Steffen Hillemacher, RWTH Aachen University
Nicolas Jäckel, FEV Europe GmbH
Christopher Kugler, FEV Europe GmbH
Philipp Orth, FEV Europe GmbH
David Schmalzing, RWTH Aachen University
Louis Wachtmeister, RWTH Aachen University

17

Artifact-Based Analysis for the Development of Collaborative Embedded Systems

One of the major challenges of heterogeneous tool environments is the management of different artifacts and their relationships. Artifacts can be interdependent in many ways, but dependencies are not always obvious. Furthermore, different artifact types are highly heterogeneous, which makes tracing and analyzing their dependencies complicated. As development projects are subject to constant change, references to other artifacts can become outdated. Artifact modeling tackles these challenges by making the artifacts and relationships explicit and providing a means of automated analysis. We present a methodology for artifact-based analysis that enables analysis of heterogeneous tool environments for architectural properties, inconsistencies, and optimizations.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_17

[HJK+21] S. Hillemacher, N. Jäckel, C. Kugler, P. Orth, D. Schmalzing, L. Wachtmeister:
Artifact-Based Analysis for the Development of Collaborative Embedded Systems.
In: Model-Based Engineering of Collaborative Embedded Systems, pp. 315-331, Springer, Jan. 2021.
www.se-rwth.de/publications/

17.1 Introduction

Consistency of artifacts during the development of collaborative embedded systems

The development of collaborative embedded systems (CESs) typically involves the creation and management of numerous interdependent development artifacts. Requirements documents specify, for example, all requirements that a system under development must fulfil during its lifetime, whereas system architectures written in the Systems Modeling Language (SysML) [SysML 2017] enable system architects to describe the logical and technical architecture of the system. If the expected behavior of a system and its system components is also modeled in SysML, automatically generated test cases [Drave et. al. 2019] can be used to check the system for compliance with these system requirements. Accordingly, the creation of these development artifacts extends through all phases of system development and thus over the entire project duration. Consequently, different developers create system requirements, architecture, and test artifacts using diverse tools of the respective application domain. Therefore, all artifacts must be checked for consistency, especially if further development artifacts are to be generated automatically in a model-driven approach. For example, it must be ensured that all components that are mentioned in the system requirements or for which system requirements exist are also present in the system architecture, or that all values checked by a test case match the respective target values specified in a requirement.

Heterogeneous development tools

Another challenge that arises during the system development process for CESs is the use of different tools during different stages of the development project. As CESs aim to connect different embedded systems handling multiple tasks in different embedding environments, heterogeneous tools adapted to the application domain are also used to create them. Furthermore, practice has shown that new tools are introduced to the project and obsolete tools are replaced by new ones to meet the challenges that arise in different development phases whenever insuperable tool boundaries are reached. As a result, the project becomes more complex, as new tools create new dependencies and other relationships, a situation that is amplified by the fact that the number of artifacts and their interdependence during development constantly increases. Since these various development tools are often incompatible with each other and do not support relationship validation across tool

interfaces, we use artifact-based analyses to enable automatic analysis of relationships and architectural consistency.

Artifact-based analysis

To tackle this challenge, automating artifact-based analysis enables the system developers to model the artifacts created during a project and to automatically analyze their relationships and changes. Artifact-based modeling and analysis were originally developed for software projects [Greifenberg et. al. 2017], but with slight modifications, also offer a decisive advantage in systems projects [Butting et. al. 2018]. For this purpose, we introduce a project-specific artifact model that is adapted to the individual project situation and thus unambiguously models the artifacts that occur in the project and illustrates their relationships.

Application of artifact-based analysis

We show the application of artifact-based analysis using the example of DOORS Next Generation (Doors NG) and Enterprise Architect (EA). To this end, we create an artifact model that models the structure and the elements of the exports of Doors NG and EA, as well as their relationships. We then describe the extraction of the structures and prepare the extracted data for further processing by analysis. For this purpose, we have developed static extractors that convert the exports into artifact data (object diagrams). Finally, we model analyses using Object Constraint Language (OCL) expressions over the artifact metamodel and show the execution of corresponding analyses on the extracted data.

17.2 Foundations

In this section, we present the modeling languages and model-processing tools used in our approach and explain how to use these to describe artifacts and artifact relationships.

UML/P

A language profile of UML

The UML/P language family [Rumpe 2016], [Rumpe 2017] is a language profile of the Unified Modeling Language (UML) [UML 2015]. Due to the large number of languages involved, their fields of application, and the lack of formalization, UML is not directly suitable for model-driven development (MDD). However, it could be made suitable by restricting the modeling languages and language constructs allowed, as has been done in the UML/P language family. A textual version of UML/P that can be used in MDD projects was developed in [Schindler 2012]. The approach for the artifact-based

analysis of MDD projects uses the languages Class Diagram (CD), Object Diagram (OD), and OCL.

Class Diagrams in UML/P

Class diagrams for analysis

Class diagrams serve to represent the structure of software systems and form the central element for modeling software systems with UML. CDs are primarily used to introduce classes and their relationships. In addition, they can be used to model enumerations and interfaces, associated properties such as attributes, modifiers, and method signatures, as well as various types of relationships and their cardinalities. CDs can be used in analysis to structure concepts of the problem domain, in addition to being utilized to represent the technical, structural view of a software system—that is, as the description of source code structures [Rumpe 2016]. For this use case in particular, [Roth 2017] developed an even more restrictive variant of the UML/P class diagrams: the language Class Diagram for Analysis (CD4A). In the approach presented here, CD4A is used to model structures in model-based development projects.

Object Diagrams in UML/P

Object diagrams for representing problem domain concepts

Object diagrams are suitable for specifying exemplary data of a software system. They describe a state of the system at a concrete point in time. ODs may conform to the structure of an associated class diagram. Checking whether an object diagram corresponds to the predefined structure of a class diagram is generally not trivial. For this reason, [Maoz et. al. 2011] describes an approach for an Alloy-based [Jackson 2011] verification technique. In object diagrams, objects and the links between objects are modeled. The object state is modeled by specifying attributes and assigned values. Depending on the intended use, object diagrams can describe a required situation of the software system or represent a prohibited or existing situation of the software system. The current version of the UML/P OD language allows the definition of hierarchically nested objects in addition to the concepts described in [Schindler 2012]. This has the advantage that hierarchical relationships can also be displayed as such in the object diagrams. In this work, CDs are not used to describe the classes of an implementation, but when used for descriptions on a conceptual level, objects of associated object diagrams also represent concepts of the problem domain instead of objects of a software system. In our approach, object diagrams are used to describe analysis data — that is, they reflect the current state of the project at the conceptual level.

OCL

OCL for analysis

OCL is a specification language of UML that allows additional conditions of other UML languages to be modeled. For example, OCL can be used to specify invariants of class diagrams, conditions in sequence diagrams, and to specify pre- or post-conditions of methods. The OCL variant of UML/P (OCL/P) is a Java-based variant of OCL. Our approach uses the OCL/P variant only. OCL is used only in conjunction with class diagrams throughout this approach. OCL expressions are modeled within class diagram artifacts.

17.3 Artifact-Based Analysis

This section provides an overview of the solution concept developed for performing artifact-based analyses and is largely based on the work published in [Greifenberg 2019]. Before we present the analyses in more detail, let us define the terms artifact, artifact model, and artifact data.

Definition 17-1*: Artifact*

An artifact is an individually storable and uniquely named unit serving a specific purpose in the context of a development process.

Artifact definition

This definition focuses more on the physical manifestation of the artifact rather than its role in the development process. It is therefore less restrictive than the level characterization presented in [Fernández et. al. 2019]. Furthermore, the definition requires an artifact to be stored as an individual, referenceable unit. Nonetheless, an artifact must serve a specific purpose within a development process, making its creation and maintenance otherwise obsolete. On the other hand, the definition does not enforce restrictions on the integration of the artifact into the development process — that is, an artifact does not necessarily have to be an input or output of a certain process step. Artifacts may also exist only as intermediate or temporary contributions of a tool chain. Moreover, the definition largely ignores the logical content of artifacts. This level of abstraction enables an effective analysis of the artifact structure taking the existing heterogeneous relationships into account instead of analyzing the internal structure of artifacts.

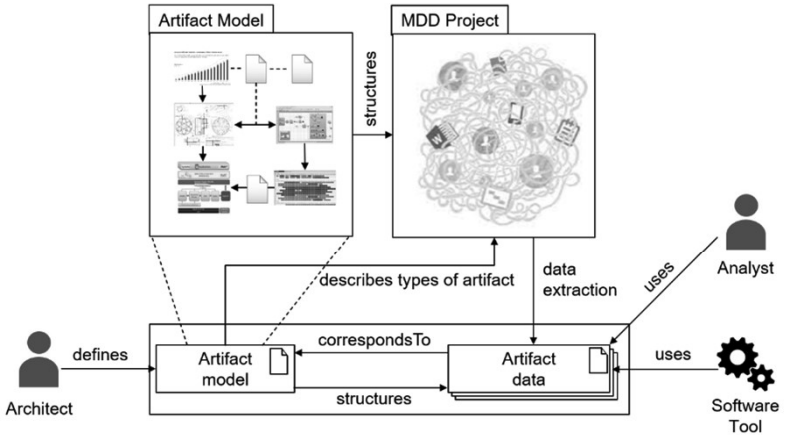

Fig. 17–2: *The role of an artifact model and corresponding artifact data within an MDD project*

Role of an artifact model and artifact data within an MDD project

An important part of the overall approach is the identification of the artifacts, tools, systems, etc. present in the development process and their relationships. Different modeling techniques provide a means to make these explicit and thus enable model-based analyses. Figure 17-2 gives an overview of the model-based solution concept. First, the types of artifacts, tools, and other elements of interest, as well as their relationships within a development process, must be defined. It is the task of an architect, who is well-informed about the entire process, to model these within an artifact model (AM). This model structures the artifact landscape of the corresponding process or a development project. The AM defines only the types of elements and relationships and not the specific instances; therefore, this model can remain unchanged over the entire life cycle of the process or the project unless new types of elements or relationships are added or removed. Moreover, once created, the model can be reused completely or partially for similar projects.

Definition 17-3*: Artifact model*

The artifact model defines the relevant artifact types and the associated relationship types of a development process to be examined.

Specific instances of an AM are called artifact data and reflect the current project status. Ideally, artifact data can be extracted automatically and saved in one or more artifacts.

Definition 17-4*: Artifact data*

Artifact data contains information about the relevant artifacts and their relationships that exist at a specific point in time in an engineering process. Artifact data are instances of a specific artifact model.

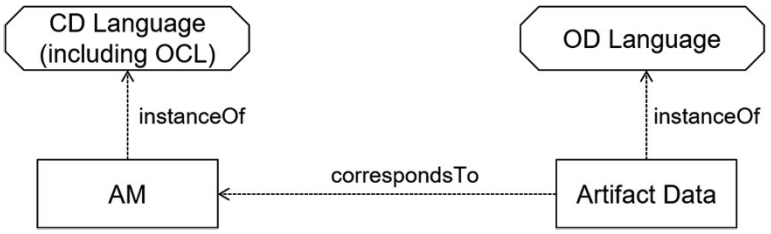

Fig. 17–5*: Modeling languages used for the artifact model and data*

Relationship of artifact data to an artifact model

Artifact data are in an ontological instance relationship [Atkinson and Kuhne 2003] to the AM. Each element and each relationship from the artifact data correspond to an element or a relationship type of the AM. The AM thus prescribes the structure of its artifact data. Figure 17-5 shows how this is achieved in terms of modeling techniques. During the artifact-based analyses, artifact data represent the project state at a certain point in time. Analysts and analysis tools use the artifact data to understand the current project state, to check certain relationships, create reports, and to check for optimization potential within the project. Ultimately, the goal is to make the software development process as efficient as possible. This approach is especially suited for checking the consistency of the architecture of model-driven software development projects or processes. It is capable of handling input models, model-driven development (MDD) tools—which themselves consist of artifacts—and handwritten or generated artifacts that belong to the end product of the development process. In such a case, the AM depends on the languages, tools, and technologies used in the development process or project. Thus, it is usually tailored specifically to a process or project.

Fig. 17–6*: Steps for enabling artifact-based analysis*

Enabling artifact-based analysis

In order to perform artifact-based analyses as shown in Figure 17-6, the first step is to create a project-specific AM. Once created, analyses based on the artifact data are specified. Finally, after the two previous steps, the artifact-based analysis can be performed.

Artifact Model Creation

Creation of an artifact model

The first step of the methodology is the creation of an AM. The AM determines the scope for specific analyses based on the corresponding artifact data. It explicitly defines the relationships between the artifacts and specifies prerequisites for the analyses. Additionally, using the CD and OCL languages, model-driven development tools can be used to analyze the artifact data. [Greifenberg 2019] presents an AM core and a comprehensive AM for model-driven development projects. If a new AM has to be created, or an existing AM has to be adapted, the AM core and parts of existing project-specific AMs should be reused. A methodology for this can also be found in [Greifenberg 2019].

Types of artifacts as central elements of an artifact model

The central elements of any AM are the types of artifacts modeled. All project-specific types of files are eligible to be contained in the AM. Artifacts can contain each other. Typical examples of artifacts that contain other artifacts are archives or folders in the file system. However, database files or models containing artifacts are also possible. Figure 17-7 shows the relevant part of the reusable AM core as presented in [Greifenberg 2019].

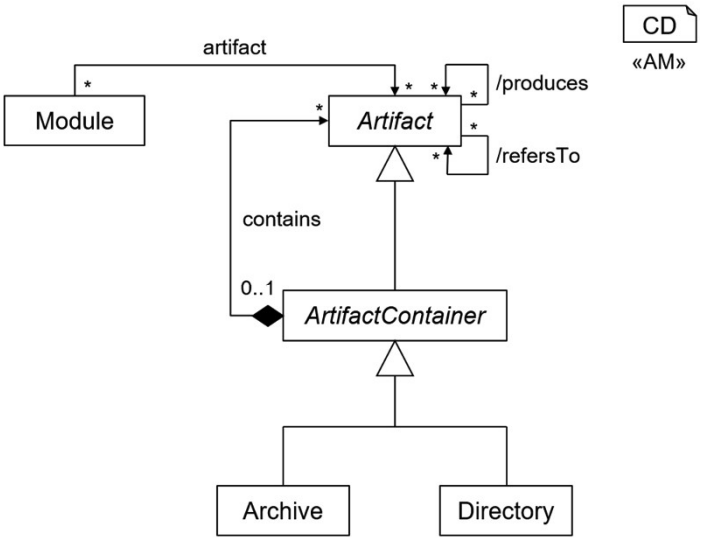

Fig. 17–7: *Reusable artifact model core as presented in [Greifenberg 2019]*

Composition of artifacts and artifact containers

In this part of the AM core, the composite pattern [Gamma et. al. 1995] ensures that archives and folders can contain each other in any order. Each type of artifact is contained in exactly one artifact container. If all available artifact types are modeled, there is exactly one type of artifact not contained by a container — that is, the root directory of the file system. Furthermore, artifacts can contribute to the creation of other artifacts (creates relationship) and they can statically refer to other artifacts (refers to relationship). These artifact relationships are defined as follows:

Definition 17-8*: Artifact reference*

If an artifact needs information from another artifact to fulfil its purpose, then it refers to the other artifact.

Definition 17-9*: Artifact contribution*

An existing artifact contributes to the creation of the new artifact (to its production) if its existence and/or its contents have an influence on the resulting artifact.

Refining an artifact model in project-specific extensions

Both relationships are defined in the AM as a derived association. Therefore, it is vital to specify these relationships further in project-specific AMs, while it is already possible to derive artifact data analyses from these associations. The specialization of associations is defined using OCL conditions [Greifenberg 2019], since the CD language is not suitable for this.

Specification of Artifact Data Analysis

Specifying analysis of artifact data

The second step of the methodology is the specification of project-specific analyses that are based on the AM created in the first step. These analyses must be repeatable and automated. They can be implemented either by one person, an analyst or analysis tool developer, or an analyst can specify the analyses as requirements for the analysis tool developer, who then implements an appropriate analysis tool. In this work, analyses are specified using OCL:

1. The CD language—used to model the AM—and OCL are well suited for use in combination to define analyses.

2. OCL has already been used to define project-specific analyses in [Greifenberg 2019]. Reusing familiar languages and providing example analyses shortens the learning curve for analysts.
3. OCL has mathematically sound semantics that enable precise analyses. Moreover, OCL expressions are suitable as input for a generator that can automatically convert them into MDD tools, thus reducing the effort for the developer of the analysis tool.

Artifact-Based Analyses

Executing artifact-based analysis

The third step in Figure 17-6 is the artifact-based analysis, which executes the previously specified analyses. This step is refined into five sub-steps. Each step is supported by automated and reusable tools. Figure 17-10 presents these steps and the corresponding tools.

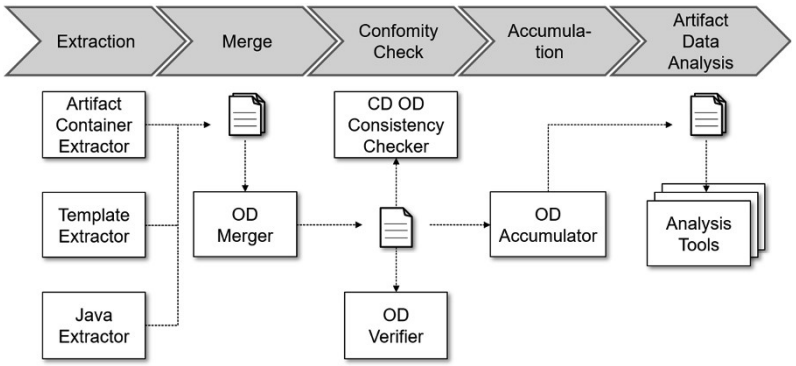

***Fig. 17–10**: Steps of artifact-based analysis with tools (rectangles), resulting files (file symbols), and the execution flow (directed arrows)*

Steps and components for performing artifact-based analysis

The first step in artifact-based analyses is to extract relevant project data. If stored in different files, the data must be merged. The entire data set is then checked for compliance with the AM. In the next step, the data is accumulated based on the specification of the AM, to ensure the derived properties are present for the last step, the artifact data analysis. [Greifenberg 2019] presents a tool chain that can be used to collect, merge, validate, accumulate, and finally, to analyze artifact data. The tool chain supports all sub-steps of the artifact-based analysis. The individual steps are each performed by one or more small tools that, combined, form the tool chain. The tools shown in Figure 17-10 are arranged according to the order of execution of the tool chain. The architecture as a tool chain is modular and adaptable. The primary data format for exchanging information between tools is

object diagrams. New tools can be added without having to adjust other tools. Existing tools can be adjusted or removed from the tool chain without the need to adjust other tools. Therefore, when using the tool chain in a new project, project-specific adjustments usually have to be made. The architecture chosen supports the reuse and adaptation of individual tools.

17.4 Artifact Model for Systems Engineering Projects with Doors NG and Enterprise Architect

Artifact-based analysis of Doors NG and Enterprise Architect

To demonstrate the practicability of the artifact-based approach, this section describes an example of artifact-based analysis of systems engineering projects with textual requirements and logical architecture components in SysML. Doors NG and Enterprise Architect are commonly used tools for these purposes. Doors NG enables engineers to define and maintain requirements in a collaborative development environment. Enterprise Architect is a solution for modeling, visualizing, analyzing, and maintaining systems and their architectures. Standards, such as UML and SysML, are supported. In our example, we focus on the definition of requirements in Doors NG and the modeling of systems and their components in Enterprise Architect. Here, system components are modeled with Internal Block Diagrams (IBD) and corresponding Block Definition Diagrams (BDD) from the SysML standard.

17.4.1 Artifact Modeling of Doors NG and Enterprise Architect

Creation of an artifact model for Doors NG and Enterprise Architect

The creation of the artifact model for this example includes the identification of artifact types used in the project as a first step. Since, in this example, we consider two tools whose files cannot be read directly via an open standard, suitable exchange formats must first be identified. The XML-based XMI exchange format, which is supported by Enterprise Architect as a tool-independent exchange format, is therefore taken as the exchange format for Enterprise Architect. Furthermore, a ReqIF export is used for the cloud-based data format of Doors NG for information exchange, which also enables a cross-tool exchange of requirements. The challenge here is that the requirements stored in the development tools are no longer present as individually stored units, but rather as what are referred to as artifact containers (cf. Figure 17-7), in which several development artifacts— which must first be identified and extracted for subsequent

analysis—have been combined. Once these basic artifact types (named "EA Export" and "Doors Export" in the artifact model in Figure 17-11) have been identified, the relevant information contained in the exports must be modeled and related.

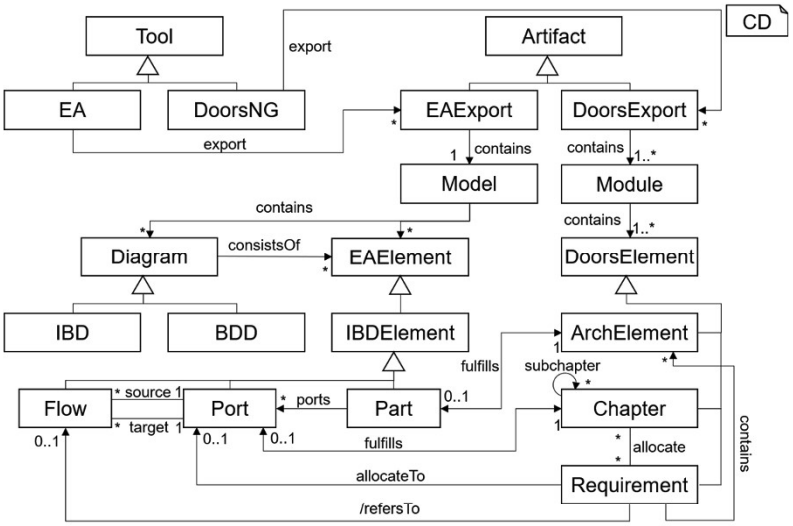

***Fig. 17-11**: Artifact model for exports of Doors NG and Enterprise Architect, as well as their relationships*

Modeling exports of Doors NG and Enterprise Architect

In the XMI export created by Enterprise Architect, exactly one model for the overall system modeled in the EA project is exported. This model contains any number of diagrams and elements (named *Diagram* and *EAElement* in the artifact model of Figure 17-11). Furthermore, each diagram has any number of elements, represented in the class diagram of the artifact model under consideration by the *consistsOf* association. Since the example considered is limited to architectural elements, not all diagram types of SysML are modeled in the artifact model; only the structural diagrams relevant for the logical architecture are modeled in the form of the Internal Block Diagram (IBD) and the Block Definition Diagram (BDD). A decisive advantage of the artifact models is that not all possible artifacts have to be modeled; the model can be limited to the artifacts relevant for the analysis. Similarly, only signal flows and parts—as the internal representation of ports in EA—are defined for the example under consideration. In the BDD, only the block is modeled as a relevant diagram element and all relationships in the BDD are no longer displayed. The ReqIF export of Doors NG is also represented as an artifact in the artifact model. Each *DoorsExport* contains at least one,

but otherwise any number of modules that also contain one or more *DoorsElements*. In this context, a mixture of *Chapters*, *Requirements,* and *ArchElements* serve as specialized *DoorsElements*.

17.4.2 Static Extractor for Doors NG and Enterprise Architect Exports

Static extraction of artifact data

To verify automatically that the current project complies with the architecture defined, all elements of the artifact model must be loaded from the two exports. To achieve this, we implemented static extractors, which parse the exports and load relevant information into our internal representation. For this purpose, the extractor transforms relevant data into an object diagram — that is, the artifact data. This workflow is shown in Figure 17-12. The artifact data extracted from the tool exports is tool-specific at first and needs further consolidation. This means that tool-specific artifact data is merged into a consistent data set (object diagram): the artifact data of the system. During this step, associations between objects of different diagrams are constructed (extracted from name references) and objects of the same type and name are merged automatically. Relationships between elements of different exports are constructed during this step. The resulting object diagram gives a view of the current project architecture and enables analysis.

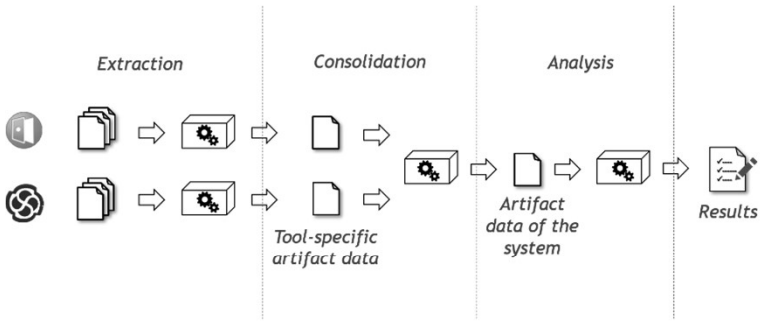

Fig. 17–12: *Tool chain workflow from artifact data extraction to analysis*

17.4.3 Analysis of the Extracted Artifact Data

Analysis of extracted artifact data

After the extraction and consolidation of artifact data, artifact-based analyses can be defined on the previously constructed project-specific artifact model and executed on the merged and consolidated artifact data. However, analyses are executed only on artifact data that conforms to the artifact model. Therefore, in a first step, the tool chain

checks whether the artifact data is an instance of the artifact model and executes further well-formedness constraints. If the merged artifact data is well-formed and conforms to the artifact model, then defined analyses are executed on the artifact data. We model analyses as constraints of OCL over the defined artifact model. This enables us to define analyses without deeper programming experience and to execute analyses automatically on the extracted data without having knowledge of the internal data structure of the analysis tool. To this end, our tool chain transforms modeled analyses into machine code and executes this code on the internal representation of the artifact data.

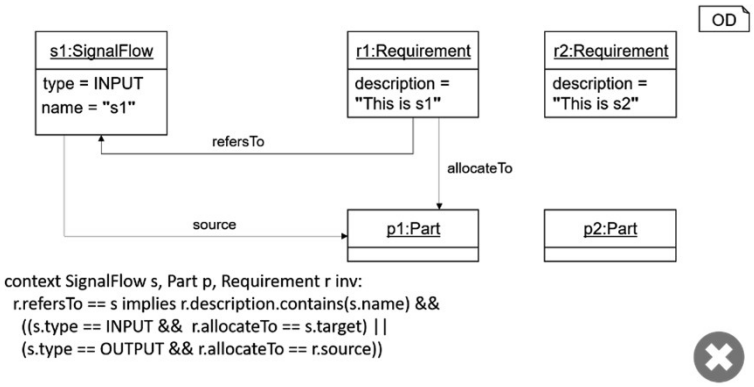

Fig. 17–13: *Example of artifact data invalidating a defined analysis constraint*

An example of modeling analysis using OCL constraints

An example of extracted artifact data invalidating an OCL analysis constraint is given in Figure 17-13. The constraints define that the name in the description of a requirement matches the name of a signal flow the requirement refers to, and that the part allocated to the requirement must be the target of this signal flow. In the artifact data extracted, however, the part *p1* allocated to the requirement *r1* is the source of the referred signal flow *s1*. The execution failure of the analysis is noted in the analysis report and implies required changes for project well-formedness. Changing the part *p1* to be the target of signal flow *s1* instead of its source validates the analysis as shown in Figure 17-14**.** The automated test in both sources checks that the models are consistent in both tools during the whole development process. The check throws an error if an inconsistency occurs, thus notifying developers of potential problems.

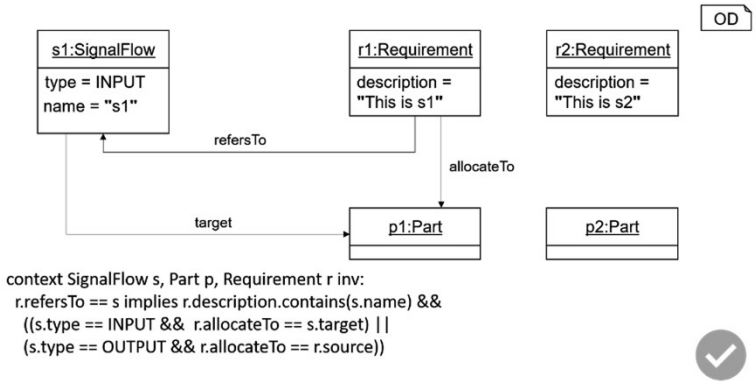

Fig. 17–14: *Example of artifact data validating a defining analysis constraint*

17.5 Conclusion

Employing artifact-based analysis to facilitate model-driven development

Model-driven development aims to reduce the complexity in the development of collaborative embedded systems by reducing the conceptual gap between problem and solution domain. The use of models and MDD tools enables at least a partial automation of the development process. In larger development projects involving several different domains in particular, the huge number of different artifacts and their relationships makes managing them difficult. This can lead to poor maintainability or an inefficient process within the project. The goal of the approach presented is the development of concepts, methods, and tools for artifact-based analysis of model-driven software development projects. Here, the artifact-based analysis describes a reverse engineering methodology that enables repeatable and automated analyses of artifact structures. In this approach, UML/P provides the basis for modeling artifacts and their relationships, as well as specifying analyses. A combination of the UML/P class diagrams and OCL is used to create project-specific artifact models. Additionally, analysis specifications can be defined using OCL while artifact data that represents the current project state is defined using object diagrams, which are instances of the artifact model. This allows the consistency between an AM and its artifact data to be checked. The models are specified in a human-readable form but can also be processed automatically by other MDD tools. The example presented for artifact-based analysis of Enterprise Architect and Doors NG shows the practicability for checking the consistency of artifacts across heterogeneous tools. Here, automated analyses enable system architects to check the conformity of specified components to

requirements and enable requirement engineers to trace the impact of changes on the specified architecture.

17.6 Literature

[Atkinson and Kuhne 2003] C. Atkinson, T. Kuhne: Model-Driven Development: A Metamodeling Foundation. In: IEEE Software, 2003, pp. 36–41.

[Atzori et. al. 2010] L. Atzori, A. Iera, G. Morabito: The Internet of Things: A Survey. In: Computer Networks, 2010, pp. 2787 – 2805.

[Brambilla et. al. 2012] M. Brambilla, J. Cabot, M. Wimmer: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, 2012.

[Butting et. al. 2018] A. Butting, T. Greifenberg, B. Rumpe, A. Wortmann: On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In: Software Technologies: Applications and Foundations, Springer, 2018, pp. 146-153.

[Cheng et. al. 2015] B. H. C. Cheng, B. Combemale, R. B. France, J. Jézéquel, B. Rumpe: On the Globalization of Domain-Specific Languages. In: Globalizing Domain-Specific Languages. LNCS 9400, Springer, 2015, pp 1–6.

[Drave et. al. 2019] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von Wenckstern, A. Wortmann: SMArDT Modeling for Automotive Software Testing. In: R. Buyya, J. Bishop, K. Cooper, R. Jonas, A. Poggi, S. Srirama: Software: Practice and Experience. 49(2), Wiley Online Library, 2019, pp. 301-328.

[Ebert and Favaro 2017] C. Ebert, J. Favaro: Automotive Software. In: IEEE Software, Vol. 34, 2017, pp. 33-39.

[Fernández et. al. 2019] D.M. Fernández, W. Böhm, A. Vogelsang, J. Mund, M. Broy, M. Kuhrmann, T. Weyer, 2019. Artefacts in Software Engineering: A Fundamental Positioning. In: Software & Systems Modeling, 18(5), pp. 2777-2786.

[France and Rumpe 2007] R. France, B. Rumpe: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering (FOSE '07), 2007, pp. 37-54

[Gamma et. al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Greifenberg 2019] T. Greifenberg: Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte. In: Aachener Informatik-Berichte, Software Engineering, Band 42, Shaker Verlag, 2019 (available in German only).

[Greifenberg et. al. 2017] T. Greifenberg, S. Hillemacher, B. Rumpe: Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects. In: Aachener Informatik-Berichte, Software Engineering, Band 30, Shaker Verlag, 2017.

[Jackson 2011] D. Jackson: Software Abstractions: Logic, Language, and Analysis. MIT press, 2011.

[Krcmar et. al. 2014] H. Krcmar, R. Reussner, B. Rumpe: Trusted Cloud Computing. Springer, Switzerland, 2014.

[Lee 2008] Edward A. Lee: Cyber-Physical Systems: Design Challenges. In 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), 2008, pp. 363–369.

[Maoz et. al. 2011] S. Maoz, J. O. Ringert, B. Rumpe: An Operational Semantics for Activity Diagrams using SMV. In: Technical Report. AIB-2011-07, RWTH Aachen University, Aachen, Germany, 2011.

[Müller et. al. 2016] Markus Müller, Klaus Hörmann, Lars Dittmann, Jörg Zimmer: Automotive SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren. 2edition, dpunkt.verlag, 2016 (available in German only).

[OCL 2014] Object Management Group: Object Constraint Language, 2014. http://www.omg.org/spec/OCL/2.4; accessed on 04/30/2020.

[Roth 2017] Alexander Roth: Adaptable Code Generation of Consistent and Customizable Data-Centric Applications with MontiDex. In: Aachener Informatik-Berichte, Software Engineering: Band 31, Shaker Verlag, 2017.

[Rumpe 2016] B. Rumpe: Modeling with UML: Language, Concepts, Methods. Springer International, 2016.

[Rumpe 2017] B. Rumpe: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer International, 2017.

[Schindler 2012] M. Schindler: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. In: Aachener Informatik-Berichte, Software Engineering. Band 11. Shaker Verlag, 2012 (available in German only).

[SysML 2017] Object Management Group. OMG Systems Modeling Language, 2017. http://www.omg.org/spec/SysML/1.5/; accessed on 04/30/2020.

[UML 2015] Object Management Group. Unified Modeling Language (UML), 2015. http://www.omg.org/spec/UML/; accessed on 04/30/2020.