Test-Driven Semantical Similarity Analysis for Software Product Line Extraction

Johannes Richenhagen
FEV GmbH
Neuenhofstraße 181
52078 Aachen, Germany
http://www.fev.com

Bernhard Rumpe
Software Engineering
Ahornstraße 55
52074 Aachen, Germany
http://www.se-rwth.de

Axel Schloßer
FEV GmbH
Neuenhofstraße 181
52078 Aachen, Germany
http://www.fev.com

Christoph Schulze
Software Engineering
Ahornstraße 55
52074 Aachen, Germany
http://www.se-rwth.de

Kevin Thissen
Software Engineering
Ahornstraße 55
52074 Aachen, Germany
http://www.se-rwth.de

Michael von Wenckstern
Software Engineering
Ahornstraße 55
52074 Aachen, Germany
http://www.se-rwth.de

ABSTRACT

Software product line engineering rests upon the assumption that a set of products share a common base of similar functionality. The correct identification of similarities between different products can be a time-intensive task. Hence, this paper proposes an automated semantical similarity analysis supporting software product line extraction and maintenance. Under the assumption of an already identified compatible interface, the degree of semantical similarity is identified based on provided test cases. Therefore, the analysis can also be applied in a test-driven development. This is done by translating available test sequences for both components into two I/O extended finite automata and performing an abstraction of the defined behavior until a simulation relation is established. The test-based approach avoids complexity issues regarding the state space explosion problem, a common issue in model checking. The proposed approach is applied on different variants and versions of industrially used software components provided by an automotive supplier to demonstrate the method's applicability.

CCS Concepts

•**Software and its engineering** → **Software product lines;** *Formal software verification; Software reverse engineering;* •**Computer systems organization** → *Embedded software;*

1. INTRODUCTION

Software Product Line Engineering [28] is a process of creating and maintaining a reusable platform for a particular application domain in a planned manner. The decision to establish and maintain a Software Product Line (SPL) in a company is mainly driven by the demand for efficient, meaning cheap and short, development cycles. The development of a generic item, being reused in different contexts, is more cost intensive as the implementation of a specific one. Therefore, in the industry the *clone-and-own* approach is often used instead, as it is an intuitive and easily available technique which does not enforce any structured planning or restrict project-driven modifications [10]. Nevertheless, *clone-and-own* is performed nonsystematic and issues with increased maintenance effort [10].

In the last years different approaches [34, 33, 14, 12, 38, 4] have been defined to support and systematize the general *clone-and-own* approach. In addition, we defined a reactive process in the tradition of Agile Product Line Engineering [9] to establish and maintain a SPL in a *clown-and-own* manner [36]. In this process the generic items of the SPL are software components and in the context of FEV GmbH these software components are defined based on PERSIST [40, 30], an in-house standard inspired by and compliant to AUTOSAR[1].

All listed approaches identify similarities between different products based on different artifacts, like requirements, models or source code and the overall performance of the suggested procedure depends on the quality of applied similarity metrics. Nevertheless, most of these frameworks use only a syntactical analysis method such as call tree graph [2], lines of code as atomic comparison elements [11], number of same traces to features in requirement documents [34], sequence call graph of artefact elements (e.g. functions, classes) [14], token- and character-based similarity of expressions [24] or structural information from the (Java) metamodel [20]. Some matching algorithms use a semi-semantical approach [38, 33], which weight syntactical parts differently depending on their semantical influence on the behavior; but the similarity calculation between different elements of the same element kind (e.g. statechart transition) is based on string comparison. In contrast to the mentioned approaches this paper presents a full-semantical-based similarity analysis technique based only on the components' behavior specified by their test definitions.

[1]http://www.autosar.org/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '16, September 16 - 23, 2016, Beijing, China

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4050-2/16/09...$15.00

DOI: http://dx.doi.org/10.1145/2934466.2934483

[RRSSTW16] J. Richenhagen, B. Rumpe, A. Schloßer, C. Schulze, K. Thissen, M. von Wenckstern:
Test-driven Semantical Similarity Analysis for Software Product Line Extraction.
In: International Systems and Software Product Line Conference (SPLC '16), pages 174-183, Beijing, China, 2016. ACM.
www.se-rwth.de/publications/

Semantical clone analysis techniques can support the identification of similar features, however analyzing semantical aspects of software components is a challenging task [32]. In the context of model checking algorithms [1], the state space explosion problem [27] provides most difficulties. In theory, semantical analyses can already be performed, but struggle with exponential complexity. Therefore, they are often hardly practical for industry-scaled software systems.

In the past we have already developed a model checking approach evaluating functional backward compatibility between Simulink models [37]. This approach, even when combining different techniques like syntactical clone detection on the extracted control flow graph or a parallel state space calculation to handle the state space explosion problem, could not avoid to become impractical for larger models.

Thus, in contrast to the previous white-box approach [37], this paper proposes a technique to perform an automated semantical similarity analysis based on an approximation of the provided behavior: related test cases.

In a test-driven development [3] test cases are provided already in an early state of the development process and, related to the given test coverage, represent a good approximation of real behavior. In addition, this approximation does not define internal variables and so it avoids the impending state space explosion.

In the following, we will describe how test specifications are transformed to Input/Output Extended Finite Automaton (I/O-EFA) and how a stepwise functional abstraction is used to measure the semantical similarity between two software components. Both steps are an extension of the compatibility analysis algorithm proposed by us in the past[37]. The proposed technique is evaluated on three different software components and results are derived in a practical time frame, while the derived semantical similarity measure is comparable to expert estimations.

The outline is as follows: Sect. 2 summarizes the necessary background to be able to follow the upcoming concept. In Sect. 3 the approach is described in all details, while in Sect. 4 its implementation is summarized. Sect. 5 describes the evaluation of the approach and in Sect. 6 similar approaches are discussed. Finally, a conclusion and an outlook regarding future work are provided.

2. FOUNDATIONS

This section provides an overview about the compatibility check framework, which will be extended in this paper by the semantical similarity analysis tool, and the classification tree method. Classification trees will be used as test specifications, which are transformed to I/O-EFAs to derive semantical similarities between different test specifications.

2.1 Compatibility Analysis Framework

The behavioral compatibility analysis framework [37] analyses with model checking techniques whether the behavior of one Component & Connector (C&C) software model, e.g. graphical Simulink or textual MontiArc [19] model, is compatible to another one. Therefore it transforms C&C components via control flow graphs to Input/Output Extended Finite Automata (I/O-EFAs) [35, 41], which contains a set of states S, set of internal variables D, sets of in-/output variables U and Y as well as a set of transitions E ($o_e \xrightarrow[y=h_e(d,u);d=f_e(d,u)]{[g_e(d,u)]} t_e$) going from a start state $o_e \in S$ to an end state $t_e \in S$ when its guard condition is satisfied by producing outputs and updating data according to its specified output $h_e : D \times U \rightarrow Y$ and update $f_e : D \times U \rightarrow D$ functions. By unfolding all I/O-EFAs' internal variable combinations to new states and moving the output calculation from transitions to states, these I/O-EFAs become Input/Output Transition Systems (I/O-TSs); this step often causes the state space explosion problem [1]. The simulation preorder algorithm [17] creates a binary relation between the states of both I/O-TSs: state q simulates state p when all possible (output-value-)traces starting from q are a superset of all possible (output-value-) traces starting from p. If I/O-TS A's start state simulates I/O-TS B's start state then the behavior of deterministic automaton A includes also the deterministic behavior of B; if additionally B also simulates A then both deterministic automata are in a (bi)simulation relation [17]. If the C&C model A is not compatible to the C&C model B, the compatibility check framework uses the in-/output trace of the simulation preorder algorithm to generate a counterexample for the input models. MontiArcAutomaton [31], an extension of MontiArc with I/O^{ω}-automata [35], is used for the technical representation of I/O-EFAs and I/O-TS in the compatibility checker.

2.2 Classification Tree Method

The classification tree method [18], a partition testing [26] approach, supports the systematic creation of black box tests. This is done in two steps: first, the input/output (I/O) domain of the software under test is divided into relevant aspects, which mostly corresponds to the software interface's I/O parameters. Then, these relevant aspects are divided into disjoint classifications where each of them may contain various classes. This gradual partitioning of I/O domain creates a tree structure.

Second, for each created classification an arbitrary class is chosen to define a test; various combinations of selected classes lead to different tests. This is usually realized with combination tables having classification tree as a head (one class is one column) and tests as individual entries.

3. SEMANTICAL ANALYSIS

This concept, deriving a similarity measure by a stepwise abstraction of the provided automata, describes a method to automatically derive the level of semantical similarity of two software components. In the following, the suggested approach be explained in detail:

Fig. 1 illustrates the basic steps to derive a similarity measure based on two test specifications from different software components. In a first step (①), structural compatibility needs to be ensured, before semantical similarity can be identified. Therefore, the similarity of two different interfaces needs to be investigated and adequate port matches needs to be identified before related behavior can be analyzed. The structural similarity analysis of software components' interfaces is a topic currently investigated by us in parallel, but this paper focuses only on the semantical similarity aspects. Therefore, it is assumed that step ① is done either due to a separate analysis technique or performed manually. In general to continue with step ② a set of compatible port matches needs to be identified. Otherwise, no similarity is provided. In ② test specifications are transformed to I/O-EFAs. This current approach creates no internal variables, and so, the update function f_e can be ignored. Steps ③ and ④ use the

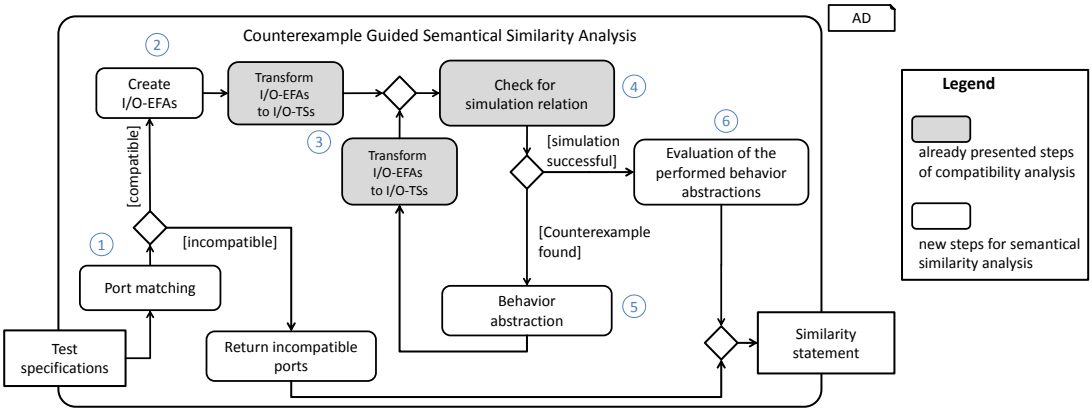

Figure 1: Overview of the analysis of semantical similarity.

I/O-EFA to I/O-TS transformation and an adapted simulation algorithm of the compatibility analysis framework (c.f. Subsect. 2.1). If the compatibility analyzer provides a counterexample in ④, this example is used to abstract the behavior of the automaton which could not be simulated. Steps ③ to ⑤ are repeated until no further counterexamples can be derived. Based on the amount of abstractions performed in step ⑥ an overall similarity measure between both test specifications is derived.

The example used throughout this paper to illustrate the concept's details consists of four different variants of a light system control unit. These four variants describe partly overlapping behavior, therefore a semantical similarity between these variants can be identified. All variants define the same interface, consisting of two inputs and one output. The indicator if a control button is currently pressed or not and the current volt value of the connected battery are the inputs and the expected volt value for the attached light bulb is the output. The first very simple light system *BasicHold* transfers directly the ingoing volt value of the battery to the light bulb, if the button is pressed and zeroes otherwise.

The second light system *LimitHold* represents nearly the same functionality, but the transferred volt value is limited to 5 which is half of the max. value. A more advanced variant (*BasicPush*) does not require to hold the button, but transfers the full volt value if the button is pressed once. If the button is pressed twice, the light is off again and the volt value is not transferred. Finally, the most advanced variant (*StepwisePush*) provides limited light (max. 5 volt) after the pressing the button once and provides full light after pressing it twice. Pressing it a third time, the light goes off again.

The described behavior of the different variants is illustrated in a classification tree (CT) in Fig. 2, whereby the test sequence for the most advanced variants are illustrated separately. To save space not all possible test sequences are shown, but only the connection between the different test steps are illustrated in CT format. For *BasicPush* and *StepwisePush* a pressed button is indicating a state change, meaning another step in the test sequence. In addition, test sequences indicating a loop are shown (second and following ones for *BasicPush* and *StepwisePush*). As demonstrated in Fig. 1 given test specifications are transformed into I/O-EFAs in a first step. As we believe that a well-structured I/O EFA provides a better illustration of the described functionality the result of the transformation is already shown in Fig. 3, before the transformation will be explained in detail.

In the following, the transformation from test specifications to I/O-EFAs is described first. Then the details of the semantical similarity analysis are explained.

3.1 Test Specifications

This section explains necessary information needed to be provided by test specifications in order to derive meaningful automata.

A test specification includes one or more test sequences for exactly one software component. Every test specification must also include the components' interface which defines for each in-/output parameter its data type as well as an ordered range of its values containing all representatives of equivalence classes this parameter belongs to (a common step done during the classification tree method). These representatives are boundary values of their equivalence classes. E.g. a valid range specification for a double input parameter $VLamp$ accepting values 0, 5, and 10 is, for example, $VLamp = double\{0, 5, 10\}$.

Fig. 2 already illustrates four test specifications for different light system controlling components. To describe an inner loop the last two test steps of a test sequence needs to define the same behavior, as demonstrated for *BasicPush* and *StepwisePush*. It is important that the test specifications specify always deterministic component behavior; otherwise they cannot be analyzed with the simulation algorithm presented in [37]. Underspecified behavior (normally provided by test specifications) is replaced with a deterministic, but unknown, behavior during the transformation. The procedure is described in the following.

3.2 Automaton Construction

This section explains how to construct I/O-EFA automata based on components' test specifications. Since tests always define components' in-/output behaviors as finite sequences, the I/O-EFA definition in Subsect. 2.1 will be extended by a final state $f_0 \in S$ expressing no further defined in-/output behavior (the I/O-EFA maybe behave arbitrary). In the result shown in Fig. 3 the final state is omitted, as the described behavior intends a closed loop. If the defined behavior shall be interpreted as a closed loop (meaning corresponding transition target the start state instead of the final state) is

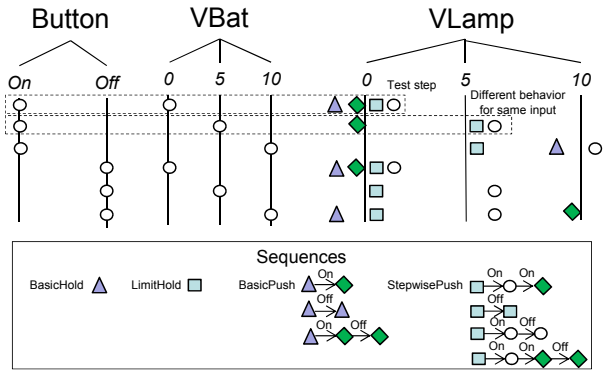

Figure 2: Test sequences for different simplified light control systems.

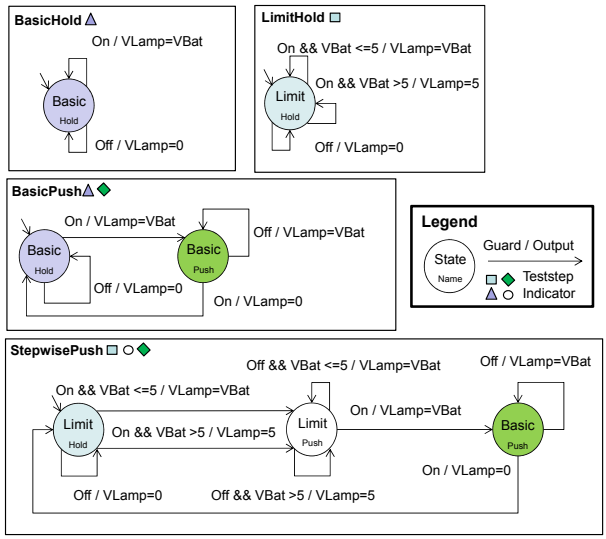

Figure 3: Derived I/O EFAs based on the test specification shown in Fig. 2.

part of an additional transformation step described later.

Test specifications are transformed into I/O-EFAs, by creating for each output one automaton in the following way:

In-/Output variables: I/O-EFA's in- and output variables with their data types and ranges can be directly derived from the component's interface in the test specification.

Internal variables: each I/O-EFA has an empty set of internal variables because tests do not specify internal variables either.

Transitions: for each test step in a test sequence a transition is created; its guard condition is a conjunction with all input parameters' values of the used tests, its output function assignment contains the values of the test's corresponding output parameters and its source as well as target state depends on the test steps occurrence in the related test sequence. If the last two steps of a test sequence are the same, the test step is interpreted as an inner loop. In this case start and target state of the transition are the same.

States: the set of states also depends on the amount of different test steps in the test sequences. Therefore, for each transition $E_i \in E$, created from a test step i of a test sequence, its own new target state $s_i \in S$ is created. These

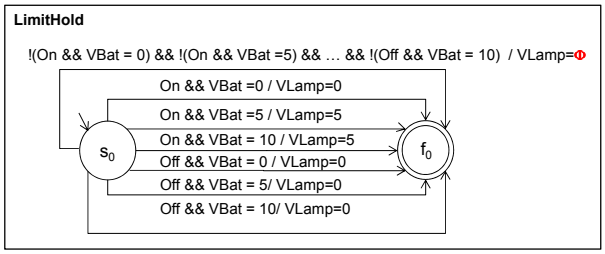

Figure 4: Formal representation of an derived I/O-EFA without loop and additional transitions to avoid underspecified behavior.

transitions have the respective target state of the transition $e_{i-1} \in E$ as a source state which was created in the previous test step $i-1$. In this way, a test sequence is represented as an I/O-EFA transition sequence.

It is possible that test sequences are partially identical meaning that two transition sequences contain a subset of identical transitions. For this reason, only transitions with a different source state or a disjunct guard condition will be added to the I/O-EFA in order to avoid creating a non-deterministic automaton.

Test specifications are often underspecified by not defining an expected output for every possible input. Therefore, the created I/O-EFA would be incomplete since not every possible input is covered by an outgoing transition. In order to fully specify the I/O-EFA, for each state an additional transition targeting the final state is added. As guard condition the conjunction of all negated guards of all other outgoing transitions, and as output assignment the special character Φ, representing undefined or arbitrary output, is added.

To illustrate the I/O-EFA construction from a test specification, in comparison to the automata represented in Fig. 3, Fig. 4 shows the concrete I/O-EFA for the software component *LimitHold*. In this example, additional transitions representing underspecified behavior (the transition guard will never become true as the test provides no underspecification) are shown. In addition no loop is defined, but all transitions end in the final state.

The previously constructed I/O-EFAs have guards with only constant input values as well as output assignments only returning fixed values, because only single fixed values were used as in-/output parameters in these test specifications. However, for continuous data types mostly different representatives for in-/outputs are used in different tests describing similar or equal behavior. In addition, intervals between two test values (equivalence class representatives) are interpreted as underspecification. Therefore, solely comparing the continuous input parameters to fixed values and evaluating the output assignment of fixed values is not sufficient for further analysis. Instead, the I/O-EFAs need to be enhanced by not explicitly defined behavior.

In a first step the guards of the transitions are extended by interval checks on input ports with continuous data types. These intervals are created by using the range values' limits that are specified in the interface of the test specification (due to the gradual partitioning of the I/O domain). Each interval is formed of a value from the value range as the lower limit and its subsequent larger value as upper limit. In this way, all input values between the values, that are

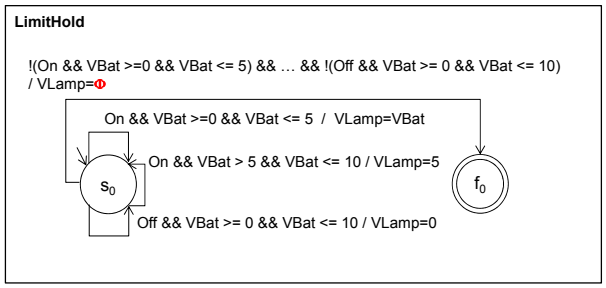

Figure 5: I/O-EFA with interpolated output.

explicitly used in the tests, are also covered by the transitions. Since the range values between the limits are elements of a common equivalence class, it can be assumed that they cause a similar behavior [7]. The behavior that is defined by the transition can therefore be assigned to them.

Next, the output assignments of the transitions of I/O-EFAs for output ports that have continuous data types are expanded. The fixed values in these output assignments are replaced by continuous functions. These continuous functions are in turn calculated by a linear interpolation based on the border values of the derived guard interval and the related output interval defined in the test specification.

Fig. 5 shows the I/O-EFA with interpolated behavior of test specification for *LimitHold*.

By removing contradictory transitions and interpreting test specifications as loops (final state equals start state for all transitions which do not represent underspecified behavior), the automaton described in Fig. 3 can be derived.

In case of a larger set of states, the interpretation of a loop is more difficult; for example, for *BasicPush* or *StepwisePush* (see Fig. 3) the target of a loop is ambiguous: if a repetition of the same behavior is not stated explicitly in the test sequence, the target state of the loop cannot be identified.

3.3 Counterexample Guided Abstraction Similarity Metric

The CEGAS (Counterexample Guided Abstraction Similarity) metric analyzes similarity of an I/O-EFA B to an I/O-EFA A by creating a simulation relation between the I/O-EFAs' initial states. For this purpose, the behavior in which B differs from A is identified and removed from B iteratively based on provided counterexamples. The CEGAS metric is inspired by the CEGAR (Counterexample Guided Abstraction Refinement) paradigm [6]. However, instead of concretizing abstract behavior, concrete behavior is abstracted until equal functionality under the established modifications is identified. The compatibility framework's simulation algorithm provides a counterexample, if I/O-EFA B cannot be simulated by I/O-EFA A. This counterexample is used to abstract the I/O-EFA B - by replacing the output function of the transition, not being simulated by A, with ϕ - before the simulation algorithm checks them again. The character ϕ is introduced to define an undefined and therefore arbitrary behavior. The simulation algorithm has been modified to take ϕ into consideration during the simulation. If the output function of a transition is defined as ϕ, every possible output is considered as equal. These new simulation and subsequent removal steps, reducing the defined behavior of B, are repeated until a simulation relation is established.

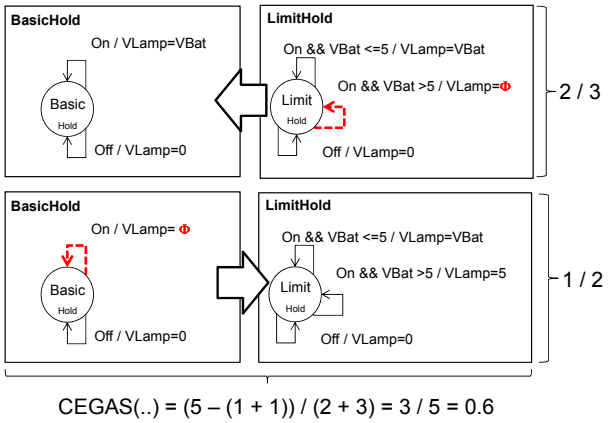

Figure 6: Application of CEGAS metric to analyze the similarity between *BasicHold* and *LimitHold*.

The semantical similarity between two components with the direction A to B (B is simulated by A) is the amount of transitions with defined outputs divided by the overall amount of transitions in the modified I/O-EFA B. As the simulation relation is not a symmetrical one, the derived degree would also be directed. By establishing a simulation relation from B to A due to the abstraction of A both simulation directions can be evaluated to derive an undirected similarity measure. Let A, B original I/O-EFAs and A', B' I/O-EFAs abstracted during an unbounded simulation; the symmetric degree of similarity $G_{CEGAS}(A, B)$ between A and B is calculated as follows:

$$G_{CEGAS}(A,B) = \frac{|M_{CEGAS}(A',B')|}{|M_{all}|}$$
$$:= \frac{\{e|e \in E_{A'} \cup E_{B'}, h_e \neq \phi\}}{\{e|e \in E_A \cup E_B\}}$$

Hence, $M_{CEGAS}(A', B')$ contains all transitions with specified behavior (no ϕ) of A' and B', while M_{all} contains all transitions of the unmodified automatons A and B.

The application of the CEGAS-metric is demonstrated in Fig. 6. For the sake of simplicity, this figure shows, similar to Fig. 5 the representation from Fig. 3 with no final state or further transitions representing underspecified behavior. *BasicHold* can simulate *LimitHold*, if the limiting transition (red one in top right automaton) is abstracted, while *LimitHold* can simulate *BasicHold* if general output assignment regarding VLamp (red transition in bottom left automaton) is abstracted. Applying the symmetric CEGAS metric for both automata a similarity of 0.6 is derived.

While the CEGAS metric provides good results for stateless functionality (only one state), for stateful functionality, like for *BasicPush* or *StepwisePush*, a related comparison could derive misleading results. Problems occur if additional states divide similar or same subgraphs of the automata. This is demonstrated in Fig. 7, where semantical similarity analysis of *BasicHold* and *BasicPush* is shown.

In this example a simulation from *BasicPush* to *BasicHold* can only be achieved if the whole automaton is abstracted. In consequence the degree of similarity is reduced significantly. Only if the derived automaton was interpreted as loopless, which is not the standard interpretation for an automaton consisting of start and end state only (as described

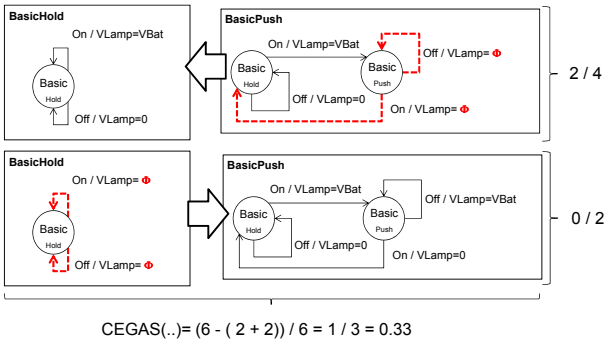

Figure 7: Application of the CEGAS metric to analyze the similarity between *BasicHold* and *BasicPush*.

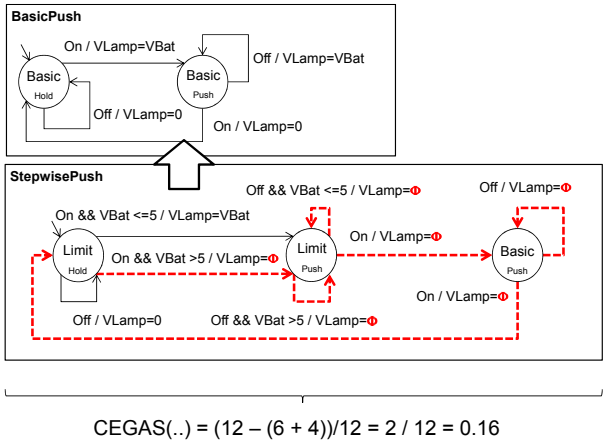

Figure 8: Application of the CEGAS metric to analyze the similarity between *StepwisePush* and *BasicPush*.

in Subsect. 3.2), a similarity of 0.66 could be derived. This result again would be too high, a similarity of 0.5 is expected. By performing an n-bounded simulation it can be identified that a 1-bounded symmetric simulation relation is given while a 2-bounded simulation relation is not provided for both directions. Nevertheless, in Fig. 8 the comparison between *StepwisePush* and *BasicPush* describes a constellation, for which both currently described approaches cannot derive a correct similarity measure.

Even the state *BasicHold* is quite similar to the state *LimitHold* (see Fig. 6: 0.6) and the *BasicPush* states are equal, the CEGAS metric can only derive a similarity of 0.16. A n-bounded approach is not detecting any similarity if the start state cannot be changed to identify subgraphs of both automata establishing a n-bounded simulation relation (state *BasicPush*, 1-bounded).

If the idea of identifying most applicable subgraphs is combined with the CEGAS approach (i.e., identify subgraphs with less needs for abstraction until a n-bounded simulation relation can be established) a similarity relation of 0.44 is derived, as shown in Fig. 9.

Starting at state *BasicPush* less abstractions needs to be performed to establish a 3-bounded simulation. Still, for simulating *BasicPush* with *StepwisePush* it is necessary to abstract all transitions from state *BasicPush*, while this is only necessary to establish a 1-bounded simulation relation between the states *LimitPush* and *BasicPush*. In addition, different subgraphs and their similarities based on different boundaries (2 and 1) are illustrated in Fig. 9. These candidates can be taken into consideration if a general behavior should be extracted based on a similarity analysis. Considering the described examples, a n-bounded counterexample guided abstraction metric is defined by:

Let A, B be original I/O-EFAs and A'_{n,a_i}, B'_{n,b_j} be abstracted I/O-EFAs based on a n-bounded simulation starting at states a_i and b_j; the symmetric degree of similarity $G_{CEGAS_n}(A,B)$ between A and B is calculated as follows:

$$G_{CEGAS_n}(A,B) = \max_{i,j}\left(\frac{|M_{CEGAS}(A'_{n,a_i},B'_{n,b_j})|}{|M_{all}|}\right)$$

The highest similarity of all possible start state constellations between A and B is extracted for a given boundary n to define the similarity. This metric supports the identification of subgraphs and, thus, parts of the overall behavior of two different software components which are strongly similar.

Instead of considering the different boundaries separately and taking only the maximum into consideration all derived subgraphs can be included by applying the sieve formula of the principle of inclusion and exclusion [39]. Thereby, overlapping subgraphs are only considered once and each subgraph is weighted by its size in relation to the overall automaton. Nevertheless, the amount of additional calculations to derive all possible subgraphs and to identify all overlapping parts seems not worth the effort. Although, the supplemental calculations do not provide any additional hints for a proper extraction of a general behavior. In addition, the resulting similarity measures did not match our expectations as the size of the resulting subgraphs influenced the result significantly. Therefore, this approach is not discussed any further in this paper.

4. IMPLEMENTATION

The software prototype is implemented in Java and uses the compatibility analysis framework.

The implemented prototype proceeds the following four steps: test specifications are read from external files and converted into an internal intermediate format. I/O-EFAs must be generated from the internal intermediate format according to the presented methodology. The generated I/O-EFAs are modified and compared according to the provided parameter. The results from the comparisons of the I/O-EFAs are evaluated depending on the parameter and a similarity measure is issued.

In the prototype, these four steps are each implemented in a separate component in order to be able to easily exchange specific logic. For the determination of details in the construction and analysis of these I/O-EFAs, such as port mapping between both I/O-EFAs or selected parameters, the prototype receives a configuration as input including these details. Therefore, this data can either be defined manually or provided by a structural similarity analysis algorithm currently under development in parallel.

The implemented prototype can process test specifications in form of classification trees with their corresponding combination table of test sequences as presented in Subsect. 2.2. A specific format for classification trees is provided by CTE

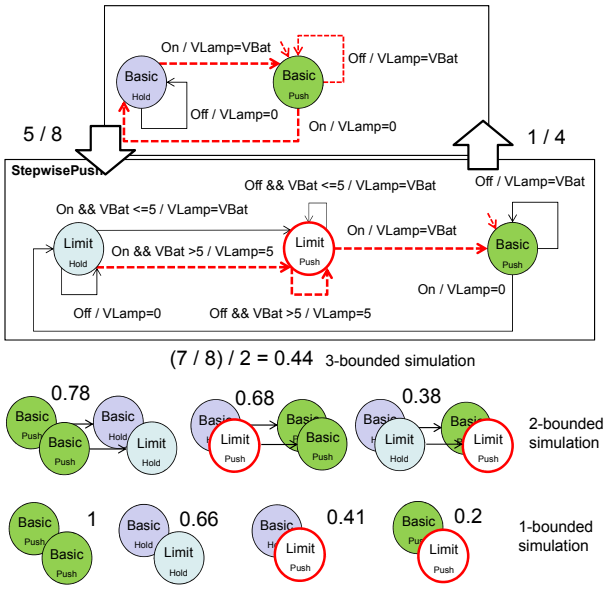

Figure 9: Application of the n-bounded CEGAS metric to analyze the similarity between *StepwisePush* and *BasicPush*.

files of the Classification Tree Editor[2] by Berner and Mattner; this tool is used for functional test specifications at FEV GmbH.

Analogous to the implementation of the compatibility analysis framework, Microsoft's SMT solver Z3 [25] is used to check satisfiability of guards and to compare output functions during the construction and analysis of I/O-EFAs.

5. EVALUATION

In this section we demonstrate the feasibility of the approach regarding runtime and regarding the quality of the derived similarity measure.

The evaluation of the proposed concept has been performed on preselected software components currently under development at FEV GmbH. To demonstrate the concept's feasibility a preselection of available software components has been performed according to the following criteria:

At least two extrinsically matches of the software component in relation to the evaluation of [36] are provided for such a software component. For at least two matches, test specifications have been specified in the supported format by different persons; therefore, these test specifications are different. For such a match parts of the software components interface needs to be compatible. Based on the available implementation the behavioral similarity can be evaluated manually in an adequate time frame. The behavior of the different software components defined for the compatible part of the interface needs to be similar to some degree or equal. For the following groups representative matches needs to be preselected for evaluation: discrete/non-discrete output, stateless/stateful behavior, additional ports/equal interface.

The last criteria focuses on evaluating different advanced mechanisms of the proposed concept and on highlighting known gaps and fields of improvements. The comparison between discrete and non-discrete outputs demonstrates the usage of interpolated intervals. Differences between stateless and stateful behavior allow identifying possible issues arising when switching between these behaviors. The comparison of similar interfaces enables highlighting issues arising while freezing additional ports in one of these interfaces.

For each of the mentioned groups an adequate match could be found. Two functions of a *virtual temperature sensor component* provide a stateless equal behavior regarding the matching parts of the non-discrete interface represented by different test specifications. Two functions of different *virtual park brake components* describe a discrete interface, whereby the calculation of the same output is influenced by additional ingoing signals for one component. Only for a more complex stateful behavior an adequate match with supporting test cases in the same format could not be provided. Therefore, a controller software component has been identified, whose evolution history provides adequate input. This example also highlights the usability of the proposed approach to improve the handling of evolution of one software component or of a whole software product line.

To protect the intellectual property of the analyzed functionality no details of the mentioned software components can be described.

Tab. 1 lists the mentioned matches (first column), their sizes regarding matching interface ports (fifth and sixth column), as well as their affiliation to the mentioned groups (second column). The fourth column list the amount of test steps available describing the behavior related to the identified port matches. The degree of behavioral similarity mentioned in the third column has been defined by an expert based on the available implementations similar to the estimations done by Berger, Rendel and Rumpe [4].

In a first step, an I/O-EFA has been created for each software component and the related test specification; the resulting duration of the transformation step is shown the last column of Tab. 1. The duration is mainly driven by the amount of SMT calls to correctly construct the automaton.

Tab. 2 represents the results on the similarity analysis. The results for the *Sensor* function are as expected and match the experts' suggestion. In the case of the *Park Brake* function one variant provides an additional input port. In both situations the behavior is identical, if the corresponding ingoing signal is fixed to a specific value. Therefore the more advanced variant A is capable of simulating variant B under specific conditions. The compatibility algorithm is capable of deriving this condition [37], but this approach can only be performed, if no abstraction is done and only for one additional port. If no condition for the additional port is derived, the simulation cannot be performed, as a non-determinism is provided during simulation. It is necessary to add the additional ports to the other automaton as well and extend the available transitions by corresponding conditions (and add new transitions) to remove possible non-determinism during simulation. How this is done is illustrated in Fig. 10 by a simplified example. Missing conditions and transitions regarding the Boolean parameter B are added without changing the behavior of automaton *OneInput*. Afterwards it is possible to apply the counterexample guided abstraction to simulate *OneInput* with *TwoInputs*. Sadly, in the given time frame we were not able to generalize and implement this procedure. Instead we applied the transformation steps manually to

[2]Newer version is called TESTONA, available at http://www.testona.net/en

Software Component	Group Affiliation	Similarity	Test Steps	In	Out	Dur. (ms)
Sensor (Var. A)	non-discrete, stateless, equal interface	33%	5	1	1	412
Sensor (Var. B)			3	1	1	268
Park Brake (Var. A)	discrete, stateless, additional ports	75%	17	3	2	3930
Park Brake (Var. B)			5	2	2	651
Controller (Ver. 1)	discrete, stateful, additional ports	90%	149	23	3	131702
Controller (Ver. 2)			165	24	3	167750

Table 1: Software components used during evaluation.

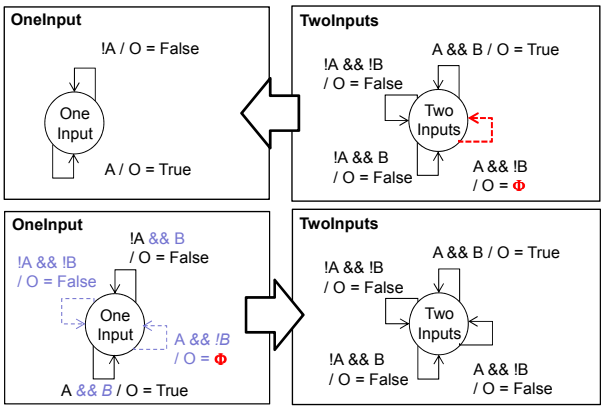

Figure 10: Handling of additional ports.

be able to derive a similarity measure without freezing the additional port. Under this condition it was able to derive an overall similarity of 75 %, a value equal to the expert suggestion.

The results for the different versions of the *controller component*, struggle with the same issue: the simulation from version 3 to 2 can only be applied if the additional port is restricted to a specific value. Again, by adapting the automaton manually, an equal similarity measure could be derived in the other direction. Summarizing the above the derived similarity measures are close to the expert estimations (as shown in Tab. 2) and, in addition, the necessary time frame to execute an unbounded counterexample guided abstraction similarity analysis is in a practical range.

Sadly, bounded similarity analysis was not necessary by the provided examples, as only parallel step sequences or slight modifications of single test steps have been introduced between versions 1 and 2. No in-between test steps have been introduced in the available sequences and, therefore, no issues, as drafted in Fig. 9, arise. Consequently, the feasibility of the bounded similarity analysis regarding runtime could not be analyzed. In addition, unequal interfaces could only be handled by manual modifications.

The described evaluation is only of a demonstrative character. It could be shown that for the shown cases runtime and precision of the measured similarity are in a practical range. In comparison to our evaluation of the compatibility analysis on Simulink models [37] the approach provides results in a suitable duration by avoiding a state space explosion based on internal variables. As before fixed-point data types are approximated by integer data types, but as long as all conditions and output calculations specified in the test cases can be represented technically the precision of the similarity analysis is not influenced.

SW comp.	Dur. (ms)	Derived Similarity	Expert Similarity
Sensor	4214	30 (0 / 60)%	33%
Park Brake	5283	88 (75 / 100*) %	75%
Controller	315784	97 (92,5 / 100*) %	90%

Table 2: Results of the similarity analysis (directed similarity in brackets,* with fixed ports).

Nevertheless, the small sample size of the available software components, the small amount of involved people defining the test specifications and the amount of available experts deriving similarity statements manually are significant threads to validity of any general statement. In addition, the proposed similarity measure is only applicable to identify branch-based differences, as the abstraction is currently done transition-wise. In consequence, a similarity between function $A + B$ and function $A + B + C$ is not identified. Internal logical loops can only be identified correctly if the test specifications are defined as requested. Normally, this is not the case and a manual transformation step needs to be performed to address this point.

6. RELATED WORK

Apart from our presented method, there are many related approaches for analyzing the behavior compatibility or similarity of software components.

Cordy et al.[8] apply CEGAR to improve model checking on software product lines. By defining SPL-specific abstractions the state space explosion problem is avoided in the majority of the evaluated cases.

Chaki et al. describe an automatic verification of C programs against specifications in the form of labelled transition systems (LTS) [5]. A method called *MAGIC* (Modular Analysis of Programs In C) is implemented, which can create a LTS based on the source code of a C program. After creating the LTS, the method then verifies by the use of a weak simulation relation, if the given specification is an abstraction of the LTS that was created from the C program. To avoid state space explosion, the verification of large software systems is divided into smaller verifications of individual procedures.

Fischbein et al. [13] define the software architecture of an entire product line and its individual products by modal transition systems (MTS) to be able to perform an analysis in order to check, whether a product is a valid implementation of the software architecture of its product line. However, the existing semantics of MTSs are not sufficed to distinguish valid implementations from invalid implementations in the context of software product line engineering [13]. The-

refore, new semantics for MTSs are defined which are based on the branching bisimulation relation [16] of LTSs.

In the general field of code clone detection techniques several approaches are defined to identify type 4 (functional) clones [32]: Gabel et al. define a scalable detection of semantical clones based on program dependency graphs [15]. The underlying graph similarity problem is reduced to a tree-based similarity problem by establishing an additional mapping to the related syntax. Other approaches focus on a l-length path matching on attributed directed graphs [22] or on the application of a lossy filter to prune the search space of underlying program dependency graphs [23]. Nevertheless, none of the approaches take test specifications as a suitable approximation of the defined functionality into consideration or perform the similarity analysis based on iterative simulation approaches, which promises very precise results.

In the software retrieval community a test-driven reuse cycle is considered [21]: promising candidates are identified based on interface specifications and afterwards a set of tests cases is executed on these candidates to retrieve the required functionality.

Reiss extends this idea to apply transformations on derived most promising candidates to match the specified test specifications [29]. For this purpose subparts of the available code are investigated. In relation to our approach test specifications needs to be executed and a detailed similarity measure cannot be derived.

7. CONCLUSION

This paper describes a complete approach to derive a semantical similarity measure based on test specifications by performing a counterexample guided abstraction. It is shown how test specifications are transformed to I/O-EFAs and how test sequences need to be defined to represent inner loops. A simplified example is introduced to illustrate the concept but also to highlight the possible issues by comparing two I/O automata derived from test specifications to derive a similarity measure based on a simulation relation. Together with a syntactical similarity analysis based on interfaces (parallel ongoing work) and a corresponding port matching, the defined approach suits the demands of the reactive process defined in [36] to identify new potentials for reuse. The evaluation illustrates the applicability of the concept on test specifications provided by an industrial context. Nevertheless, after the application of extrinsically and structural filters, the amount of available test samples was highly restricted due to access, file format and timing constraints. Furthermore advanced temporal aspects are currently not included in either the provided test specifications or the transformed automata. In addition, the suggested approach requires available test specifications as input. A white box semantical similarity analysis based on available Simulink models is not supported. The Simulink Design Verifier[3] generates test input to achieve a definable level of model coverage (up to MC/DC). Combining the features of the Design Verifier with the presented concepts enables us to derive test specifications from Simulink models semi-automatically and derive a semantical similarity measure in whitebox scenarios, without facing the state space explosion problem. In addition, the Design Verifier establishes a traceability between different test steps and blocks of the analyzed Simulink model. Combined with the high precision of the CEGAS metric common semantical parts could be identified and extracted from a Simulink model directly.

Furthermore, a much larger data set would be available for evaluation and, in addition, the currently introduced concept will be applicable in more scenarios.

[3] http://www.mathworks.com/products/sldesignverifier/

8. REFERENCES

[1] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[2] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel. Transitioning legacy assets to a product line architecture. In *Software Engineering ESEC/FSE'99*, pages 446–463. Springer, 1999.

[3] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 1 edition, Nov. 2002.

[4] C. Berger, H. Rendel, and B. Rumpe. Measuring the ability to form a product line from existing products. In *Variability Modelling of Software-intensive Systems*, 2010.

[5] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference for Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, 2000.

[7] J.-F. Collard and I. Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[8] M. Cordy, P. Heymans, A. Legay, P.-Y. Schobbens, B. Dawagne, and M. Leucker. Counterexample guided abstraction refinement of product-line behavioural models. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 190–201, 2014.

[9] J. Díaz, J. Pérez, P. P. Alarcón, and J. Garbajosa. Agile product line engineering - a systematic literature review. *Software: Practice and Experience*, 41(8):921–941, July 2011.

[10] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 25–34, Washington, DC, USA, 2013. IEEE Computer Society.

[11] S. Duszynski, J. Knodel, and M. Becker. Analyzing the source code of multiple software variants for reuse potential. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 303–307. IEEE, 2011.

[12] D. Faust and C. Verhoef. Software product line migration and deployment. *Software: Practice and Experience*, 33(10):933–955, 2003.

[13] D. Fischbein, S. Uchitel, and V. A. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA, ISSTA 2006 workshop*, pages 39–48. ACM, 2006.

[14] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 391–400, Washington, DC, USA, 2014. IEEE Computer Society.

[15] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *30th International Conference on Software Engineering*, pages 321–330, May 2008.

[16] R. Glabbeek and W. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[17] R. v. Glabbeek. The Linear Time-Branching Time Spectrum I - The Semantics of Concrete, Sequential Processes. In *Handbook of Process Algebra, chapter 1*. Elsevier, 2001.

[18] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[19] A. Haber, J. O. Ringert, and B. Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[20] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4):20, 2012.

[21] O. Hummel and W. Janjic. Test-driven reuse: Key to improving precision of search engines for software reuse. In *Finding Source Code on the Web for Remix and Reuse*, pages 227–250. Springer, 2013.

[22] J. Krinke. Identifying similar code with program dependence graphs. In *Eighth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[23] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, New York, 2006.

[24] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 163–172. IEEE, 2008.

[25] L. Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*. Springer, 2008.

[26] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[27] R. Pelánek. Fighting State Space Explosion: Review and Evaluation. In *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 37–52. Springer Berlin Heidelberg, 2009.

[28] K. Pohl, G. Böckle, and F. J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, 2005.

[29] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[30] J. Richenhagen, H. Venkitachalam, S. Pischinger, and I. A. Schloßer. Persist–a scalable software architecture for the control of diverse automotive hybrid topologies. In *15. Internationales Stuttgarter Symposium*, pages 37–56. Springer, 2015.

[31] J. O. Ringert, B. Rumpe, and A. Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Number 20 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2014.

[32] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[33] J. Rubin and M. Chechik. Combining related products into product lines. In *Fundamental Approaches to Software Engineering*, pages 285–300. Springer, 2012.

[34] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 101–110, New York, NY, USA, 2013. ACM.

[35] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996.

[36] B. Rumpe, C. Schulze, J. Richenhagen, and A. Schloßer. Agile Synchronization between a Software Product Line and its Products. In *Informatik 2015*, volume P-246 of *LNI*, pages 1687–1698. Bonner Köllen Verlag, 2015.

[37] B. Rumpe, C. Schulze, M. v. Wenckstern, J. O. Ringert, and P. Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *International Conference on Software Product Line (SPLC)*, pages 141–150, Nashville, Tennessee, 2015. ACM New York.

[38] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic variation-point identification in function-block-based models. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 23–32, New York, NY, USA, 2010. ACM.

[39] A. Steger. *Diskrete Strukturen 1. Kombinatorik, Graphentheorie, Algebra*. Springer, Berlin, 2. edition, 2007.

[40] H. Venkitachalam, J. Richenhagen, and S. Pischinger. A generic control software architecture for battery management systems. Technical report, SAE Technical Paper, 2015.

[41] C. Zhou and R. Kumar. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.