					0
29	5					0
30	1					0
30	2					0
30	3					0
30	4					0
30	5					0

Table A.31.: FDE for Test Scenario Families with one valid value per parameter, five invalid values per parameter, and an equal number of EHs (4/4)

l	t	6-2-2-6	12-2-2-12	18-2-2-18	24-2-2-24	30-2-2-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	86	87	83	88	83
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	51	45	46	46	49
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	26	24	25	24	19
4	2	86	98	89	95	97
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	16	8	18	13	12
5	2	62	67	77	72	77
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	6	7	6	7	6
6	2	29	44	45	41	60
6	3	88	97	100	98	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100
7	1		3	2	3	1
7	2		23	26	22	21
7	3		80	75	89	85
7	4		99	100	100	100
7	5		100	100	100	100
8	1		0	2	1	1
8	2		14	23	14	9
8	3		50	55	59	68
8	4		98	99	99	100
8	5		100	100	100	100

Table A.32.: FDE for Test Scenario Families with two valid values per parameter, two invalid values per parameter, and an equal number of EHs (1/4)

l	t	6-2-2-6	12-2-2-12	18-2-2-18	24-2-2-24	30-2-2-30
9	1		2	0	0	1
9	2		9	8	8	5
9	3		25	33	33	41
9	4		82	91	96	94
9	5		100	100	100	100
10	1		1	1	0	0
10	2		2	3	2	7
10	3		12	26	21	20
10	4		53	71	72	77
10	5		99	97	100	100
11	1		0	1	1	0
11	2		1	1	1	3
11	3		9	9	11	9
11	4		30	40	52	44
11	5		82	97	95	99
12	1		0	0	0	0
12	2		1	1	2	5
12	3		6	3	6	4
12	4		15	19	30	29
12	5		55	66	80	90
13	1			0	0	0
13	2			0	0	0
13	3			3	5	5
13	4			19	13	12
13	5			53	54	68
14	1			0	0	0
14	2			0	0	1
14	3			2	5	1
14	4			8	7	8
14	5			27	34	25
15	1			0	0	0
15	2			0	2	0
15	3			0	0	1
15	4			3	7	5
15	5			11	18	28
16	1			0	0	0
16	2			0	0	0
16	3			0	0	0
16	4			2	1	0
16	5			9	9	5

Table A.33.: FDE for Test Scenario Families with two valid values per parameter, two invalid values per parameter, and an equal number of EHs (2/4)

l	t	6-2-2-6	12-2-2-12	18-2-2-18	24-2-2-24	30-2-2-30
17	1			0	0	0
17	2			0	0	0
17	3			0	0	0
17	4			2	1	2
17	5			4	5	10
18	1			0	0	0
18	2			0	0	0
18	3			0	0	0
18	4			0	0	1
18	5			1	1	2
19	1				0	0
19	2				0	0
19	3				0	0
19	4				0	1
19	5				2	2
20	1				0	0
20	2				0	1
20	3				0	0
20	4				0	0
20	5				1	0
21	1				0	0
21	2				0	0
21	3				0	0
21	4				0	0
21	5				0	0
22	1				0	0
22	2				0	0
22	3				0	0
22	4				0	0
22	5				0	0
23	1				0	0
23	2				0	0
23	3				0	0
23	4				0	0
23	5				0	0
24	1				0	0
24	2				0	0
24	3				0	0
24	4				0	0
24	5				0	0

Table A.34.: FDE for Test Scenario Families with two valid values per parameter, two invalid values per parameter, and an equal number of EHs (3/4)

l	t	6-2-2-6	12-2-2-12	18-2-2-18	24-2-2-24	30-2-2-30
25	1				0	0
25	2					0
25	3					0
25	4					0
25	5					0
26	1					0
26	2					0
26	3					0
26	4					0
26	5					0
27	1					0
27	2					0
27	3					0
27	4					0
27	5					0
28	1					0
28	2					0
28	3					0
28	4					0
28	5					0
29	1					0
29	2					0
29	3					0
29	4					0
29	5					0
30	1					0
30	2					0
30	3					0
30	4					0
30	5					0

Table A.35.: FDE for Test Scenario Families with two valid values per parameter, two invalid values per parameter, and an equal number of EHs (4/4)

l	t	6-3-3-6	12-3-3-12	18-3-3-18	24-3-3-24	30-3-3-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	96	97	96	93	96
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	66	61	71	55	65
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	35	32	33	34	37
4	2	100	100	100	100	100
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	13	13	14	24	11
5	2	90	91	96	95	96
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	10	6	8	7	4
6	2	78	74	75	80	74
6	3	99	100	100	100	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100
7	1		4	6	6	4
7	2		40	43	54	54
7	3		98	100	100	100
7	4		100	100	100	100
7	5		100	100	100	100
8	1		1	3	0	4
8	2		20	23	43	21
8	3		91	96	96	99
8	4		100	100	100	100
8	5		100	100	100	100

Table A.36.: FDE for Test Scenario Families with three valid values per parameter, three invalid values per parameter, and an equal number of EHs (1/4)

l	t	6-3-3-6	12-3-3-12	18-3-3-18	24-3-3-24	30-3-3-30
9	1		0	2	1	3
9	2		11	11	17	15
9	3		69	81	81	83
9	4		100	100	100	100
9	5		100	100	100	100
10	1		1	0	0	0
10	2		11	5	6	9
10	3		45	49	48	60
10	4		98	100	100	100
10	5		100	100	100	100
11	1		0	0	0	0
11	2		1	5	5	7
11	3		24	22	31	39
11	4		92	96	97	99
11	5		100	100	100	100
12	1		1	2	1	0
12	2		2	1	1	0
12	3		13	19	16	20
12	4		71	72	76	85
12	5		100	100	100	100
13	1			0	0	0
13	2			0	1	0
13	3			11	11	12
13	4			49	53	62
13	5			98	100	99
14	1			0	0	0
14	2			1	0	0
14	3			6	1	4
14	4			28	31	40
14	5			92	96	97
15	1			0	0	0
15	2			0	0	0
15	3			2	2	7
15	4			22	11	16
15	5			76	81	83
16	1			0	0	0
16	2			0	0	0
16	3			0	0	0
16	4			7	9	13
16	5			44	51	57

Table A.37.: FDE for Test Scenario Families with three valid values per parameter, three invalid values per parameter, and an equal number of EHs (2/4)

l	t	6-3-3-6	12-3-3-12	18-3-3-18	24-3-3-24	30-3-3-30
17	1			0	0	0
17	2			0	0	0
17	3			0	1	1
17	4			2	2	4
17	5			33	32	30
18	1			0	0	0
18	2			0	0	0
18	3			0	0	0
18	4			1	0	0
18	5			14	25	24
19	1				0	0
19	2				0	0
19	3				0	0
19	4				2	0
19	5				7	7
20	1				0	0
20	2				0	0
20	3				0	0
20	4				0	0
20	5				6	5
21	1				0	0
21	2				0	0
21	3				0	0
21	4				1	1
21	5				2	1
22	1				0	0
22	2				0	0
22	3				0	0
22	4				0	0
22	5				2	4
23	1				0	0
23	2				0	0
23	3				0	0
23	4				0	0
23	5				1	1
24	1				0	0
24	2				0	0
24	3				0	0
24	4				0	0
24	5				0	1

Table A.38.: FDE for Test Scenario Families with three valid values per parameter, three invalid values per parameter, and an equal number of EHs (3/4)

l	t	6-3-3-6	12-3-3-12	18-3-3-18	24-3-3-24	30-3-3-30
25	1				0	0
25	2					0
25	3					0
25	4					0
25	5					0
26	1					0
26	2					0
26	3					0
26	4					0
26	5					2
27	1					0
27	2					0
27	3					0
27	4					0
27	5					0
28	1					0
28	2					0
28	3					0
28	4					0
28	5					0
29	1					0
29	2					0
29	3					0
29	4					0
29	5					0
30	1					0
30	2					0
30	3					0
30	4					0
30	5					0

Table A.39.: FDE for Test Scenario Families with three valid values per parameter, three invalid values per parameter, and an equal number of EHs (4/4)

l	t	6-2-1-6	12-2-1-12	18-2-1-18	24-2-1-24	30-2-1-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	63	63	66	77	66
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	43	48	55	39	49
3	2	98	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	30	34	27	25	32
4	2	90	86	92	97	95
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	22	22	20	15	23
5	2	72	81	79	81	85
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	12	12	11	3	17
6	2	61	63	67	69	66
6	3	95	98	100	99	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100
7	1		8	7	4	9
7	2		40	53	52	45
7	3		92	94	99	97
7	4		100	100	100	100
7	5		100	100	100	100
8	1		5	6	8	3
8	2		27	34	32	35
8	3		78	86	91	87
8	4		100	100	100	100
8	5		100	100	100	100

Table A.40.: FDE for Test Scenario Families with two valid values per parameter, one invalid value per parameter, and an equal number of EHs (1/4)

l	t	6-2-1-6	12-2-1-12	18-2-1-18	24-2-1-24	30-2-1-30
9	1		6	3	5	5
9	2		22	28	36	21
9	3		69	73	70	72
9	4		100	99	100	100
9	5		100	100	100	100
10	1		2	4	2	0
10	2		13	17	17	17
10	3		52	52	61	55
10	4		95	96	98	99
10	5		100	100	100	100
11	1		4	4	2	1
11	2		10	11	11	7
11	3		33	47	52	56
11	4		85	84	94	97
11	5		100	100	100	100
12	1		1	2	1	2
12	2		7	5	12	8
12	3		34	29	30	34
12	4		67	78	88	83
12	5		99	100	100	100
13	1			2	0	1
13	2			7	4	6
13	3			17	22	28
13	4			52	51	70
13	5			97	99	100
14	1			1	0	0
14	2			3	5	8
14	3			23	12	18
14	4			50	50	47
14	5			91	95	97
15	1			0	0	0
15	2			4	1	3
15	3			15	11	12
15	4			31	37	39
15	5			80	91	85
16	1			0	0	0
16	2			1	3	4
16	3			6	9	5
16	4			20	23	26
16	5			69	68	77

Table A.41.: FDE for Test Scenario Families with two valid values per parameter, one invalid value per parameter, and an equal number of EHs (2/4)

l	t	6-2-1-6	12-2-1-12	18-2-1-18	24-2-1-24	30-2-1-30
17	1			0	0	0
17	2			2	2	2
17	3			2	8	8
17	4			15	18	18
17	5			48	52	61
18	1			0	0	1
18	2			3	0	2
18	3			2	2	2
18	4			13	16	16
18	5			38	36	44
19	1				0	0
19	2				0	1
19	3				2	2
19	4				14	10
19	5				29	34
20	1				0	0
20	2				0	0
20	3				1	2
20	4				10	6
20	5				19	19
21	1				0	0
21	2				1	0
21	3				2	3
21	4				5	3
21	5				12	14
22	1				0	0
22	2				1	0
22	3				0	0
22	4				2	3
22	5				11	8
23	1				0	0
23	2				0	1
23	3				0	0
23	4				1	2
23	5				4	8
24	1				0	0
24	2				0	0
24	3				0	2
24	4				2	2
24	5				3	5

Table A.42.: FDE for Test Scenario Families with two valid values per parameter, one invalid value per parameter, and an equal number of EHs (3/4)

l	t	6-2-1-6	12-2-1-12	18-2-1-18	24-2-1-24	30-2-1-30
25	1				0	0
25	2					0
25	3					0
25	4					1
25	5					2
26	1					0
26	2					0
26	3					2
26	4					1
26	5					2
27	1					0
27	2					0
27	3					0
27	4					1
27	5					0
28	1					0
28	2					1
28	3					0
28	4					0
28	5					1
29	1					0
29	2					0
29	3					0
29	4					0
29	5					0
30	1					0
30	2					0
30	3					0
30	4					0
30	5					0

Table A.43.: FDE for Test Scenario Families with two valid values per parameter, one invalid value per parameter, and an equal number of EHs (4/4)

l	t	6-3-1-6	12-3-1-12	18-3-1-18	24-3-1-24	30-3-1-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	71	69	83	81	72
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	59	50	58	54	68
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	34	51	48	48	47
4	2	100	99	100	100	100
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	30	37	40	21	31
5	2	97	97	99	99	100
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	24	18	26	24	21
6	2	85	90	95	94	96
6	3	100	100	100	100	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100
7	1		18	22	16	20
7	2		80	86	86	85
7	3		100	100	100	100
7	4		100	100	100	100
7	5		100	100	100	100
8	1		17	10	16	9
8	2		72	76	80	72
8	3		100	100	100	100
8	4		100	100	100	100
8	5		100	100	100	100

Table A.44.: FDE for Test Scenario Families with three valid values per parameter, one invalid value per parameter, and an equal number of EHs (1/4)

l	t	6-3-1-6	12-3-1-12	18-3-1-18	24-3-1-24	30-3-1-30
9	1		9	8	17	13
9	2		54	75	65	65
9	3		99	100	100	100
9	4		100	100	100	100
9	5		100	100	100	100
10	1		7	4	6	5
10	2		53	49	57	54
10	3		99	99	100	100
10	4		100	100	100	100
10	5		100	100	100	100
11	1		5	3	4	4
11	2		31	47	45	52
11	3		95	99	100	100
11	4		100	100	100	100
11	5		100	100	100	100
12	1		4	1	3	3
12	2		21	38	49	34
12	3		86	92	95	98
12	4		100	100	100	100
12	5		100	100	100	100
13	1			10	4	2
13	2			27	20	29
13	3			81	85	94
13	4			100	100	100
13	5			100	100	100
14	1			2	1	2
14	2			22	25	18
14	3			67	78	77
14	4			100	100	100
14	5			100	100	100
15	1			3	2	2
15	2			18	20	21
15	3			56	63	85
15	4			100	100	100
15	5			100	100	100
16	1			0	1	2
16	2			11	14	12
16	3			54	67	62
16	4			94	100	98
16	5			100	100	100

Table A.45.: FDE for Test Scenario Families with three valid values per parameter, one invalid value per parameter, and an equal number of EHs (2/4)

l	t	6-3-1-6	12-3-1-12	18-3-1-18	24-3-1-24	30-3-1-30
17	1			0	0	0
17	2			10	10	9
17	3			42	49	50
17	4			94	94	97
17	5			100	100	100
18	1			0	0	0
18	2			8	13	2
18	3			32	30	38
18	4			85	93	95
18	5			100	100	100
19	1				1	1
19	2				6	7
19	3				31	29
19	4				88	86
19	5				100	100
20	1				0	1
20	2				6	6
20	3				32	25
20	4				75	74
20	5				100	100
21	1				0	0
21	2				4	3
21	3				16	16
21	4				74	74
21	5				100	100
22	1				0	1
22	2				7	2
22	3				9	12
22	4				63	51
22	5				98	99
23	1				0	0
23	2				0	1
23	3				14	13
23	4				35	48
23	5				95	98
24	1				0	0
24	2				0	0
24	3				8	13
24	4				37	33
24	5				88	91

Table A.46.: FDE for Test Scenario Families with three valid values per parameter, one invalid value per parameter, and an equal number of EHs (3/4)

l	t	6-3-1-6	12-3-1-12	18-3-1-18	24-3-1-24	30-3-1-30
25	1				0	0
25	2					1
25	3					5
25	4					31
25	5					89
26	1					1
26	2					0
26	3					6
26	4					23
26	5					82
27	1					0
27	2					1
27	3					4
27	4					11
27	5					63
28	1					0
28	2					1
28	3					5
28	4					15
28	5					53
29	1					0
29	2					0
29	3					2
29	4					13
29	5					48
30	1					0
30	2					1
30	3					2
30	4					7
30	5					33

Table A.47.: FDE for Test Scenario Families with three valid values per parameter, one invalid value per parameter, and an equal number of EHs (4/4)

l	t	6-4-1-6	12-4-1-12	18-4-1-18	24-4-1-24	30-4-1-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	77	80	83	79	87
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	65	61	57	68	61
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	59	55	49	54	54
4	2	100	100	100	100	100
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	46	45	43	34	46
5	2	100	100	99	100	100
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	33	31	36	35	28
6	2	97	100	99	100	99
6	3	100	100	100	100	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100
7	1		25	26	31	27
7	2		98	99	98	99
7	3		100	100	100	100
7	4		100	100	100	100
7	5		100	100	100	100
8	1		27	25	13	17
8	2		91	94	96	95
8	3		100	100	100	100
8	4		100	100	100	100
8	5		100	100	100	100

Table A.48.: FDE for Test Scenario Families with four valid values per parameter, one invalid value per parameter, and an equal number of EHs (1/4)

l	t	6-4-1-6	12-4-1-12	18-4-1-18	24-4-1-24	30-4-1-30
9	1		13	22	23	21
9	2		87	91	95	92
9	3		100	100	100	100
9	4		100	100	100	100
9	5		100	100	100	100
10	1		8	15	15	12
10	2		77	82	77	82
10	3		100	100	100	100
10	4		100	100	100	100
10	5		100	100	100	100
11	1		13	10	11	16
11	2		68	57	82	70
11	3		100	100	100	100
11	4		100	100	100	100
11	5		100	100	100	100
12	1		7	13	8	7
12	2		54	69	70	70
12	3		100	100	100	100
12	4		100	100	100	100
12	5		100	100	100	100
13	1			10	9	8
13	2			53	50	66
13	3			100	100	100
13	4			100	100	100
13	5			100	100	100
14	1			3	5	6
14	2			47	47	55
14	3			100	100	100
14	4			100	100	100
14	5			100	100	100
15	1			3	6	6
15	2			46	40	42
15	3			98	99	98
15	4			100	100	100
15	5			100	100	100
16	1			2	5	1
16	2			27	30	37
16	3			97	97	98
16	4			100	100	100
16	5			100	100	100

Table A.49.: FDE for Test Scenario Families with four valid values per parameter, one invalid value per parameter, and an equal number of EHs (2/4)

l	t	6-4-1-6	12-4-1-12	18-4-1-18	24-4-1-24	30-4-1-30
17	1			3	0	1
17	2			23	28	36
17	3			93	96	89
17	4			100	100	100
17	5			100	100	100
18	1			3	0	5
18	2			22	21	32
18	3			90	89	96
18	4			100	100	100
18	5			100	100	100
19	1				1	1
19	2				21	11
19	3				85	84
19	4				100	100
19	5				100	100
20	1				0	1
20	2				16	25
20	3				77	78
20	4				100	100
20	5				100	100
21	1				1	1
21	2				14	9
21	3				63	77
21	4				100	100
21	5				100	100
22	1				1	0
22	2				6	9
22	3				55	62
22	4				99	100
22	5				100	100
23	1				0	1
23	2				9	12
23	3				57	60
23	4				100	100
23	5				100	100
24	1				0	0
24	2				6	7
24	3				34	32
24	4				98	99
24	5				100	100

Table A.50.: FDE for Test Scenario Families with four valid values per parameter, one invalid value per parameter, and an equal number of EHs (3/4)

l	t	6-4-1-6	12-4-1-12	18-4-1-18	24-4-1-24	30-4-1-30
25	1				0	0
25	2					6
25	3					41
25	4					96
25	5					100
26	1					1
26	2					3
26	3					35
26	4					92
26	5					100
27	1					0
27	2					4
27	3					29
27	4					88
27	5					100
28	1					2
28	2					1
28	3					18
28	4					79
28	5					100
29	1					0
29	2					3
29	3					18
29	4					78
29	5					100
30	1					0
30	2					1
30	3					11
30	4					66
30	5					100

Table A.51.: FDE for Test Scenario Families with four valid values per parameter, one invalid value per parameter, and an equal number of EHs (4/4)

l	t	6-5-1-6	12-5-1-12	18-5-1-18	24-5-1-24	30-5-1-30
1	1	100	100	100	100	100
1	2	100	100	100	100	100
1	3	100	100	100	100	100
1	4	100	100	100	100	100
1	5	100	100	100	100	100
2	1	80	85	89	87	86
2	2	100	100	100	100	100
2	3	100	100	100	100	100
2	4	100	100	100	100	100
2	5	100	100	100	100	100
3	1	61	67	66	59	65
3	2	100	100	100	100	100
3	3	100	100	100	100	100
3	4	100	100	100	100	100
3	5	100	100	100	100	100
4	1	63	51	58	60	55
4	2	100	100	100	100	100
4	3	100	100	100	100	100
4	4	100	100	100	100	100
4	5	100	100	100	100	100
5	1	46	48	53	54	49
5	2	100	100	100	100	100
5	3	100	100	100	100	100
5	4	100	100	100	100	100
5	5	100	100	100	100	100
6	1	45	39	40	42	46
6	2	99	100	100	100	100
6	3	100	100	100	100	100
6	4	100	100	100	100	100
6	5	100	100	100	100	100
7	1		39	42	33	33
7	2		99	100	100	100
7	3		100	100	100	100
7	4		100	100	100	100
7	5		100	100	100	100
8	1		26	26	28	27
8	2		98	98	99	100
8	3		100	100	100	100
8	4		100	100	100	100
8	5		100	100	100	100

Table A.52.: FDE for Test Scenario Families with five valid values per parameter, one invalid value per parameter, and an equal number of EH (1/4)

l	t	6-5-1-6	12-5-1-12	18-5-1-18	24-5-1-24	30-5-1-30
9	1		23	24	20	27
9	2		96	99	99	100
9	3		100	100	100	100
9	4		100	100	100	100
9	5		100	100	100	100
10	1		24	21	20	19
10	2		90	93	96	96
10	3		100	100	100	100
10	4		100	100	100	100
10	5		100	100	100	100
11	1		16	18	19	10
11	2		86	93	97	96
11	3		100	100	100	100
11	4		100	100	100	100
11	5		100	100	100	100
12	1		16	14	15	12
12	2		81	85	90	90
12	3		100	100	100	100
12	4		100	100	100	100
12	5		100	100	100	100
13	1			15	10	13
13	2			85	80	78
13	3			100	100	100
13	4			100	100	100
13	5			100	100	100
14	1			7	7	8
14	2			70	69	66
14	3			100	100	100
14	4			100	100	100
14	5			100	100	100
15	1			6	3	9
15	2			68	68	67
15	3			100	100	100
15	4			100	100	100
15	5			100	100	100
16	1			7	11	3
16	2			56	73	51
16	3			100	100	100
16	4			100	100	100
16	5			100	100	100

Table A.53.: FDE for Test Scenario Families with five valid values per parameter, one invalid value per parameter, and an equal number of EH (2/4)

l	t	6-5-1-6	12-5-1-12	18-5-1-18	24-5-1-24	30-5-1-30
17	1			2	1	6
17	2			50	58	59
17	3			99	100	100
17	4			100	100	100
17	5			100	100	100
18	1			2	8	5
18	2			28	46	57
18	3			98	99	100
18	4			100	100	100
18	5			100	100	100
19	1				1	2
19	2				37	49
19	3				100	100
19	4				100	100
19	5				100	100
20	1				3	3
20	2				36	35
20	3				97	100
20	4				100	100
20	5				100	100
21	1				2	4
21	2				32	37
21	3				96	98
21	4				100	100
21	5				100	100
22	1				3	4
22	2				31	28
22	3				98	98
22	4				100	100
22	5				100	100
23	1				0	1
23	2				28	23
23	3				90	94
23	4				100	100
23	5				100	100
24	1				1	2
24	2				20	17
24	3				92	88
24	4				100	100
24	5				100	100

Table A.54.: FDE for Test Scenario Families with five valid values per parameter, one invalid value per parameter, and an equal number of EH (3/4)

l	t	6-5-1-6	12-5-1-12	18-5-1-18	24-5-1-24	30-5-1-30
25	1				0	1
25	2					14
25	3					83
25	4					100
25	5					100
26	1					1
26	2					13
26	3					84
26	4					100
26	5					100
27	1					1
27	2					10
27	3					70
27	4					100
27	5					100
28	1					0
28	2					10
28	3					65
28	4					100
28	5					100
29	1					0
29	2					9
29	3					59
29	4					100
29	5					100
30	1					0
30	2					10
30	3					49
30	4					100
30	5					100

Table A.55.: FDE for Test Scenario Families with five valid values per parameter, one invalid value per parameter, and an equal number of EH (4/4)

Appendix B.

Data of Evaluating Combinatorial Robustness Test Selection Strategies

This is the appendix to Section 13.2 (Page 219 and following) which contains FDE values and test suites sizes for all test scenario used in the experiment.

Strategy	IPOG-C									
Strength	1		2		3		4		5	
Scenario	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.
1-1-1	5.00	0.27	36.60	0.59	199.00	0.88	922.80	0.99	3433.50	1.00
1-1-2	5.00	0.22	36.60	0.53	199.00	0.84	922.80	0.99	3433.20	1.00
1-1-3	5.00	0.20	36.60	0.39	199.00	0.74	922.80	0.92	3433.20	0.99
1-1-4	5.00	0.19	36.60	0.34	199.00	0.55	922.80	0.76	3433.40	0.90
1-2-1	7.00	0.23	70.00	0.55	552.00	0.86	3618.80	0.99	19427.60	1.00
1-2-2	7.00	0.17	70.00	0.47	552.00	0.78	3618.80	0.97	19436.80	1.00
1-2-3	7.00	0.15	70.00	0.34	552.00	0.63	3618.80	0.85	19437.30	0.97
1-2-4	7.00	0.15	70.00	0.30	552.00	0.45	3618.80	0.69	19440.00	0.83
1-3-1	5.00	0.23	50.20	0.61	324.10	0.91	1952.80	1.00	10736.70	1.00
1-3-2	5.00	0.20	50.20	0.56	324.10	0.88	1949.00	1.00	10738.10	1.00
1-3-3	5.00	0.16	50.20	0.42	324.10	0.77	1952.00	0.96	10739.70	1.00
1-3-4	5.00	0.16	50.20	0.35	324.10	0.58	1954.30	0.84	10740.10	0.94
1-4-1	7.00	0.23	93.40	0.58	881.30	0.90	7572.30	1.00	58758.60	1.00
1-4-2	7.00	0.16	93.40	0.51	881.30	0.82	7553.00	0.99	58723.40	1.00
1-4-3	7.00	0.15	93.40	0.32	881.20	0.67	7571.40	0.89	58724.00	0.99
1-4-4	7.00	0.15	93.40	0.29	881.20	0.50	7565.10	0.74	58715.10	0.90
2-1-1	5.00	0.24	50.20	0.55	324.10	0.85	1948.60	0.98	10740.50	1.00
2-1-2	5.00	0.20	50.20	0.54	324.10	0.87	1950.30	1.00	10746.80	1.00
2-1-3	5.00	0.17	50.20	0.40	324.10	0.78	1951.80	0.94	10746.90	0.99
2-1-4	5.00	0.17	50.20	0.34	324.10	0.58	1948.70	0.82	10740.60	0.96
2-2-1	7.00	0.20	93.40	0.55	881.30	0.85	7556.90	1.00	58725.40	1.00
2-2-2	7.00	0.16	93.40	0.52	881.30	0.80	7563.00	0.98	58728.30	1.00
2-2-3	7.00	0.15	93.40	0.34	881.50	0.66	7560.50	0.89	58767.50	0.99
2-2-4	7.00	0.15	93.40	0.31	881.60	0.49	7562.30	0.73	58715.30	0.91

Table B.1.: FDE and Test Suite Sizes for all Test Scenarios by `IPOG-C`

Strategy	ROBUSTA							
Strength	1-0		1-1		1-2		1-3	
Scenario	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.
1-1-1	9.00	0.43	24.00	0.54	119.60	0.67	443.80	0.69
1-1-2	10.20	0.42	30.60	0.53	154.60	0.67	625.50	0.68
1-1-3	10.10	0.43	30.10	0.54	137.20	0.67	310.00	0.68
1-1-4	10.00	0.43	30.00	0.54	60.00	0.67	179.40	0.68
1-2-1	11.00	0.33	36.00	0.46	248.90	0.60	1518.70	0.64
1-2-2	12.20	0.34	42.80	0.46	292.20	0.60	1791.80	0.65
1-2-3	12.20	0.33	42.20	0.46	262.50	0.61	957.80	0.64
1-2-4	12.10	0.33	42.10	0.46	131.70	0.61	499.90	0.64
1-3-1	10.00	0.39	30.00	0.51	223.50	0.66	1243.40	0.67
1-3-2	10.30	0.38	30.90	0.50	239.70	0.65	1438.20	0.66
1-3-3	10.00	0.38	30.00	0.50	228.20	0.65	1332.10	0.66
1-3-4	10.00	0.38	30.00	0.50	214.90	0.65	1197.20	0.66
1-4-1	12.00	0.35	42.00	0.47	425.00	0.63	3641.70	0.66
1-4-2	12.20	0.35	42.60	0.46	445.50	0.64	3965.90	0.66
1-4-3	12.00	0.35	42.00	0.46	426.40	0.63	3705.50	0.66
1-4-4	12.00	0.35	42.00	0.46	404.30	0.63	3397.50	0.66
2-1-1	10.00	0.39	30.00	0.51	223.50	0.66	1243.40	0.67
2-1-2	10.30	0.39	30.90	0.51	239.70	0.66	1438.20	0.67
2-1-3	10.00	0.39	30.00	0.51	228.20	0.66	1332.10	0.67
2-1-4	10.00	0.39	30.00	0.51	214.90	0.66	1197.20	0.67
2-2-1	12.00	0.34	42.00	0.45	425.00	0.62	3641.70	0.65
2-2-2	12.20	0.34	42.60	0.45	445.50	0.63	3965.90	0.65
2-2-3	12.00	0.34	42.00	0.45	426.40	0.62	3705.50	0.65
2-2-4	12.00	0.34	42.00	0.45	404.30	0.62	3397.50	0.65

Table B.2.: FDE and Test Suite Sizes for all Test Scenarios by ROBUSTA with $t = 1$

Strategy	ROBUSTA							
Strength	2-0		2-1		2-2		2-3	
Scenario	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.
1-1-1	30.40	0.57	45.40	0.68	141.00	0.81	465.20	0.82
1-1-2	41.60	0.58	62.00	0.69	186.00	0.82	656.90	0.83
1-1-3	42.00	0.57	62.00	0.68	169.10	0.81	341.90	0.82
1-1-4	41.70	0.57	61.70	0.68	91.70	0.80	211.10	0.82
1-2-1	58.40	0.48	83.40	0.61	296.30	0.75	1566.10	0.79
1-2-2	73.90	0.48	104.50	0.61	353.90	0.75	1853.50	0.79
1-2-3	74.70	0.48	104.70	0.61	325.00	0.76	1020.30	0.80
1-2-4	75.20	0.48	105.20	0.61	194.80	0.76	563.00	0.79
1-3-1	49.30	0.58	69.30	0.71	262.80	0.86	1282.70	0.87
1-3-2	56.30	0.55	76.90	0.68	285.70	0.83	1484.20	0.84
1-3-3	55.50	0.56	75.50	0.68	273.70	0.83	1377.60	0.84
1-3-4	55.60	0.56	75.60	0.68	260.50	0.83	1242.80	0.84
1-4-1	89.80	0.51	119.80	0.62	502.80	0.78	3719.50	0.82
1-4-2	98.30	0.50	128.70	0.61	531.60	0.79	4052.00	0.81
1-4-3	99.10	0.48	129.10	0.59	513.50	0.76	3792.60	0.79
1-4-4	98.40	0.48	128.40	0.59	490.70	0.77	3483.90	0.79
2-1-1	49.30	0.54	69.30	0.66	262.80	0.81	1282.70	0.82
2-1-2	56.30	0.54	76.90	0.66	285.70	0.82	1484.20	0.82
2-1-3	55.50	0.53	75.50	0.66	273.70	0.81	1377.60	0.82
2-1-4	55.60	0.54	75.60	0.67	260.50	0.81	1242.80	0.82
2-2-1	89.80	0.49	119.80	0.60	502.80	0.77	3719.50	0.80
2-2-2	98.30	0.50	128.70	0.61	531.60	0.79	4052.00	0.81
2-2-3	99.10	0.51	129.10	0.62	513.50	0.79	3792.60	0.81
2-2-4	98.40	0.50	128.40	0.61	490.70	0.78	3483.90	0.81

Table B.3.: FDE and Test Suite Sizes for all Test Scenarios by ROBUSTA with $t = 2$

Strategy	ROBUSTA							
Strength	**3-0**		**3-1**		**3-2**		**3-3**	
Scenario	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**	**Size**	**Eff.**
1-1-1	109.40	0.69	124.40	0.80	220.00	0.93	544.20	0.95
1-1-2	199.10	0.70	219.50	0.81	343.50	0.94	814.40	0.95
1-1-3	208.10	0.69	228.10	0.80	335.20	0.93	508.00	0.95
1-1-4	205.20	0.70	225.20	0.81	255.20	0.93	374.60	0.95
1-2-1	365.20	0.62	390.20	0.75	603.10	0.89	1872.90	0.93
1-2-2	546.70	0.61	577.30	0.74	826.70	0.88	2326.30	0.92
1-2-3	559.00	0.61	589.00	0.73	809.30	0.88	1504.60	0.92
1-2-4	556.20	0.61	586.20	0.73	675.80	0.88	1044.00	0.92
1-3-1	259.50	0.68	279.50	0.81	473.00	0.96	1492.90	0.97
1-3-2	329.40	0.68	350.00	0.80	558.80	0.96	1757.30	0.96
1-3-3	328.80	0.68	348.80	0.80	547.00	0.95	1650.90	0.96
1-3-4	327.70	0.67	347.70	0.79	532.60	0.93	1514.90	0.95
1-4-1	748.90	0.66	778.90	0.77	1161.90	0.93	4378.60	0.96
1-4-2	885.50	0.64	915.90	0.76	1318.80	0.93	4839.20	0.95
1-4-3	885.80	0.64	915.80	0.75	1300.20	0.92	4579.30	0.94
1-4-4	886.40	0.63	916.40	0.74	1278.70	0.92	4271.90	0.94
2-1-1	259.50	0.67	279.50	0.79	473.00	0.94	1492.90	0.95
2-1-2	329.40	0.68	350.00	0.80	558.80	0.96	1757.30	0.96
2-1-3	328.80	0.67	348.80	0.80	547.00	0.94	1650.90	0.96
2-1-4	327.70	0.68	347.70	0.80	532.60	0.94	1514.90	0.96
2-2-1	748.60	0.63	778.60	0.74	1161.60	0.91	4378.30	0.94
2-2-2	885.40	0.62	915.80	0.74	1318.70	0.92	4839.10	0.93
2-2-3	885.90	0.64	915.90	0.75	1300.30	0.92	4579.40	0.94
2-2-4	886.60	0.63	916.60	0.74	1278.90	0.91	4272.10	0.93

Table B.4.: FDE and Test Suite Sizes for all Test Scenarios by ROBUSTA with $t = 3$

Strategy	ROBUSTA							
Strength	4-0		4-1		4-2		4-3	
Scenario	Size	Eff.	Size	Eff.	Size	Eff.	Size	Eff.
1-1-1	402.30	0.74	417.30	0.86	512.90	0.98	837.10	1.00
1-1-2	844.70	0.74	865.10	0.86	989.10	0.99	1460.00	1.00
1-1-3	916.20	0.74	936.20	0.86	1043.30	0.98	1216.10	1.00
1-1-4	922.90	0.74	942.90	0.86	972.90	0.98	1092.30	1.00
1-2-1	2040.50	0.69	2065.50	0.82	2278.40	0.96	3548.20	1.00
1-2-2	3461.90	0.69	3492.50	0.82	3741.90	0.96	5241.50	1.00
1-2-3	3617.10	0.69	3647.10	0.82	3867.40	0.97	4562.70	1.00
1-2-4	3615.10	0.69	3645.10	0.82	3734.70	0.97	4102.90	1.00
1-3-1	1366.50	0.72	1386.50	0.84	1580.00	0.99	2599.90	1.00
1-3-2	1949.60	0.72	1970.20	0.84	2179.00	0.99	3377.50	1.00
1-3-3	1961.90	0.72	1981.90	0.84	2180.10	0.99	3284.00	1.00
1-3-4	1957.20	0.72	1977.20	0.84	2162.10	0.98	3144.40	1.00
1-4-1	5894.80	0.69	5924.80	0.81	6307.80	0.97	9524.50	1.00
1-4-2	7536.10	0.69	7566.50	0.81	7969.40	0.98	11489.80	1.00
1-4-3	7589.40	0.69	7619.40	0.81	8003.80	0.97	11282.90	1.00
1-4-4	7578.30	0.69	7608.30	0.81	7970.60	0.98	10963.80	1.00
2-1-1	1368.60	0.72	1388.60	0.84	1582.10	0.99	2602.00	1.00
2-1-2	1944.60	0.72	1965.20	0.84	2174.00	0.99	3372.50	1.00
2-1-3	1965.60	0.72	1985.60	0.84	2183.80	0.99	3287.70	1.00
2-1-4	1955.50	0.72	1975.50	0.84	2160.40	0.98	3142.70	1.00
2-2-1	5898.50	0.69	5928.50	0.81	6311.50	0.97	9528.20	1.00
2-2-2	7542.90	0.69	7573.30	0.81	7976.20	0.98	11496.60	1.00
2-2-3	7586.60	0.69	7616.60	0.81	8001.00	0.97	11280.10	1.00
2-2-4	7560.70	0.69	7590.70	0.81	7953.00	0.98	10946.20	1.00

Table B.5.: FDE and Test Suite Sizes for all Test Scenarios by ROBUSTA with $t = 4$

Appendix C.

Data of Evaluating Semi-Automatic Repair of Over-Constrained RIPMs

This is the appendix to Section 14.2 (Page 249 and following) which contains all results for all RIPMs used in the experiment.

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	8	0	1	0	0
2	**7**	**0**	**0**	**0**	**0**

Table C.1.: Results for Example-1 RIPM

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	29	0	1	1	5
2	29	0	1	1	5
3	25	4	1	0	7
4	29	0	1	1	7
5	30	0	1	1	7
6	**29**	**0**	**0**	**0**	**0**
7	30	0	1	1	7
8	30	0	1	1	7
9	30	0	1	1	5
10	23	4	1	1	6
11	23	4	1	2	7
12	23	4	1	1	6
13	23	4	1	2	7
14	19	9	1	0	4
15	24	4	1	1	6
16	24	4	1	1	6

Table C.2.: Results for HealthCare-4 RIPM

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	39	0	2	3	0
2	42	0	3	4	**0**
3	36	1	1	3	0
4	39	0	2	3	0
5	36	0	1	2	0
6	39	0	2	3	0
7	34	0	0	1	0
8	36	0	1	2	0
9	39	0	2	4	0
10	42	0	3	5	0
11	36	0	1	3	0
12	39	0	2	4	0
13	36	0	1	3	0
14	39	0	2	4	0
15	34	0	0	2	0
16	36	0	1	3	0
17	39	0	2	2	0
18	42	0	2	2	0
19	36	0	1	1	0
20	39	0	2	2	0
21	36	0	1	1	0
22	39	0	2	2	0
23	**34**	**0**	**0**	**0**	**0**
24	36	0	1	1	0
25	39	0	2	3	0
26	42	0	3	4	0
27	36	0	1	2	0
28	39	0	2	3	0
29	36	0	1	2	0
30	39	0	2	3	0
31	34	0	0	1	0
32	36	0	1	2	0

Table C.3.: Results for Registration RIPM

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	110	8	0	0	0
2	95	24	0	0	0
3	113	4	0	0	0
4	98	20	0	0	0
5	113	4	0	0	0
6	98	20	0	0	0
7	**116**	**0**	**0**	**0**	**0**
8	101	16	0	0	0
9	107	12	0	0	0
10	92	28	0	0	0
11	110	8	0	0	0
12	95	24	0	0	0
13	110	8	0	0	0
14	95	24	0	0	0
15	113	4	0	0	0
16	98	20	0	0	0

Table C.4.: Results for Banking-1 RIPM

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	9	0	1	0	0
2	8	1	1	0	0
3	**8**	**0**	**0**	**0**	**0**
4	7	1	0	0	0

Table C.5.: Results for Banking-2 RIPM

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	**27**	**0**	**0**	**0**	**0**
2	22	6	0	0	0
3	27	1	0	0	0
4	22	7	0	0	0

Table C.6.: Results for HealthCare-1 RIPM

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	**33**	**0**	**0**	**0**	**0**
2	**33**	**0**	**0**	**0**	**0**
3	**33**	**0**	**0**	**0**	**0**
4	**29**	**0**	**0**	**0**	**0**
5	32	2	0	0	0
6	32	2	0	0	0
7	32	2	0	0	0
8	32	2	0	0	0

Table C.7.: Results for HealthCare-2 RIPM

$\triangle_{HS}$	Test Suite Size	NIS	DIS	NVTI	NSITI
1	36	0	1	2	6
2	**37**	**0**	**0**	**0**	**0**
3	36	0	1	2	6
4	28	8	1	0	2
5	28	8	1	0	2
6	30	6	2	1	4
7	30	6	2	1	4
8	30	6	2	1	4
9	30	6	1	1	3
10	30	6	1	1	3

Table C.8.: Results for HealthCare-3 RIPM

Appendix D.

Data of Evaluating Automatic Generation of Error-Constraints

This is the appendix to Section 15 (Page 259 and following) which contains test suites sizes, execution times, precision, and recall for all test scenarios used in the experiment.

Scenario	Algorithm	Test Suite Size	Execution Time	Precision	Recall
Example-1	OFOT	22.700	0.270	0.350	0.180
	TRT	27.000	0.095	1.000	1.000
	IDD	13.000	0.003	0.750	0.600
	MixTgTe	19.300	0.007	0.823	0.840
Delivery	OFOT	252.400	1.474	0.581	0.162
	TRT	863.800	1.317	0.980	0.923
	IDD	67.000	0.007	0.000	0.000
	MixTgTe	172.600	0.165	0.894	0.392
BibTeX	OFOT	711.700	15.420	0.094	0.028
	TRT	15722.400	10463.000	1.000	0.957
	IDD	73.000	0.012	0.038	0.006
	MixTgTe	226.000	122.000	0.706	0.213
Banking-1	OFOT	49.400	0.277	1.000	0.033
	TRT	39.000	0.141	1.000	0.032
	IDD	33.000	0.006	0.000	0.000
	MixTgTe	28.000	0.010	0.000	0.000
Banking-2	OFOT	121.700	1.167	1.000	1.000
	TRT	4141.800	15.137	1.000	1.000
	IDD	34.000	0.005	0.500	0.333
	MixTgTe	35.000	0.075	1.000	1.000
HealthCare-1	OFOT	172.400	2.870	0.307	0.114
	TRT	2348.700	3.995	1.000	0.757
	IDD	45.200	0.005	0.000	0.000
	MixTgTe	122.500	1.753	0.583	0.271
HealthCare-2	OFOT	174.600	2.133	0.436	0.032
	TRT	1633.900	38.283	1.000	0.098
	IDD	28.200	0.004	0.000	0.000
	MixTgTe	63.900	0.519	0.152	0.015
HealthCare-3	OFOT	54.300	3.501	0.000	0.000
	TRT	85104.400	173.000	0.475	0.873
	IDD	46.000	0.012	0.00	0.000
	MixTgTe	341.300	4184.000	0.00	0.000

Table D.1.: Average Values for Experiment 1

Scenario	Algorithm	Test Suite Size	Execution Time	Precision	Recall
Example-1	OFOT	23.200	0.166	0.700	0.220
	TRT	27.000	0.089	1.000	1.000
	IDD	13.000	0.004	0.850	0.600
	MixTgTe	20.600	0.0081	1.000	1.000
Delivery	OFOT	273.300	1.670	0.678	0.162
	TRT	825.500	1.332	0.971	0.800
	IDD	67.000	0.007	0.000	0.000
	MixTgTe	182.600	0.199	0.940	0.450
BibTeX	OFOT	682.900	15.164	0.072	0.011
	TRT	16019.700	10655.000	1.000	0.978
	IDD	76.200	0.385	0.091	0.011
	MixTgTe	224.700	103.000	0.729	0.209
Banking-1	OFOT	48.200	0.255	1.000	0.031
	TRT	40.300	0.146	1.000	0.034
	IDD	33.000	0.035	0.000	0.000
	MixTgTe	31.200	0.012	0.000	0.000
Banking-2	OFOT	127.100	1.194	1.000	1.000
	TRT	4127.400	15.364	1.000	1.000
	IDD	34.000	0.006	0.550	0.333
	MixTgTe	32.900	0.064	1.000	1.000
HealthCare-1	OFOT	203.100	2.360	0.779	0.143
	TRT	2133.100	4.810	1.000	0.621
	IDD	45.200	0.007	0.000	0.000
	MixTgTe	122.300	1.579	0.765	0.350
HealthCare-2	OFOT	185.500	2.806	0.522	0.037
	TRT	1844.200	142.000	1.000	0.108
	IDD	29.400	0.005	0.000	0.000
	MixTgTe	67.100	1.143	0.314	0.017
HealthCare-3	OFOT	63.500	10.116	0.000	0.000
	TRT	86612.200	1128.000	1.000	0.420
	IDD	46.000	0.008	0.000	0.000
	MixTgTe	346.000	8553.515	0.000	0.000

Table D.2.: Average Values for Experiment 2

Scenario	Algorithm	Test Suite Size	Execution Time	Precision	Recall
Example-1	OFOT	24.800	0.163	0.600	0.400
	TRT	27.000	0.091	1.000	1.000
	IDD	27.000	0.006	0.600	0.600
	MixTgTe	27.000	0.021	1.000	1.000
Delivery	OFOT	360.800	13.255	0.500	0.115
	TRT	991.400	12.766	1.000	1.000
	IDD	287.000	0.108	0.000	0.000
	MixTgTe	446.800	4.018	0.985	0.761
BibTeX	OFOT	789.300	257.000	0.219	0.066
	TRT	15838.300	10124.000	1.000	1.000
	IDD	262.000	2.793	0.000	0.000
	MixTgTe	906.000	7180.000	0.995	0.921
Banking-1	OFOT	125.000	1.139	1.000	0.095
	TRT	108.000	0.645	1.000	0.096
	IDD	70.700	0.062	0.000	0.000
	MixTgTe	89.400	0.091	0.000	0.000
Banking-2	OFOT	146.900	19.204	1.000	1.000
	TRT	4141.500	18.838	1.000	1.000
	IDD	51.000	0.058	0.000	0.000
	MixTgTe	95.900	3.147	1.000	1.000
HealthCare-1	OFOT	273.200	40.419	0.550	0.294
	TRT	2682.800	18.174	1.000	0.929
	IDD	128.300	19.979	0.000	0.000
	MixTgTe	384.200	45.725	0.954	0.657
HealthCare-2	OFOT	376.800	59.734	0.432	0.058
	TRT	4537.900	218.000	1.000	0.253
	IDD	130.500	0.250	0.202	0.014
	MixTgTe	243.500	31.784	0.694	0.095
HealthCare-3	OFOT	83.000	375.000	0.100	0.002
	TRT	95210.200	6503.000	1.000	0.447
	IDD	243.000	0.028	0.000	0.000
	MixTgTe	-	-	-	-

Table D.3.: Average Values for Experiment 3

Abbreviations

A

AFDE average fault detection effectiveness
API   application programming interface

B

B-SCH basic soft constraint handling
BVT   boundary value testing

C

CRT   combinatorial robustness testing
CSP   constraint satisfaction problem
CT   combinatorial testing

D

D-SCH diagnostic soft constraint handling
DIS   duplicate invalid schemata
DOE   design of experiments
DSL   domain-specific language

E

ECT   equivalence class testing
EH   exception handling
EHM   exception handling mechanism

F

FC   fault characterization
FCA   fault characterization algorithm
FDE   fault detection effectiveness

H

HCH   hard constraint handling
HS-tree hitting set tree

I

IDE   integrated development environment
IPM   input parameter model
IPO   input parameter order

L

LNCS   Springer Lecture Notes in Computer Science

M

MFCS   minimal failure-causing schema
MIS   missing invalid schema

N

NIS   not present invalid schemata
NSITI   not strong invalid test input
NVTI   not valid test input

R

RIPM   robustness input parameter model

S

SCH   soft constraint handling
SUT   system under test

T

TSE   IEEE Transactions on Software Engineering

X

XML   extensible markup language

Symbols and Functions

A

α	assignment of a CSP
a	robustness degree
A	set of all schemata
$AI \subseteq I$	anticipated input domain
$\texttt{AllInteractions}(c_i, b)$	function to compute the sets of parameter indices with b-wise robustness interaction for c_i
$\texttt{AllSubsets}(d)$	function to compute all d-sized subsets of parameter indices for ϕ_n

B

b	robustness interaction degree
$\mathcal{B}$	background constraints

C

c	constraint
$c_{i,j}$	schema-level constraint for $s_j \in L_{c_i}$
$\mathcal{C}$	set of relaxable constraints
$C \in C^*$	set of constraints
C^*	set of all possible constraint sets
C^{ex}	set of exclusion-constraints
C^{err}	set of error-constraints
$\texttt{Complement}(\phi)$	function to compute the complement of ϕ_d relative to ϕ_n

D

$d = |\tau|$	degree of schema τ
$\texttt{Degree}(c_i)$	function to compute the number of parameters involved in the condition of error-constraint
$\triangle_{c_{i,j}}$	diagnosis set for MIS $s_j \in M_{c_i}$
$\triangle_{c_{i,j}}^{all}$	set of all minimal diagnosis sets for MIS $s_j \in M_{c_i}$
$\triangle^{all}$	Cartesian product of all sets of all minimal diagnosis sets
$\triangle_{HS}$	diagnosis hitting set

E

η	threshold for the utility of soft-constraints
$e \in \triangle^{all}$	tuple with one minimal diagnosis set for each MIS
$E \subseteq I$	exhaustive test suite
$EI \subset AI$	exception input domain
ES	exception specification
$\mathtt{Exclude}(c_i, s_j)$	function to exclude schema $s_j \in L_{c_i}$ from constraint c_i

H

$\mathcal{H}$	set of hard-constraints

I

I	input domain
I_{c_i}	set of all strong invalid schemata specified by error-constraint c_i
$IPM = \langle \tilde{P}, V, C^{ex} \rangle$	input parameter model
$\mathtt{Indices}(c_i)$	function to compute the set of parameter indices ϕ for parameters involved in the condition of error-constraint c_i
$\mathtt{Interactions}(c_i, a, b)$	function to compute the a- and b-wise parameter index interactions for error-constraint c_i
$\mathtt{IsInvalid}(s)$	predicate to check if schema s is invalid
$\mathtt{IsMinimalInvalid}(s)$	predicate to check if schema s is minimal invalid
$\mathtt{IsRelevant}(s)$	predicate to check if schema s is relevant
$\mathtt{IsStrongInvalid}(s)$	predicate to check if schema s is strong invalid
$\mathtt{IsStrongInvalid}(s, c_i)$	predicate to check if schema s is strong invalid for error-constraint c_i
$\mathtt{IsValid}(s)$	predicate to check if schema s is valid

L

l	location of an incorrect EH
L_{c_i}	set of invalid schemata locally specified by error-constraint c_i

M

m_i	number of values modeled by V_i
$M_{c_i} \subseteq L_{c_i}$	set of missing invalid schemata for error-constraint c_i

N

n	number of parameters modeled by an IPM

O

$o \in O$	output
$\mathcal{O}_{c_i,j}$	conflict set for MIS $s_j \in L_{c_i}$
O	output domain

$OK(\tau, P(\tau))$	passing test input τ
$\neg OK(\tau, P(\tau))$	failure-causing test input

P

$\pi = (p_i, v_j)$	parameter-value pair π with $v_j \in V_i$ for parameter p_i parameter value pair
$\phi = \{r_1, ..., r_d\}$	set of $d = |\phi|$ parameter indices
$\phi_n = \{r_1 = 1, ..., r_n = n\}$	set of all n parameter indices
Φ_d	set of d-sized parameter index sets
$p_i \in \tilde{P}$	parameter with label i ($0 \le i \le n$)
$\tilde{P} = \{p_1, ..., p_n\}$	set of all parameters modeled by an IPM
$P : I \rightarrow O$	program
$P(\tau)$	execution of program P
$\texttt{Pairs}(r)$	function to compute all parameter-value pairs of parameter with index r

R

$r \in [1, n]$	parameter index
$\mathcal{R} \subset \mathcal{C}$	relaxation
$RIPM = \langle \tilde{P}, V, C^{ex}, C^{err} \rangle$	input parameter model
$\texttt{RemainingSubsets}(c_i, b)$	function to compute the set of all sets of b-sized remaining parameter indices

S

$\texttt{SAT} : C^* \times A \rightarrow Bool$	satisfiability function
$s = \{(p_i, v_j), ...\}$	schema (alternative symbol)
$\mathcal{S}$	set of soft-constraints
$(\tau, o) \in S$	specified output $o \in O$ for truly-relevant input $\tau \in AI$
$S \subseteq I \times O$	specification
SC	test scenario
SC^*	test scenario family
$SI \subseteq AI$	standard input domain
STS	standard specification
$\texttt{SpecifiedSubsets}(c_i, a)$	function to compute the set of all sets of a-sized parameter indices for $\texttt{Indices}(c_i)$
$\texttt{Subsets}(\phi, d)$	function to compute all d-sized subsets of parameter indices for ϕ

T

$\tau = \{(p_i, v_j), ...\}$	schema
$\tau \in AI$	truly-relevant input
$\tau \in E$	test input
$\tau \in EI$	truly-invalid input

$\tau \in I$	input
$\tau \in SI$	truly-valid input
$\tau \in UI$	truly-irrelevant input
$\tau_a \subseteq \tau_b$	coverage relation between two schemata
t	testing strength
$T \subseteq E$	test suite
T^*	test suite family

U

$\mathcal{U}(\alpha)$	function to compute the utility of soft-constraints
$UI \subset I$	unanticipated input domain

V

$v_j \in V_i$	value with a unique index j of parameter p_i
$V = \{V_1, ..., V_n\}$	set of all value domains of all parameters
V_i	domain of values associated with parameter p_i

X

$X_{c_i,a}^{invalid}$	set of sets of all schemata $X_{c_i,\phi}^{invalid}$ for all $\phi \in \texttt{SpecifiedSubsets}(c_i, a)$
$X_{c_i,a,b}^{invalid}$	set of sets of all schemata $X_{c_i,\phi}^{invalid}$ for all $\phi \in \texttt{Interactions}(c_i, a, b)$
$X_{c_i,b}^{invalid}$	set of sets of all strong invalid schemata $X_{c_i,\phi}^{invalid}$ for all $\phi \in \texttt{AllInteractions}(c_i, b)$
$X_{\texttt{Indices}(c_i)}$	set of all schemata defined by error-constraint c_i
$X_{c_i,\phi}^{invalid}$	set of all strong invalid schemata for parameters in ϕ and error-constraint c_i
X_d	set of sets of all schemata X_ϕ for all $\phi \in \texttt{AllSubsets}(d)$
$X_d^{relevant}$	set of sets of all relevant schemata $X_\phi^{relevant}$ for all $\phi \in \texttt{AllSubsets}(d)$
X_d^{valid}	set of sets of all valid schemata X_ϕ^{valid} for all $\phi \in \texttt{AllSubsets}(d)$
X_ϕ	set of all d-sized schemata of parameters indexed by ϕ
$X_\phi^{relevant}$	set of all relevant d-sized schemata of parameters indexed by ϕ
X_ϕ^{valid}	set of all valid d-sized schemata of parameters indexed by ϕ
$X_{s,b}^{invalid}$	set of all strong invalid schemata for $s \in I_{c_i}$

Glossary

A

anticipated input domain

> The anticipated input domain denotes a subset of the input domain for which some output is defined by the specification. See Definition 18 on Page 23.

C

cardinality-minimal diagnosis hitting set

> A diagnosis hitting set is cardinality-minimal if and only if there exists no diagnosis hitting set that contains fewer constraints. See Definition 107 on Page 173.

combinatorial robustness testing (CRT)

> Combinatorial robustness testing (CRT) is an extension of CT with the objective to separate valid and invalid test inputs such that all values and value combinations are tested as expected.

combinatorial testing (CT)

> Combinatorial testing (CT) is a specification-based test method where the selection of test inputs is based on an input parameter model.

conflict

> A conflict is a contradiction between a negated error-constraint and some other error- or exclusion-constraints. See Definition 98 on Page 162.

conflict set

> A conflict set is a set of constraints that explains the absence of a missing invalid schema. See Definition 101 on Page 167.

correct program

> A program is a correct program if and only if its behavior is consistent with its standard specification entire standard input domain. See Definition 59 on Page 40.

D

dependable program

> A program is a dependable program if and only if its behavior is consistent with its specification. See Definition 17 on Page 22.

diagnosis hitting set

A diagnosis hitting set is a set of constraints such that the relaxation of all its constraints removes all conflicts of the RIPM. See Definition 105 on Page 172.

diagnosis set

A diagnosis set is a set of constraints such that relaxing all conflicts of a diagnosis set removes all conflicts between an error-constraint and some other constraints. See Definition 103 on Page 170.

E

error-constraints

The set of error-constraints is a set of constraints to distinguish valid from invalid schemata. See Definition 87 on Page 111.

exception

An exception is a characterization of abnormal or unexpected conditions that can appear in a program's state and may require extraordinary computation. See Definition 67 on Page 43.

exception handling (EH)

Exception handling (EH) is an umbrella term that encompasses all activities of detecting exception occurrences and providing appropriate responses for recovery. See Definition 70 on Page 44.

exception input domain

The exception input domain covers inputs and intended outputs for all external faults that are anticipated by the exception specification. See Definition 56 on Page 40.

exception specification

The exception specification defines the functions of programs for exceptional situations with external faults. See Definition 52 on Page 39.

exclusion-constraints

The set of exclusion-constraints is a set of constraints to distinguish relevant from irrelevant schemata. See Definition 45 on Page 32.

F

failure-causing input

A truly-relevant input is a failure-causing input if and only if the output of the program does not match the intended output of the specification.. See Definition 22 on Page 24.

failure-causing schema

A truly-relevant schema is a failure-causing schema if and only if each truly-relevant input that covers the schema is failure-causing. See Definition 39 on Page 30.

failure-causing test input

A test input for which the test oracle determines that the output of the program does not conform to the specification is a failure-causing test input. Also refer to *failure-causing input*.

fault detection effectiveness (FDE)

Fault detection effectiveness (FDE) is a common metric to evaluate test selection strategies. It is defined as the ratio of the number of test suites that fail and the number of all test suites.

fault model

A fault model is a description of hypothesized faults and how to activate them. See Definition 32 on Page 27.

H

hard constraint handling

Hard constraint handling is a strategy for transforming a RIPM and a schema into a CSP in which all constraints must be satisfied.

I

input

An input is an abstraction that may encompass anything explicable that can actually stimulate or affect the behavior of a program and anything explicable that can be expected to stimulate or affect the behavior of a program. See Definition 13 on Page 21.

input domain

The input domain denotes the set of all possible inputs. See Definition 12 on Page 21.

input masking effect

The input masking effect is an effect that prevents a test input from testing all combinations of input values, which the test input is normally expected to test. See Definition 42 on Page 31.

input parameter model (IPM)

An input parameter model (IPM) is a specification-based model of the input domain that structures the input domain via parameters, associated value domains, and constraints. See Definition 34 on Page 28.

intended output

The intended output is the expected output as defined by a specification for a truly-relevant input.

invalid schema

A relevant schema is an invalid schema if and only if all exclusion-constraints are satisfied but at least one error-constraint is not satisfied. See Definition 90 on Page 112.

irrelevant schema

A schema is an irrelevant schema if and only if at least one exclusion-constraint is not satisfied. See Definition 46 on Page 33.

L

locally-specified schema

A locally-specified schema is *specified* by an error-constraint, i.e. it satisfies the error-constraint and covers a parameter-value pair for each parameter of the error-constraint.

M

minimal conflict set

A conflict set is minimal if and only if no proper subset is a conflict set. See Definition 102 on Page 167.

minimal diagnosis hitting set

A diagnosis hitting set is minimal if and only if no proper subset is a diagnosis hitting set. See Definition 106 on Page 173.

minimal diagnosis set

A diagnosis set is minimal if and only if no proper subset is a diagnosis set. See Definition 104 on Page 170.

minimal failure-causing schema (MFCS)

A failure-causing schema is a MFCS if and only if no proper subset schema exists that is itself a failure-causing schema. See Definition 40 on Page 30.

minimal invalid schema

A strong invalid schema is a minimal invalid schema if and only if no proper subset schema exists that is itself an invalid schema. See Definition 92 on Page 113.

missing invalid schema (MIS)

A locally-specified schema for an error-constraint is a MIS if a conflict with some other constraints prevents it from appearing in any strong invalid test input. See Definition 99 on Page 162.

O

output

> An output encompasses anything explicable that can be actually observed after a program has been stimulated and anything explicable that can be expected to be observed after a program has been stimulated. See Definition 15 on Page 21.

output domain

> The output domain denotes the set of all possible outputs. See Definition 14 on Page 21.

P

program

> A program is a combination of computer instructions and data definitions that enables hardware to perform computational or control functions. See Definition 11 on Page 19.

R

relevant schema

> A schema is a relevant schema if and only if it satisfies all exclusion-constraints. See Definition 47 on Page 33.

robust program

> A program is a robust program if and only if its behavior is consistent with its exception specification for the entire exception input domain. See Definition 60 on Page 41.

robustness

> Robustness is the degree to which a program behaves correctly in the presence of invalid inputs or stressful environmental conditions.

robustness degree

> The robustness degree of a minimal failure-causing truly-invalid schema is the number of parameter-value pairs that constitute the truly-minimal invalid sub-schema. See Definition 85 on Page 107.

robustness input parameter model (RIPM)

> The robustness input parameter model (RIPM) is an extension of the IPM structure which includes the separate set of error-constraints. See Definition 88 on Page 111.

robustness interaction degree

> The robustness interaction degree of a minimal failure-causing truly-invalid schema is defined by the number of parameter-value pairs that do not constitute the truly-minimal invalid sub-schema. See Definition 86 on Page 108.

S

schema

A schema is a set of parameter-value pairs for d $(1 \leq d \leq n)$ distinct parameters. See Definition 36 on Page 29.

soft constraint handling

Soft constraint handling is a strategy for transforming a RIPM and a schema into a CSP in which as many constraints as possible should be satisfied.

specification

A specification describes the functions that a program must realize and constraints its development and execution. See Definition 16 on Page 22.

standard input domain

The standard input domain encompasses all inputs for which intended outputs are described by the standard specification. See Definition 55 on Page 40.

standard specification

The standard specification defines the functions of programs for standard situations without external faults. See Definition 51 on Page 39.

strong invalid schema

An invalid schema is a strong invalid schema if and only if exactly one error-constraint remains unsatisfied. See Definition 91 on Page 113.

system under test (SUT)

Programs and sub-systems of programs are also called SUT when they are the subject under investigation.

T

test adequacy criterion

A test adequacy criterion is a predicate that specifies conditions a test suite mist satisfy. See Definition 31 on Page 27.

test input

A test input is a schema with degree $d = n$.

test oracle

A test oracle is a mechanism that checks the observed outputs of a test and decides whether the outputs match the indented outputs. See Definition 30 on Page 26.

test selection strategy

A test selection strategy is a technique that describes how to systematically select a test suite such that one or more test adequacy criteria are satisfied. See Definition 33 on Page 28.

truly-invalid input

An truly-relevant input is truly-invalid if and only if it belongs to the exception input domain. See Definition 58 on Page 40.

truly-invalid schema

A schema is truly-invalid if and only if no truly-valid input exists exist that covers the schema and at least one truly-invalid test input exists that covers the schema. See Definition 74 on Page 50.

truly-irrelevant input

An input is truly-irrelevant if and only if it is not anticipated by the specification. See Definition 21 on Page 24.

truly-irrelevant schema

A schema is truly-irrelevant if and only if no truly-relevant input exists that covers the schema. See Definition 43 on Page 31.

truly-minimal invalid input

A truly-invalid schema is a truly-minimal invalid schema if and only if no proper subset schema $s' \subset s$ exists that is itself a truly-invalid schema. See Definition 75 on Page 50.

truly-relevant input

An input is truly-relevant if and only if it is anticipated by the specification. See Definition 20 on Page 23.

truly-relevant schema

A schema is truly-relevant if and only if at least one truly-relevant input exists that covers the schema. See Definition 44 on Page 31.

truly-strong invalid input

A truly-invalid schema is a truly-strong invalid schema if and only if it covers exactly one truly-minimal invalid schema. See Definition 76 on Page 51.

truly-valid input

A truly-relevant input is truly-valid if and only if it belongs to the standard input domain. See Definition 57 on Page 40.

truly-valid schema

   A schema is truly-valid if and only if at least one truly-valid input exist that covers the schema. See Definition 73 on Page 50.

U

unanticipated input domain

   The unanticipated input domain denotes a subset of the input domain for which no output is defined by the specification. See Definition 19 on Page 23.

V

valid schema

   A relevant schema is a valid schema if and only if it satisfies all exclusion- and error-constraints. See Definition 89 on Page 112.

List of Figures

List of Listings

List of Tables

Supervised Bachelor and Master Theses

[Eki16]   Kübra Ekici. "Analysis of Metrics for outsourced Software Testing". Bachelor Thesis. RWTH Aachen University, May 2016.

[Lob16]   Ekaterina Lobanova. "A Model-based Approach to Synthetic Test Data Generation". Master Thesis. RWTH Aachen University, Sept. 2016.

[Sch17]   Daniel Schiller. "Semi-automatic Domain Model-based Test Data Generation". Master Thesis. RWTH Aachen University, Feb. 2017.

[Ver17]   Tatiana Vert. "A Domain Model-based Approach to Test Data Selection". Master Thesis. RWTH Aachen University, Mar. 2017.

[Man17]   Andre Mann. "Domain Model-based Test Data Generation with Expected Values". Master Thesis. RWTH Aachen University, Sept. 2017.

[Wan17]   Shuo Wang. "Domain Model Variants for Effective Test Data Design". Master Thesis. RWTH Aachen University, Oct. 2017.

[Fab18]   Soosaithasan Fabian. "Reconstructing Input Parameter Models from Combinatorial Test Suites". Bachelor Thesis. RWTH Aachen University, June 2018.

[Oss18]   Stefan Ossendorf. "Model-based Test Script Generation". Master Thesis. RWTH Aachen University, June 2018.

[Bon18]   Joshua Bonn. "Automated Fault Localization for Combinatorial Testing". Bachelor Thesis. RWTH Aachen University, Sept. 2018.

[Kir19]   Ivan Kirchev. "Java-based Combinatorial Test Modeling". Bachelor Thesis. RWTH Aachen University, May 2019.

[Off19]   Raoul Offizier. "State of the Art in Defensive Programming". Bachelor Thesis. RWTH Aachen University, Aug. 2019.

[Fri19]   Torben Friedrichs. "A Benchmark Infrastructure for Fault Characterization in Combinatorial Testing". Master Thesis. RWTH Aachen University, Sept. 2019.

[Pro19]   Kateryna Prokhorova. "A Framework for Cost-aware Combinatorial Testing". Master Thesis. RWTH Aachen University, Sept. 2019.

[Ber19]   Lukas Bernwald. "Development of an Automated Combinatorial Testing Framework". Bachelor Thesis. RWTH Aachen University, Dec. 2019.

[Maß20]   Eric Maßelter. "Generating Error-Constraints for Combinatorial Robustness Testing". Master Thesis. RWTH Aachen University, Mar. 2020.

[Bur20]   Semih Burak. "Constructing Input Parameter Models from Sample Data". Bachelor Thesis. RWTH Aachen University, Apr. 2020.

[Hop20]   Josef Hoppe. "An Infrastructure for Mutation-based Evaluation of Testing Strate-
           gies". Bachelor Thesis. RWTH Aachen University, Apr. 2020.

[Sch20]   Alexander Schnackenberg. "Development of a Web-based Combinatorial Testing
           Framework". Master Thesis. RWTH Aachen University, Apr. 2020.

[Bon20]   Joshua Bonn. "Test Case Prioritization for Combinatorial Testing". Master Thesis.
           RWTH Aachen University, Sept. 2020.

Bibliography

[AG01]   Michalis Anastasopoules and Cristina Gacek. "Implementing product line variabilities". In: *Proceedings of the ACM SIGSOFT Symposium on Software Reusability: Putting Software Reuse in Context, SSR 2001, May 18-20, 2001, Toronto, Ontario, Canada*. 2001, pp. 109–117.

[AGR19]   P. Arcaini, A. Gargantini, and M. Radavelli. "Efficient and Guaranteed Detection of t-Way Failure-Inducing Combinations". In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2019, pp. 200–209.

[AGV14]   Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. "Validation of Models and Tests for Constrained Combinatorial Interaction Testing". In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*. 2014, pp. 98–107.

[Ahm+17]   Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. "Constrained Interaction Testing: A Systematic Literature Study". In: *IEEE Access* 5 (2017), pp. 25706–25730.

[AL85]   T. Anderson and P. A. Lee. "Fault Tolerance Terminology Proposals". In: *Reliable Computer Systems: Collected Papers of the Newcastle Reliability Project*. Ed. by Santosh Kumar Shrivastava. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 6–13.

[Alm+14]   A. Almeida, E. Barreiros, J. Saraiva, F. Castor, and S. Soares. "Is Exception Handling a Reusable Aspect?" In: *2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse*. Sept. 2014, pp. 32–41.

[AO16]   Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd. New York, NY, USA: Cambridge University Press, 2016.

[AO94]   Paul Ammann and Jeff Offutt. "Using formal methods to derive test frames in category-partition testing". In: *Proceedings of COMPASS'94 - 1994 IEEE 9th Annual Conference on Computer Assurance*. 1994, pp. 69–79.

[Avi+04]   Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Trans. Dependable Sec. Comput.* 1.1 (2004), pp. 11–33.

[Bai99]   William Bail. "Exception handling design patterns". In: *Advances in Computers* 49 (1999), pp. 191–238.

[Bal09]   Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3rd. Spektrum Akademischer Verlag, 2009.

[Bar+15]   Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Trans. Software Eng.* 41.5 (2015), pp. 507–525.

[Bar+16]   E. A. Barbosa, A. Garcia, M. P. Robillard, and B. Jakobus. "Enforcing Exception Handling Policies with a Domain-Specific Language". In: *IEEE Transactions on Software Engineering* 42.6 (June 2016), pp. 559–584.

[BAZ00]   Margaret M. Burnett, Anurag Agrawal, and Pieter van Zee. "Exception Handling in the Spreadsheet Paradigm". In: *IEEE Trans. Software Eng.* 26.10 (2000), pp. 923–942.

[BDT06]   Magiel Bruntink, Arie van Deursen, and Tom Tourwé. "Discovering Faults in Idiom-based Exception Handling". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 242–251.

[BE11]   Matt Bishop and Chip Elliott. "Robust Programming by Example". In: *Information Assurance and Security Education and Training - 8th IFIP WG 11.8 World Conference on Information Security Education, WISE 8, Auckland, New Zealand, July 8-10, 2013, Proceedings, WISE 7, Lucerne Switzerland, June 9-10, 2011, and WISE 6, Bento Gonçalves, RS, Brazil, July 27-31, 2009, Revised Selected Papers*. 2011, pp. 140–147.

[Ben01]   Mordechai Ben-Ari. "The bug that destroyed a rocket". In: *ACM SIGCSE Bull.* 33.2 (2001), pp. 58–59.

[Ber+08]   Ivo Augusto Bertoncello, Marcelo Oliveira Dias, Patrick H. S. Brito, and Cecília M. F. Rubira. "Explicit Exception Handling Variability in Component-based Product Line Architectures". In: *Proceedings of the 4th International Workshop on Exception Handling*. WEH '08. Atlanta, Georgia: ACM, 2008, pp. 47–54.

[Ber+11]   R. D. Bernardo, R. Sales Jr., F. Castor, R. Coelho, N. Cacho, and S. Soares. "Agile Testing of Exceptional Behavior". In: *2011 25th Brazilian Symposium on Software Engineering*. Sept. 2011, pp. 204–213.

[Bez+19]   Vladymir L. Bezerra, Lincoln S. Rocha, João Bosco F. Filho, and Fernando A. M. Trinta. "An Empirical Study on Inter-Component Exception Notification in Android Platform". In: *XXXIII Brazilian Symposium on Software Engineering (SBES 2019), September 23–27, 2019, Salvador, Brazil*. 2019, pp. 73–83.

[BF04]   Matt Bishop and Deborah A. Frincke. "Teaching Robust Programming". In: *IEEE Security & Privacy* 2.2 (2004), pp. 54–57.

[BG18]   E. A. Barbosa and A. Garcia. "Global-Aware Recommendations for Repairing Violations in Exception Handling". In: *IEEE Transactions on Software Engineering* 44.9 (Sept. 2018), pp. 855–873.

[BGB14]   E. A. Barbosa, A. Garcia, and S. D. J. Barbosa. "Categorizing Faults in Exception Handling: A Study of Open Source Projects". In: *2014 Brazilian Symposium on Software Engineering*. Sept. 2014, pp. 11–20.

[Bin00]   Robert V. Binder. *Testing Object Oriented Systems: Models, Patterns and Tools*. Addison Wesley, 2000.

[Bin94]   Robert V. Binder. "Design for Testability in Object-Oriented Systems". In: *Commun. ACM* 37.9 (1994), pp. 87–101.

[Bla82]   Andrew P. Black. "Exception Handling: The Case Against". PhD thesis. Balliol College, 1982.

[BM00]   Peter A. Buhr and W.Y. Russell Mok. "Advanced Exception Handling Mechanisms". In: *IEEE Trans. Software Eng.* 26.9 (Sept. 2000).

[Bor+13]   M. N. Borazjany, L. S. Ghandehari, Y. Lei, R. Kacker, and R. Kuhn. "An Input Space Modeling Methodology for Combinatorial Testing". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. Mar. 2013, pp. 372–381.

[Bou85]   Luc Bougé. "A Contribution to the Theory of Program Testing". In: *Theor. Comput. Sci.* 37 (1985), pp. 151–181.

[Bra+05]   Christian Braun, Felix Wortmann, Martin Hafner, and Robert Winter. "Method construction - a core approach to organizational engineering". In: *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*. 2005, pp. 1295–1299.

[BV05]   K. Z. Bell and M. A. Vouk. "On effectiveness of pairwise methodology for testing network-centric software". In: *2005 International Conference on Information and Communication Technology*. Dec. 2005, pp. 221–235.

[BVJ12]   William Alton Ballance, Sergiy Vilkomir, and William Jenkins. "Effectiveness of Pair-Wise Testing for Software with Boolean Inputs". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. Ed. by Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche. IEEE Computer Society, 2012, pp. 580–586.

[Cac+08]   Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. "EJFlow: Taming Exceptional Control Flows in Aspect-oriented Programming". In: *Proceedings of the 7th International Conference on Aspect-oriented Software Development*. AOSD '08. Brussels, Belgium: ACM, 2008, pp. 72–83.

[Cac+09]   N. Cacho, F. Dantas, A. Garcia, and F. Castor. "Exception Flows Made Explicit: An Exploratory Study". In: *2009 XXIII Brazilian Symposium on Software Engineering*. Oct. 2009, pp. 43–53.

[Cac+14]   Nélio Cacho, Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel García, Eiji Adachi Barbosa, and Alessandro Garcia. "Trading robustness for maintainability: an empirical study of evolving c# programs". In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 2014, pp. 584–595.

[Cau+13]   Adnan Causevic, Rakesh Shukla, Sasikumar Punnekkat, and Daniel Sundmark. "Effects of Negative Testing on TDD: An Industrial Experiment". In: *Agile Processes in Software Engineering and Extreme Programming - 14th International Conference, XP 2013, Vienna, Austria, June 3-7, 2013. Proceedings*. 2013, pp. 91–105.

[CC16]   Byeong-Mo Chang and Kwanghoon Choi. "A review on exception analysis". In: *Information & Software Technology* 77 (2016), pp. 1–16.

[CDS07]   Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. "Interaction testing of highly-configurable systems in the presence of constraints". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. 2007, pp. 129–139.

[CDS08]   Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach". In: *IEEE Trans. Software Eng.* 34.5 (2008), pp. 633–650.

[Che+19]   Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. "Understanding Exception-Related Bugs in Large-Scale Cloud Systems". In: *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. 2019.

[Cho+16]   Eun-Hye Choi, Cyrille Artho, Takashi Kitamura, Osamu Mizuno, and Akihisa Yamada. "Distance-Integrated Combinatorial Testing". In: *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*. IEEE Computer Society, 2016, pp. 93–104.

[CM07]   Bruno Cabral and Paulo Marques. "Exception Handling: A Field Study in Java and .NET". In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. ECOOP'07. Berlin, Germany: Springer-Verlag, 2007, pp. 151–175.

[CM08]   Charles J. Colbourn and Daniel W. McClary. "Locating and detecting arrays for interaction faults". In: *J. Comb. Optim.* 15.1 (2008), pp. 17–48.

[CMH16]   Eun-Hye Choi, Osamu Mizuno, and Yifan Hu. "Code Coverage Analysis of Combinatorial Testing". In: *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2016) and 1st International Workshop on Technical Debt Analytics (TDA 2016) co-located with the 23rd Asia-Pacific Software Engineering Conference (APSEC 2016), Hamilton, New Zealand, December 6, 2016*. 2016, pp. 43–49.

[Coe+08a]   Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Cutigi Ferrari, Nélio Cacho, Uirá Kulesza, Arndt von Staa, and Carlos José Pereira de Lucena. "Assessing the Impact of Aspects on Exception Flows: An Exploratory Study". In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. 2008, pp. 207–234.

[Coe+08b]   Roberta Coelho, Awais Rashid, Arndt von Staa, James Noble, Uirá Kulesza, and Carlos Lucena. "A Catalogue of Bug Patterns for Exception Handling in Aspect-oriented Programs". In: *Proceedings of the 15th Conference on Pattern Languages of Programs*. PLoP '08. Nashville, Tennessee, USA: ACM, 2008, 23:1–23:13.

[Coh+03]   Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, Charles J. Colbourn, and James S. Collofello. "Variable Strength Interaction Testing of Components". In: *27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems, 3-6 November 2003, Dallas, TX, USA, Proceedings*. 2003, p. 413.

[Coh+94]   David M. Cohen, Siddhartha R. Dalal, A. Kajla, and Gardner C. Patton. "The Automatic Efficient Test Generator (AETG) system". In: *5th International Symposium on Software Reliability Engineering, ISSRE 1994, Monterey, CA, USA, November 6-9, 1994*. 1994, pp. 303–309.

[Coh+97]   David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. "The AETG System: An Approach to Testing Based on Combinatiorial Design". In: *IEEE Trans. Software Eng.* 23.7 (1997), pp. 437–444.

[Coo+17]   Steve Cook, Conrad Bock, Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, and Doug Tolbert. *Unified Modeling Language (UML) Version 2.5.1*. Standard. Object Management Group (OMG), Dec. 2017.

[Cot+16]   Domenico Cotroneo, Roberto Pietrantuono, Stefano Russo, and Kishor S. Trivedi. "How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation". In: *Journal of Systems and Software* 113 (2016), pp. 27–43.

[Cri82]   Flaviu Cristian. "Exception Handling and Software Fault Tolerance". In: *IEEE Trans. Computers* 31.6 (1982), pp. 531–540.

[Cri84]   Flaviu Cristian. "Correct and Robust Programs". In: *IEEE Trans. Software Eng.* 10.2 (1984), pp. 163–174.

[Cri89]   Flaviu Cristian. "Exception Handling". In: *Dependability of Resilient Computers*. Ed. by T. Anderson. BSP Professional Books, 1989, pp. 68–97.

[CYZ10]   Baiqiang Chen, Jun Yan, and Jian Zhang. "Combinatorial Testing with Shielding Parameters". In: *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. Ed. by Jun Han and Tran Dan Thu. IEEE Computer Society, 2010, pp. 280–289.

[CZ11]   Baiqiang Chen and Jian Zhang. "Tuple density: a new metric for combinatorial test suites". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 2011, pp. 876–879.

[Cza99]   Krzysztof Czarnecki. "Generative programming - principles and techniques of software engineering based on automated configuration and fragment-based component models". PhD thesis. Technische Universität Illmenau, Germany, 1999.

[Cze06]   Jacek Czerwonka. "Pairwise testing in real world". In: *24th Pacific Northwest Software Quality Conference*. Vol. 200. Citeseer. 2006.

[DG94]   Steven J. Drew and K. John Gough. "Exception Handling: Expecting the Unexpected". In: *Comput. Lang.* 20.2 (1994), pp. 69–87.

[DM98]   Siddhartha R. Dalal and Colin L. Mallows. "Factor-Covering Designs for Testing Software". In: *Technometrics* 40.3 (1998), pp. 234–243.

[Don+06]   Christophe Dony, Jørgen Lindskov Knudsen, Alexander B. Romanovsky, and Anand Tripathi, eds. *Advanced Topics in Exception Handling Techniques (the book grow out of ECOOP workshops)*. Vol. 4119. Lecture Notes in Computer Science. Springer, 2006.

[ECS15]   Felipe Ebert, Fernando Castor, and Alexander Serebrenik. "An exploratory study on exception handling bugs in Java programs". In: *Journal of Systems and Software* 106 (2015), pp. 82–101.

[Fel+14]   Alexander Felfernig, Stefan Reiterer, Florian-Christoph Reinfrank, Gerald Ninaus, and Michael Jeran. "Conflict Detection and Diagnosis in Configuration". English. In: *Knowledge-based Configuration: From Research to Business Cases*. 1st ed. Netherlands: Elsevier B.V., 2014, pp. 73–87.

[FG10]   Gordon Fraser and Angelo Gargantini. "Generating Minimal Fault Detecting Test Suites for Boolean Expressions". In: *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*. IEEE Computer Society, 2010, pp. 37–45.

[Fil+06]   Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília M. F. Rubira. "Exceptions and aspects: the devil is in the details". In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*. 2006, pp. 152–162.

[Fil+09]   Fernando Castor Filho, Nélio Cacho, Eduardo Figueiredo, Alessandro Garcia, Cecília M. F. Rubira, Jefferson Silva de Amorim, and Hítalo Oliveira da Silva. "On the modularization and reuse of exception handling with aspects". In: *Softw., Pract. Exper.* 39.17 (2009), pp. 1377–1417.

[For+08]   Michael Forbes, Jim Lawrence, Yu Lei, Raghu N Kacker, and Richard D Kuhn. "Refining the In-Parameter-Order Strategy for Constructing Covering Arrays". In: *Journal of Research of the National Institute of Standards and Technology* 113.5 (Sept. 2008), pp. 287–297.

[FR07]   C. Fu and B. G. Ryder. "Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications". In: *29th International Conference on Software Engineering (ICSE'07)*. May 2007, pp. 230–239.

[Fre91]   Roy S. Freedman. "Testability of Software Components". In: *IEEE Trans. Software Eng.* 17.6 (1991), pp. 553–564.

[FW88]   Phyllis G. Frankl and Elaine J. Weyuker. "An Applicable Family of Data Flow Testing Criteria". In: *IEEE Trans. Software Eng.* 14.10 (1988), pp. 1483–1498.

[FW92]   Eugene C. Freuder and Richard J. Wallace. "Partial Constraint Satisfaction". In: *Artif. Intell.* 58.1-3 (1992), pp. 21–70.

[FW93a]   Phyllis G. Frankl and Stewart N. Weiss. "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing". In: *IEEE Trans. Software Eng.* 19.8 (1993), pp. 774–787.

[FW93b]   Phyllis G. Frankl and Elaine J. Weyuker. "A Formal Analysis of the Fault-Detecting Ability of Testing Methods". In: *IEEE Trans. Software Eng.* 19.3 (1993), pp. 202–213.

[Gao+14]   Shiwei Gao, Jianghua Lv, Binglei Du, Yaruo Jiang, and Shilong Ma. "General Optimization Strategies for Refining the In-Parameter-Order Algorithm". In: *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014*. 2014, pp. 21–26.

[Gao+15]   Shi-Wei Gao, Jiang-Hua Lv, Bing-Lei Du, Charles J. Colbourn, and Shi-Long Ma. "Balancing Frequencies and Fault Detection in the In-Parameter-Order Algorithm". In: *J. Comput. Sci. Technol.* 30.5 (2015), pp. 957–968.

[Gar+01]   Alessandro F. Garcia, Cecília M. F. Rubira, Alexander B. Romanovsky, and Jie Xu. "A comparative study of exception handling mechanisms for building dependable object-oriented software". In: *J. Syst. Softw.* 59.2 (2001), pp. 197–222.

[Gar+16]   Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. "Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing". In: *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*. 2016, pp. 49–63.

[Gar11]   Angelo Gargantini. "Dealing with Constraints in Boolean Expression Testing". In: *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*. IEEE Computer Society, 2011, pp. 322–327.

[Geh92]   Narain H. Gehani. "Exceptional C or C with Exceptions". In: *Softw., Pract. Exper.* 22.10 (1992), pp. 827–848.

[GG75]   John B. Goodenough and Susan L. Gerhart. "Toward a Theory of Test Data Selection". In: *IEEE Trans. Software Eng.* 1.2 (1975), pp. 156–173.

[GH05]   Alexander Gruler and Christian Heinlein. "Exception Handling with Resumption: Design and Implementation in Java". In: *Proceedings of The 2005 International Conference on Programming Languages and Compilers, PLC 2005, Las Vegas, Nevada, USA, June 27-30, 2005.* 2005, pp. 165–171.

[Gha+12]   Laleh Shikh Gholamhossein Ghandehari, Yu Lei, Tao Xie, D. Richard Kuhn, and Raghu Kacker. "Identifying Failure-Inducing Combinations in a Combinatorial Test Set". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012.* 2012, pp. 370–379.

[GO07]   Mats Grindal and Jeff Offutt. "Input Parameter Modeling for Combination Strategies". In: *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering.* SE'07. Innsbruck, Austria: ACTA Press, 2007, pp. 255–260.

[GOA05]   Mats Grindal, Jeff Offutt, and Sten F. Andler. "Combination testing strategies: a survey". In: *Softw. Test., Verif. Reliab.* 15.3 (2005), pp. 167–199.

[GOM07]   Mats Grindal, Jeff Offutt, and Jonas Mellin. "Managing Conflicts When Using Combination Strategies to Test Software". In: *18th Australian Software Engineering Conference (ASWEC 2007), April 10-13, 2007, Melbourne, Australia.* 2007, pp. 255–264.

[Goo75a]   John B. Goodenough. "Exception Handling Design Issues". In: *SIGPLAN Not.* 10.7 (1975).

[Goo75b]   John B. Goodenough. "Exception Handling: Issues and a Proposed Notation". In: *Commun. ACM* 18.12 (1975), pp. 683–696.

[Goo75c]   John B. Goodenough. "Structured Exception Handling". In: *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, USA, January 1975.* 1975, pp. 204–224.

[Gou83]   John S. Gourlay. "A Mathematical Framework for the Investigation of Testing". In: *IEEE Trans. Software Eng.* 9.6 (1983), pp. 686–709.

[GPR17]   A. Gargantini, J. Petke, and M. Radavelli. "Combinatorial Interaction Testing for Automated Constraint Repair". In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* Mar. 2017, pp. 239–248.

[Gri+03]   Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F Andler. *An Evaluation of Combination Strategies for Test Case Selection (Technical Report).* Tech. rep. HS-IDA-TR-03-001. University of Skövde, School of Humanities and Informatics, 2003.

[Gri+06]   Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F. Andler. "An evaluation of combination strategies for test case selection". In: *Empirical Software Engineering* 11.4 (2006), pp. 583–611.

[Hal+19]   Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. "Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack". In: *Empirical Software Engineering* 24.2 (2019), pp. 674–717.

[HCG15]   Sylvain Hallé, Edmond La Chance, and Sébastien Gaboury. "Graph Methods for Generating Test Cases with Universal and Existential Constraints". In: *Testing Software and Systems - 27th IFIP WG 6.1 International Conference, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, November 23-25, 2015, Proceedings*. Ed. by Khaled El-Fakih, Gerassimos D. Barlas, and Nina Yevtushenko. Vol. 9447. Lecture Notes in Computer Science. Springer, 2015, pp. 55–70.

[Hie02]   Robert M. Hierons. "Comparing test sets and criteria in the presence of test hypotheses and fault domains". In: *ACM Trans. Softw. Eng. Methodol.* 11.4 (2002), pp. 427–448.

[How76]   William E. Howden. "Reliability of the Path Analysis Testing Strategy". In: *IEEE Trans. Software Eng.* 2.3 (1976), pp. 208–215.

[HP85]   D. Harel and A. Pnueli. "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems*. Ed. by Krzysztof R. Apt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 477–498.

[IEE90a]   IEEE. "IEEE Standard Glossary of Data Management Terminology". In: *IEEE Std 610.5-1990* (1990).

[IEE90b]   IEEE. "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std* 610.12-1990 (Dec. 1990).

[Jak+15]   B. Jakobus, E. A. Barbosa, A. Garcia, and C. J. P. de Lucena. "Contrasting exception handling code across languages: An experience report involving 50 open source projects". In: *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2015, pp. 183–193.

[JF88]   Ralph E Johnson and Brian Foote. "Designing reusable classes". In: *J. Object Oriented Program.* 1.2 (1988), pp. 22–35.

[JG93]   Guy L. Steele Jr. and Richard P. Gabriel. "The Evolution of Lisp". In: *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*. 1993, pp. 231–270.

[Jin+17]   Jinyu Liu, Shanshan Li, Zhouyang Jia, Xiaodong Liu, Bin Lin, and Xiangke Liao. "IdenEH: Identify error-handling code snippets in large-scale software". In: *2017 17th International Conference on Computational Science and Its Applications (ICCSA)*. July 2017, pp. 1–8.

[JK18]   Rekha Jayaram and R. Krishnan. "Approaches to Fault Localization in Combinatorial Testing: A Survey". In: *Smart Computing and Informatics*. Ed. by Suresh Chandra Satapathy, Vikrant Bhateja, and Swagatam Das. Singapore: Springer Singapore, 2018, pp. 533–540.

[Jo+04]   Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. "An uncaught exception analysis for Java". In: *Journal of Systems and Software* 72.1 (2004), pp. 59–69.

[Jor14]   Paul C. Jorgensen. *Software testing - a craftsman's approach (3. ed.)* Taylor & Francis, 2014.

[JSS15]   Dietmar Jannach, Thomas Schmitz, and Kostyantyn M. Shchekotykhin. "Parallelized Hitting Set Computation for Model-Based Diagnosis". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. 2015, pp. 1503–1510.

[JST20]   Hao Jin, Ce Shi, and Tatsuhiro Tsuchiya. "Constrained detecting arrays for fault localization in combinatorial testing". In: *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*. Ed. by Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini. ACM, 2020, pp. 1971–1978.

[JT20]   Hao Jin and Tatsuhiro Tsuchiya. "Constrained locating arrays for combinatorial interaction testing". In: *J. Syst. Softw.* 170 (2020), p. 110771.

[Jun04]   Ulrich Junker. "QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems". In: *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*. 2004, pp. 167–172.

[KD00]   Phil Koopman and John DeVale. "The Exception Handling Effectiveness of POSIX Operating Systems". In: *IEEE Trans. Software Eng.* 26.9 (2000), pp. 837–848.

[KJS98]   Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. "Automated Robustness Testing of Off-the-Shelf Software Components". In: *Digest of Papers: FTCS-28, The Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23-25, 1998*. 1998, pp. 230–239.

[KKL13]   D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. *Introduction to Combinatorial Testing*. 1st. Chapman & Hall/CRC, 2013.

[KKL16]   D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. "Estimating t-Way Fault Profile Evolution During Testing". In: *40th IEEE Annual Computer Software and Applications Conference, COMPSAC Workshops 2016, Atlanta, GA, USA, June 10-14, 2016*. 2016, pp. 596–597.

[KL14]   Sunint Kaur Khalsa and Yvan Labiche. "An Orchestrated Survey of Available Algorithms and Tools for Combinatorial Testing". In: *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*. 2014, pp. 323–334.

[Knu87]   Jørgen Lindskov Knudsen. "Better Exception-Handling in Block-Structured Systems". In: *IEEE Software* 4.3 (1987), pp. 40–49.

[KO06]   D. Richard Kuhn and Vadim Okun. "Pseudo-Exhaustive Testing for Software". In: *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*. 2006, pp. 153–158.

[Koo+97]   Philip Koopman, John Sung, Christopher P. Dingman, Daniel P. Siewiorek, and Ted Marz. "Comparing Operating Systems Using Robustness Benchmarks". In: *The Sixteenth Symposium on Reliable Distributed Systems, SRDS 1997, Durham, North Carolina, USA, October 22-24, 1997, Proceedings*. 1997, pp. 72–79.

[KPP95]   Barbara A. Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. "Case Studies for Method and Tool Evaluation". In: *IEEE Softw.* 12.4 (1995), pp. 52–62.

[KR02]   D. R. Kuhn and M. J. Reilly. "An investigation of the applicability of design of experiments to software testing". In: *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.* Dec. 2002, pp. 91–95.

[KS18]   Kristoffer Kleine and Dimitris E. Simos. "An Efficient Design and Implementation of the In-Parameter-Order Algorithm". In: *Mathematics in Computer Science* 12.1 (2018), pp. 51–67.

[KSS17]   Maria Kechagia, Tushar Sharma, and Diomidis Spinellis. "Towards a context dependent Java exceptions hierarchy". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 2017, pp. 347–349.

[Kuh+15]   D. Richard Kuhn, Renée C. Bryce, Feng Duan, Laleh Shikh Gholamhossein Ghandehari, Yu Lei, and Raghu N. Kacker. "Combinatorial Testing: Theory and Practice". In: *Advances in Computers* 99 (2015), pp. 1–66.

[Kuh+20]   Rick Kuhn, Raghu N. Kacker, Yu Lei, and Dimitris E. Simos. "Input Space Coverage Matters". In: *IEEE Computer* 53.1 (2020), pp. 37–44.

[Kuh99]   D. Richard Kuhn. "Fault classes and error detection capability of specification-based testing". In: *ACM Trans. Softw. Eng. Methodol.* 8.4 (1999), pp. 411–424.

[KWG04]   D. R. Kuhn, D. R. Wallace, and A. M. Gallo. "Software fault interactions and implications for software testing". In: *IEEE Transactions on Software Engineering* 30.6 (June 2004), pp. 418–421.

[Lan97]   Gérard Le Lann. "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective". In: *1997 Workshop on Engineering of Computer-Based Systems (ECBS '97), March 24-28, 1997, Monterey, CA, USA*. 1997, pp. 339–246.

[LCM19]   Erin Lanus, Charles J. Colbourn, and Douglas C. Montgomery. "Partitioned Search with Column Resampling for Locating Array Construction". In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*. IEEE, 2019, pp. 214–223.

[Lee83]   P. A. Lee. "Exception Handling in C Programs". In: *Softw., Pract. Exper.* 13.5 (1983), pp. 389–405.

[Lei+07]   Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. "IPOG: A General Strategy for T-Way Software Testing". In: *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007), 26-29 March 2007, Tucson, Arizona, USA*. 2007, pp. 549–556.

[Lei+08]   Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing". In: *Softw. Test. Verification Reliab.* 18.3 (2008), pp. 125–148.

[Lev+93]   Laura M. Leventhal, Barbee Teasley, Diane S. Rohlman, and Keith Instone. "Positive Test Bias in Software Testing Among Professionals: A Review". In: *Human-Computer Interaction, Third International Conference, EWHCI '93, Moscow, Russia, August 3-7, 1993, Selected Papers*. 1993, pp. 210–218.

[Li+18]   Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, and J. Xuan. "EH-Recommender: Recommending Exception Handling Strategies Based on Program Context". In: *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*. Dec. 2018, pp. 104–114.

[Lig09]   Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2nd. Spektrum Akademischer Verlag, 2009.

[LL00]   M. Lippert and C. V. Lopes. "A study on exception detection and handling using aspect-oriented programming". In: *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. June 2000, pp. 418–427.

[LL13]   Jochen Ludewig and Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. 3rd. dpunkt.verlag, 2013.

[LNL12]   J. Li, C. Nie, and Y. Lei. "Improved Delta Debugging Based on Combinatorial Testing". In: *2012 12th International Conference on Quality Software*. Aug. 2012, pp. 102–105.

[LO17]   N. Li and J. Offutt. "Test Oracle Strategies for Model-Based Testing". In: *IEEE Transactions on Software Engineering* 43.4 (Apr. 2017), pp. 372–395.

[LS98]   Jun Lang and David B. Stewart. "A Study of the Applicability of Existing Exception-handling Techniques to Component-based Real-time Software Technology". In: *ACM Trans. Program. Lang. Syst.* 20.2 (Mar. 1998), pp. 274–301.

[LT98]   Yu Lei and Kuo-Chung Tai. "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing". In: *3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98), 13-14 November 1998, Washington, D.C, USA, Proceedings*. 1998, pp. 254–261.

[Mar11]   Cristina Marinescu. "Are the classes that use exceptions defect prone?" In: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, EVOL/IWPSE 2011, Szeged, Hungary, September 5-6, 2011.* 2011, pp. 56–60.

[MCK12]   H. Melo, R. Coelho, and U. Kulesza. "On a Feature-Oriented Characterization of Exception Flows in Software Product Lines". In: *2012 26th Brazilian Symposium on Software Engineering*. Sept. 2012, pp. 121–130.

[Mel+13]   Hugo Melo, Roberta Coelho, Uirá Kulesza, and Demóstenes Sena. "In-depth characterization of exception flows in software product lines: an empirical study". In: *J. Software Eng. R&D* 1 (2013), p. 3.

[Mey92]   Bertrand Meyer. "Applying "Design by Contract"". In: *IEEE Computer* 25.10 (1992), pp. 40–51.

[Mic+12]   Zoltán Micskei, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, and Nuno Antunes. "Robustness Testing Techniques and Tools". In: *Resilience Assessment and Evaluation of Computing Systems*. Springer, 2012, pp. 323–339.

[Mit13]   Jürgen Mittelstraß, ed. *Enzyklopädie Philosophie und Wissenschaftstheorie*. Vol. 2. J.B. Metzler, 2013.

[Mon+18]   Taiza Montenegro, Hugo Melo, Roberta Coelho, and Eiji Barbosa. "Improving developers awareness of the exception handling policy". In: *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. 2018, pp. 413–422.

[Mor90]   Larry J. Morell. "A Theory of Fault-Based Testing". In: *IEEE Trans. Software Eng.* 16.8 (1990), pp. 844–857.

[MPZ15]   Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. "Recent Advances in Automatic Black-Box Testing". In: *Advances in Computers* 99 (2015), pp. 157–193.

[MR77]   P. M. Melliar-Smith and Brian Randell. "Software Reliability: The Role of Programmed Exception Handling". In: *Proceedings of an ACM Conference on Language Design for Reliable Software (LDRS), Raleigh, North Carolina, USA, March 28-30, 1977*. 1977, pp. 95–100.

[MSB11]   Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011.

[MT97]   Robert Miller and Anand R. Tripathi. "Issues with Exception Handling in Object-Oriented Systems". In: *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*. 1997, pp. 85–103.

[MTH90]   Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.

[Niu+13]   X. Niu, C. Nie, Y. Lei, and A. T. S. Chan. "Identifying Failure-Inducing Combinations Using Tuple Relationship". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. Mar. 2013, pp. 271–280.

[Niu+18]   X. Niu, n. changhai, H. K. N. Leung, Y. Lei, X. Wang, J. Xu, and Y. Wang. "An in-
            terleaving approach to combinatorial testing and failure-inducing interaction iden-
            tification". In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1.

[NL11a]   Changhai Nie and Hareton Leung. "A survey of combinatorial testing". In: *ACM
            Comput. Surv.* 43.2 (2011), 11:1–11:29.

[NL11b]   Changhai Nie and Hareton Leung. "The Minimal Failure-Causing Schema of Com-
            binatorial Testing". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (Sept. 2011), 15:1–
            15:38.

[Off+01]   Jeff Offutt, Roger T. Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson.
            "A Fault Model for Subtype Inheritance and Polymorphism". In: *12th International
            Symposium on Software Reliability Engineering (ISSRE 2001), 27-30 November
            2001, Hong Kong, China*. 2001, pp. 84–95.

[Oli+16]   Juliana Oliveira, Nélio Cacho, Deise Borges, Thaisa Silva, and Fernando Castor.
            "An Exploratory Study of Exception Handling Behavior in Evolving Android and
            Java Applications". In: *Proceedings of the 30th Brazilian Symposium on Software
            Engineering, SBES 2016, Maringá, Brazil, September 19 - 23, 2016*. 2016, pp. 23–
            32.

[Oli+18]   Juliana Oliveira, Deise Borges, Thaisa Silva, Nélio Cacho, and Fernando Castor.
            "Do android developers neglect error handling? a maintenance-Centric study on the
            relationship between android abstractions and uncaught exceptions". In: *Journal of
            Systems and Software* 136 (2018), pp. 1–18.

[Pan99]   Jiantao Pan. "The Dimensionality of Failures - A Fault Model for Characterizing
            Software Robustness". In: *Proceedings of FTCS'99, 15-18 June 1999, Madison,
            Wisconsin*. 1999.

[Par76]   David Lorge Parnas. "On the Design and Development of Program Families". In:
            IEEE Trans. Software Eng. 2.1 (1976), pp. 1–9.

[PBL05]   Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line En-
            gineering - Foundations, Principles, and Techniques*. Springer, 2005.

[Pet+13]   Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. "Efficiency and early
            fault detection with lower and higher strength combinatorial interaction testing".
            In: *Joint Meeting of the European Software Engineering Conference and the ACM
            SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13,
            Saint Petersburg, Russian Federation, August 18-26, 2013*. 2013, pp. 26–36.

[Pet+15]   Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. "Practical Combina-
            torial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detec-
            tion". In: *IEEE Trans. Software Eng.* 41.9 (2015), pp. 901–924.

[Pet15]   Justyna Petke. "Constraints: The Future of Combinatorial Interaction Testing". In:
            *8th IEEE/ACM International Workshop on Search-Based Software Testing, SBST
            2015, Florence, Italy, May 18-19, 2015*. 2015, pp. 17–18.

[PH18]   Krishna Patel and Robert M. Hierons. "A mapping study on testing non-testable systems". In: *Software Quality Journal* 26.4 (2018), pp. 1373–1413.

[PKS99]   J. Pan, P. Koopman, and D. Siewiorek. "A dimensionality model approach to testing and improving software robustness". In: *1999 IEEE AUTOTESTCON Proceedings (Cat. No.99CH36323)*. Aug. 1999, pp. 493–501.

[Pre+13]   Alexander Pretschner, Dominik Holling, Robert Eschbach, and Matthias Gemmar. "A Generic Fault Model for Quality Assurance". In: *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*. 2013, pp. 87–103.

[PRT00a]   Dewayne E. Perry, Alexander B. Romanovsky, and Anand Tripathi. "Guest Editors' Introduction - Current Trends in Exception Handling". In: *IEEE Trans. Software Eng.* 26.10 (2000), pp. 921–922.

[PRT00b]   Dewayne E. Perry, Alexander Romanovsky, and Anand R. Tripathi. "Guest Editors' Introduction - Current Trends in Exception Handling". In: *IEEE Trans. Software Eng.* 26.9 (2000), pp. 817–819.

[PS17a]   G. B. D. Pádua and W. Shang. "Studying the Prevalence of Exception Handling Anti-Patterns". In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. May 2017, pp. 328–331.

[PS17b]   G. B. d. Pádua and W. Shang. "Revisiting Exception Handling Practices with Exception Flow Analysis". In: *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Sept. 2017, pp. 11–20.

[PW17]   Ingo Pill and Franz Wotawa. "Model-Based Diagnosis Meets Combinatorial Testing For Generating an Abductive Diagnosis Model". In: *28th International Workshop on Principles of Diagnosis (DX'17), Brescia, Italy, September 26-29, 2017*. Ed. by Marina Zanella, Ingo Pill, and Alessandro Cimatti. Vol. 4. Kalpa Publications in Computing. EasyChair, 2017, pp. 248–263.

[PW18]   Ingo Pill and Franz Wotawa. "On Using an I/O Model for Creating an Abductive Diagnosis Model via Combinatorial Exploration, Fault Injection, and Simulation". In: *Proceedings of the 29th International Workshop on Principles of Diagnosis co-located with 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS 2018), Warsaw, Poland, 27-30 August, 2018*. Ed. by Louise Travé-Massuyès and Anna Sztyber. Vol. 2289. CEUR Workshop Proceedings. CEUR-WS.org, 2018.

[PW19]   Ingo Pill and Franz Wotawa. "Exploiting Observations from Combinatorial Testing for Diagnostic Reasoning". In: *Proceedings of the 30th International Workshop on Principles of Diagnosis (DX'19), Klagenfurt, Austria, 11-13 November, 2019*. 2019.

[PY08]   Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. USA: John Wiley & Sons, Inc., 2008.

[Ran75]   Brian Randell. "System Structure for Software Fault Tolerance". In: *IEEE Trans. Software Eng.* 1.2 (1975), pp. 221–232.

[Rat+16]   Zachary B. Ratliff, D. Richard Kuhn, Raghu N. Kacker, Yu Lei, and Kishor S. Trivedi. "The Relationship between Software Bug Type and Number of Factors Involved in Failures". In: *2016 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2016, Ottawa, ON, Canada, October 23-27, 2016*. 2016, pp. 119–124.

[Rei87]   Raymond Reiter. "A Theory of Diagnosis from First Principles". In: *Artif. Intell.* 32.1 (1987), pp. 57–95.

[RH09]   Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164.

[RM00]   Martin P. Robillard and Gail C. Murphy. "Designing Robust Java Programs with Exceptions". In: *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*. SIGSOFT '00/FSE-8. San Diego, California, USA: ACM, 2000, pp. 2–10.

[RM03]   Martin P. Robillard and Gail C. Murphy. "Static Analysis to Support the Evolution of Exception Structure in Object-oriented Systems". In: *ACM Trans. Softw. Eng. Methodol.* 12.2 (Apr. 2003), pp. 191–221.

[RM99]   Martin P. Robillard and Gail C. Murphy. "Analyzing Exception Flow in Java Programs". In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-7. Toulouse, France: Springer-Verlag, 1999, pp. 322–337.

[RN10]   Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall PTR, 2010.

[Rom+01]   Alexander B. Romanovsky, Christophe Dony, Jørgen Lindskov Knudsen, and Anand Tripathi, eds. *Advances in Exception Handling Techniques (the book grow out of a ECOOP 2000 workshop)*. Vol. 2022. Springer, 2001.

[SA16]   P. Sawadpong and E. B. Allen. "Software Defect Prediction Using Exception Handling Call Graphs: A Case Study". In: *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*. Jan. 2016, pp. 55–62.

[SAW12]   P. Sawadpong, E. B. Allen, and B. J. Williams. "Exception Handling Defects: An Empirical Study". In: *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*. Oct. 2012, pp. 90–97.

[SCK16]   Demóstenes Sena, Roberta Coelho, and Uirá Kulesza. "Integrated Analysis of Exception Flows and Handler Actions in Java Libraries: An Empirical Study". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: ACM, 2016, pp. 1520–1526.

[SF13]   Ali Shahrokni and Robert Feldt. "A systematic review of software robustness". In: *Information & Software Technology* 55.1 (2013), pp. 1–17.

[SGH10]   H. Shah, C. Gorg, and M. J. Harrold. "Understanding Exception Handling: Viewpoints of Novices and Experts". In: *IEEE Transactions on Software Engineering* 36.2 (Mar. 2010), pp. 150–161.

[SH00]   Saurabh Sinha and Mary Jean Harrold. "Analysis and Testing of Programs with Exception Handling Constructs". In: *IEEE Trans. Software Eng.* 26.9 (2000), pp. 849–871.

[She14]   George B. Sherwood. "Functional Dependence and Equivalence Class Factors in Combinatorial Test Designs". In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*. 2014, pp. 108–117.

[She15]   George B. Sherwood. "Embedded functions in combinatorial test designs". In: *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. 2015, pp. 1–10.

[She16]   George B. Sherwood. "Embedded Functions for Constraints and Variable Strength in Combinatorial Testing". In: *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. 2016, pp. 65–74.

[She17]   George B. Sherwood. "Embedded Functions for Test Design Automation". In: *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*. 2017, pp. 221–224.

[She94]   George B. Sherwood. "Effective Testing of Factor Combinations". In: *Proceedings of the Third International Conference on Software Testing, Analysis and Review, Washington, DC*. 1994, pp. 151–166.

[Sho04]   Jim Shore. "Fail Fast". In: *IEEE Software* 21.5 (2004), pp. 21–25.

[Sil13]   Silva. "New Exception Interfaces for Java-like Languages". In: 2013.

[SLM12]   Suman Saha, Julia Lawall, and Gilles Muller. "Finding Resource-release Omission Faults in Linux". In: *SIGOPS Oper. Syst. Rev.* 45.3 (Jan. 2012), pp. 5–9.

[SNX05]   Liang Shi, Changhai Nie, and Baowen Xu. "A Software Debugging Method Based on Pairwise Testing". In: *Computational Science - ICCS 2005, 5th International Conference, Atlanta, GA, USA, May 22-25, 2005, Proceedings, Part III*. 2005, pp. 1088–1091.

[Som16]   Ian Sommerville. *Software Engineering, Global Edition*. 10th. Pearson, 2016.

[STF11]   Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. "Using binary decision diagrams for combinatorial test design". In: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. 2011, pp. 254–264.

[STZ12]   Itai Segall, Rachel Tzoref-Brill, and Aviad Zlotnick. "Common Patterns in Combinatorial Models". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. Ed. by Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche. IEEE Computer Society, 2012, pp. 624–629.

[TBL17]   Joel Kamdem Teto, Ruth Bearden, and Dan Chia-Tien Lo. "The Impact of Defensive Programming on I/O Cybersecurity Attacks". In: *Proceedings of the 2017 ACM Southeast Regional Conference, Kennesaw, GA, USA, April 13-15, 2017*. 2017, pp. 102–111.

[Tea+94] Barbee E Teasley, Laura Marie Leventhal, Clifford R Mynatt, and Diane S Rohlman. "Why software testing is sometimes ineffective: Two applied studies of positive test strategy." In: *Journal of Applied Psychology* 79.1 (1994), p. 142.

[Teu99]   Rolf Alexander Teubner. *Organisations- und Informationssystem- gestaltung: Theoretische Grundlagen und integrierte Methoden*. Deutscher Universitätsverlag, Wiesbaden, 1999.

[TQ17]   Wei-Tek Tsai and Guanqiu Qi. *Combinatorial Testing in Cloud Computing*. Springer Briefs in Computer Science. Springer, 2017.

[Tzo19]   Rachel Tzoref-Brill. "Chapter Two - Advances in Combinatorial Testing". In: *Advances in Computers* 112 (2019), pp. 79–134.

[UL07]   Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007.

[UQ17]   Hiroshi Ukai and Xiao Qu. "Test Design as Code: JCUnit". In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. 2017, pp. 508–515.

[VA15]   Sergiy Vilkomir and David Anderson. "Relationship between pair-wise and MC/DC testing: Initial experimental results". In: *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–4.

[Vil16]   Sergiy Vilkomir. "Combinatorial Testing of Software with Binary Inputs: A State-of-the-Art Review". In: *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Companion, Vienna, Austria, August 1-3, 2016*. IEEE, 2016, pp. 55–60.

[VSB13]   Sergiy Vilkomir, Oleksii Starov, and Ranjan Bhambroo. "Evaluation of t-wise Approach for Testing Logical Expressions in Software". In: *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*. 2013, pp. 249–256.

[VTP94]   Mladen A. Vouk, Kuo-Chung Tai, and Amit M. Paradkar. "Empirical studies of predicate-based software testing". In: *5th International Symposium on Software Reliability Engineering, ISSRE 1994, Monterey, CA, USA, November 6-9, 1994*. 1994, pp. 55–64.

[Wei89]   S. N. Weiss. "Comparing Test Data Adequacy Criteria". In: *SIGSOFT Softw. Eng. Notes* 14.6 (Oct. 1989), pp. 42–49.

[Wei90]   Stewart N. Weiss. "Methods of comparing test data adequacy criteria". In: *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference, COMPSAC 1990, Chicago, IL, USA, October 31 1990 - November 2, 1990*. 1990, pp. 1–6.

[Wey02]   Elaine J Weyuker. "Thinking formally about testing without a formal specification". In: *Proceedings of Formal Approaches to Testing of Software (FATES'02)*. 2002, pp. 1–10.

[Wey82]   Elaine J. Weyuker. "On Testing Non-Testable Programs". In: *Comput. J.* 25.4 (1982), pp. 465–470.

[Wey86]   Elaine J. Weyuker. "Axiomatizing Software Test Data Adequacy". In: *IEEE Trans. Software Eng.* 12.12 (1986), pp. 1128–1138.

[WGS94]   Elaine J. Weyuker, Tarak Goradia, and Ashutosh Singh. "Automatically Generating Test Data from a Boolean Specification". In: *IEEE Trans. Software Eng.* 20.5 (1994), pp. 353–363.

[Wie03]   Roelf J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate and the UML*. Morgan Kaufmann Publishers, 2003.

[Wie14]   Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.

[WJ91]   Elaine J. Weyuker and Bingchiang Jeng. "Analyzing Partition Testing Strategies". In: *IEEE Trans. Software Eng.* 17.7 (1991), pp. 703–711.

[WK01]   Dolores R. Wallace and D. Richard Kuhn. "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data". In: *International Journal of Reliability, Quality and Safety Engineering* 08.04 (2001), pp. 351–371.

[WN08]   Westley Weimer and George C. Necula. "Exceptional Situations and Program Reliability". In: *ACM Trans. Program. Lang. Syst.* 30.2 (Mar. 2008), 8:1–8:51.

[WO80]   Elaine J. Weyuker and Thomas J. Ostrand. "Theories of Program Testing and the the Application of Revealing Subdomains". In: *IEEE Trans. Software Eng.* 6.3 (1980), pp. 236–246.

[WP01]   A. W. Williams and R. L. Probert. "A measure for component interaction test coverage". In: *Proceedings ACS/IEEE International Conference on Computer Systems and Applications*. June 2001, pp. 304–311.

[WQ15]   Ziyuan Wang and Yuanchao Qi. "Why combinatorial testing works: Analyzing minimal failure-causing schemas in logic expressions". In: *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. 2015, pp. 1–5.

[WT14]   Paul Wojciak and Rachel Tzoref-Brill. "System Level Combinatorial Testing in Practice - The Concurrent Maintenance Case Study". In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. 2014, pp. 103–112.

[Wu+19a]   H. Wu, N. Changhai, J. Petke, Y. Jia, and M. Harman. "Comparative Analysis of Constraint Handling Techniques for Constrained Combinatorial Testing". In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1.

[Wu+19b]   Huayao Wu, Changhai Nie, Justyna Petke, Yue Jia, and Mark Harman. "A Survey of Constrained Combinatorial Testing". In: *CoRR* abs/1908.02480 (2019).

[WWH91]   Elaine J. Weyuker, Stewart N. Weiss, and Richard G. Hamlet. "Comparison of Program Testing Strategies". In: *Proceedings of the Symposium on Testing, Analysis, and Verification, TAV 1991, Victoria, British Columbia, Canada, October 8-10, 1991*. 1991, pp. 1–10.

[Xu+16]   Chiya Xu, Yuanchao Qi, Ziyuan Wang, and Weifeng Zhang. "Analyzing Minimal Failure-Causing Schemas in Siemens Suite". In: *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. 2016, pp. 35–38.

[YB85]   Shaula Yemini and Daniel M. Berry. "A Modular Verifiable Exception-Handling Mechanism". In: *ACM Trans. Program. Lang. Syst.* 7.2 (1985), pp. 214–243.

[YCP06]   C. Yilmaz, M. B. Cohen, and A. A. Porter. "Covering arrays for efficient fault characterization in complex configuration spaces". In: *IEEE Transactions on Software Engineering* 32.1 (Jan. 2006), pp. 20–34.

[Yil+14a] Cemal Yilmaz, Emine Dumlu, Myra B. Cohen, and Adam A. Porter. "Reducing Masking Effects in CombinatorialInteraction Testing: A Feedback DrivenAdaptive Approach". In: *IEEE Trans. Software Eng.* 40.1 (2014), pp. 43–66.

[Yil+14b] Cemal Yilmaz, Sandro Fouché, Myra B. Cohen, Adam A. Porter, Gülsen Demiröz, and Ugur Koc. "Moving Forward with Combinatorial Interaction Testing". In: *IEEE Computer* 47.2 (2014), pp. 37–45.

[Yu+13a]   Linbin Yu, Yu Lei, Mehra Nouroz Borazjany, Raghu Kacker, and D. Richard Kuhn. "An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation". In: *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. 2013, pp. 242–251.

[Yu+13b]   Linbin Yu, Yu Lei, Raghu Kacker, and D. Richard Kuhn. "ACTS: A Combinatorial Test Generation Tool". In: *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. 2013, pp. 370–375.

[Yu+15]   Linbin Yu, Feng Duan, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. "Constraint handling in combinatorial test generation using forbidden tuples". In: *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. 2015, pp. 1–9.

[Yu+18]   Min Yu, Feiyan She, Yuanchao Qi, Ziyuan Wang, and Weifeng Zhang. "A Revisit of Fault-Detecting Probability of Combinatorial Testing for Boolean-Specifications (P)". In: *The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018*. 2018, pp. 711–710.

[Zhe+16]   Wei Zheng, Xiaoxue Wu, Desheng Hu, and Qi-Hai Zhu. "Locating Minimal Fault Interaction in Combinatorial Testing". In: *Adv. Software Engineering* 2016 (2016), 2409521:1–2409521:10.

[ZHM97]   Hong Zhu, Patrick A. V. Hall, and John H. R. May. "Software Unit Test Coverage and Adequacy". In: *ACM Comput. Surv.* 29.4 (1997), pp. 366–427.

[Zhu96]   Hong Zhu. "A Formal Analysis of the Subsume Relation Between Software Test Adequacy Criteria". In: *IEEE Trans. Software Eng.* 22.4 (1996), pp. 248–255.

[ZZM14]   Jian Zhang, Zhiqiang Zhang, and Feifei Ma. *Automatic Generation of Combinatorial Test Data*. Springer Briefs in Computer Science. Springer, 2014.

Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview on related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR⁺19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question how digital and organizational techniques help to cope with physical distance of developers and [RRSW17] addresses how to teach agile modeling. Modeling will increasingly be used in development projects, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR⁺06, GKR⁺08, HR17] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR⁺09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG⁺14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, size, and number of the artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17]. To keep track of relevant structures, artifacts, and their relations in order to be able e.g. to evolve or adapt models and their implementing code, the *artifact model* [GHR17] was introduced. [BGRW18] explains its applicability in systems engineering based on MDSE projects.

An artifact model basically is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model therefore covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRvW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks. According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Smart semantic differencing operators have been defined for Activity Diagrams [MRR11a], Class Diagrams [MRR11d], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18].

We apply logic, knowledge representation and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests or find counterexamples using a theorem prover. And we have applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, HRR12] and based it on the core ideas of Broy's Focus theory [RR11, BR07]. Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11a, KLPR12] and city quarters [GLPR15] to optimize the operation efficiency and prevent unneeded CO_2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HR17]. In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKRS06, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12], we demonstrate how to systematically derive a transformation language in concrete syntax. [HMSNRW16] presents how to generate extensible and statically type-safe visitors. In [MSNRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLMSN+15b, HLMSN+15a]. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] respectively [Rum12, Rum13] and is implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation

points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11e] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98], and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HR17] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HR17] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11, HLMSN+15b, HLMSN+15a, HRW18, BEK+18a, BEK+18b, BEK+19] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages in [KRV14]. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15] and the concern-oriented language development approach [CKM+18]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HRW18, BEK+19]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08, HLMSN+15b, HLMSN+15a, HMSNRW16, HR17, BEK+18a, BEK+18b, BEK+19] and the backend [RRRW15, MSNRR16, GMR+16, HR17, BEK+18b]. In [GHK+15b, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV14] describes the roles in software development using domain specific languages. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages [HRW18]. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets. The derivation of internal DSLs from grammars is discussd in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. In [RRW13a], we introduce a code generation framework for MontiArc. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [NPR13] and with the robotics domain [AHRW17a, AHRW17b]. [GHK+07] and [GHK+08a] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14b] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. Co-evolution of architecture is discussed in [MMR10] and modeling techniques to describe dynamic architectures are shown in [HRR98, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HR17] that can even be used to develop modeling tools in a compositional form [HR17, HLMSN+15b, HLMSN+15a, HMSNRW16, MSNRR16, HRW18, BEK+18a, BEK+18b, BEK+19]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test

case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied for the development of CPS.

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12], and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08a] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for compositional reuse [BEK+18b], and applied it as a semantic language refinement on Statecharts in [GR11].

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied for the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12], autonomous driving [BR12a, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics

is discussed in [RRW13b, RRW14]. In [RRW13a], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition on contributing to systems engineering in automotive [GHK+08b], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration. To facilitate modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for machining based on these concepts [BKL+18]. Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20] and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14, RRW13a, RW18] as well as in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behaviour (2) based on information from previously stored and real-time monitored structural context and behaviour data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20] or UML/P based languages [MNRV19]. [MM15] describes a process how languages for assistive systems can be created.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view on the system design

allows to track who does what, when, why, where and how with personal data, makes information about it available via information services and provides support using assistive services.

Modelling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15, HR17] that perfectly fit robotic architectural modeling. The LightRocks [THR$^+$13] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17a, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible to different robotics middleware platforms.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK$^+$07, GHK$^+$08a]. [HKM13] describes a tool for delta modeling for Simulink [HKM$^+$13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW$^+$15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we introduce a framework for modeling the dynamic reconfiguration of component and connector architectures and apply it to the domain of cooperating vehicles. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO_2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP$^+$11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK⁺14, HHK⁺15b], Big Data, App, and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

Model-Driven Engineering of Information Systems

Information Systems provide information to different user groups as main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HR17], we developed several generators for such data-centric information systems. *MontiGem* [AMN⁺20] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN⁺20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular and incremental, handwritten and generated code pieces are well integrated [GHK⁺15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH⁺18].

[AHRW17a] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AMN⁺20] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BDH⁺20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.

[BDL⁺18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BEK⁺18a] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*, pages 75–82. ACM, January 2018.

[BEK⁺18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software*, 152:50–69, June 2019.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. Part of the Grand Challenges in Modeling (GRAND'17) Workshop, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software*, 149:437–461, March 2019.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a]   Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b]   Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15]   Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15]   Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14]   Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08]   María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09]   María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[CKM+18]   Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018.

[DEKR19]   Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[DGH+18]   Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[DGH+19]   Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 49(2):301–328, February 2019.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11a] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67-81, 2006.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze*

im Umbruch: Digitale Strategien für Services im Mobilen Internet, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015.

[GLRR15]   Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMN+20]   Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMR+16]   Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.

[GR95]   Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11]   Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12]   Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10]   Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13]   Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14]   Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a]   Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b]   Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13]   Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR⁺07]   Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR⁺09]   Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR⁺11]   Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12]   Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HLMSN⁺15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[HLMSN⁺15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMR⁺19]   Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, 18(1):1–60, July 2019.

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[HR04]   David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HR17]   Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR98]   Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR⁺11]   Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12]   Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11]   Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12]   Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12]   Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012.

[HRW15]   Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HRW18]   Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, 54:386–405, 2018.

[JWCR18]   Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35(6):40–47, 2018.

[KER99]   Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09]   Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KKR19]   Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications.

[KLPR12]   Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMR+20]   Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

[KMS+18]   Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[KNP+19]   Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[KPR97]   Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12]   Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[KPRS19]   Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE'19. Software Engineering Intelligence Workshop (SEI'19)*, pages 126–133. IEEE, November 2019.

[KR18]   Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.

[Kra10]   Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96]   Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14]   Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRRS19]   Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRS12]   Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012.

[KRV06]   Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KRV14] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modeling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM' 06)*. CoRR arXiv, 2014.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020.

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017.

[MNRV19]   Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[MRR10]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.

[MRR11e]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11f]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR11g]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.

[MRR13]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14a]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

[MRR14b]   Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[MRV20]   Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.

[MS17]   Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

[NPR13]   Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02]   Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94]   Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99]   Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01]   Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03]   Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14]   Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96]   Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95]   Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RR11]   Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRRW15]   Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRSW17]   Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

[RRW13a]   Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b]   Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14]   Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW⁺15]   Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96]   Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02]   Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03]   Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04]   Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11]   Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12]   Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum13]   Bernhard Rumpe. Towards Model and Language Composition. In Benoit Combemale, Walter Cazzola, and Robert Bertrand France, editors, *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013.

[Rum16]   Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17]   Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18]   Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[Sch12]   Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SHH+20]   Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.

[SM18]   Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0 (Extended Abstract): Integration of non-smart resources into cognitive assistance systems. *EMISA Forum*, 38(1):35–36, Nov 2018.

[SM20]   Claudia Steinberger and Judith Michael. *Using Semantic Markup to Boost Context Awareness for Assistive Systems*, pages 227–246. Computer Communications and Networks. Springer International Publishing, 2020.

[SRVK10]   Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13]   Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11]   Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12]   Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11]   Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.