Applied Artifact-Based Analysis for Architecture Consistency Checking

Timo Greifenberg, Steffen Hillemacher, and Katrin Hölldobler

Abstract The usage of models within model-driven software development aims at facilitating complexity management of the system under development and closing the gap between the problem and the solution domain. Utilizing model-driven software development (MDD) tools for agile development can also increase the complexity within a project. The huge number of different artifacts and relations, their different kinds, and the high degree of automation hinder the understanding, maintenance, and evolution within MDD projects. A systematic approach to understand and manage MDD projects with a focus on its artifacts and corresponding relations is necessary to handle the complexity. The artifact-based analysis presented in this paper is such an approach. This paper gives an overview of different contributions of the artifact-based analysis but focuses on a specific kind of analysis: architecture consistency checking of model-driven development projects. By applying this kind of analyses, differences between the desired architecture and the actual architecture of the project at a specific point in time can be revealed.

1 Introduction

The complexity of developing modern software systems or software-intensive systems continues to rise. Software is already considered the most important factor of the competition within the automotive industry, which leads to an increasing complexity especially in automotive software [EF17]. Further examples of complex systems with a high amount of software are cloud-based systems [KRR14], cyberphysical systems (CPS) [Lee08], or Internet of Things (IoT) [AIM10]. When developing complex software systems, there is a wide conceptual gap between the problem and the implementation domains [FR07]. Bridging this gap by extensive handcrafting of implementations can lead to accidental complexities that make

T. Greifenberg (✉) · S. Hillemacher · K. Hölldobler
Software Engineering, RWTH Aachen University, Aachen, Germany
e-mail: greifenberg@se-rwth.de; hillemacher@se-rwth.de; hoelldobler@se-rwth.de

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2019*,
https://doi.org/10.1007/978-3-030-58617-1_5

the development of complex software difficult and costly [FR07]. Additionally, some development processes demand tracing of development artifacts [MHDZ16]. Creating and maintaining such relationships consequently lead to additional effort during the development.

In order to be able to control the complexity of the software as well as to close the conceptual gap between problem and solution domain, approaches of the field of model-driven software development (MDD) arose. MDD technologies have been successfully applied in industrial software development processes, which lead to improved quality and productivity [WHR14]. Platform-independent models [OMG14] provide a means to abstract from technical aspects of the solution domain, making the complexity easier to control. Such models of the problem domain can often be developed directly by domain experts. The usage of domain-specific languages (DSL) for modeling different aspects of the system helps to focus on single aspects of the system in more detail [CCF+15]. The resulting domain-specific models can then be used as primary input for an MDD tool chain. These tool chains can guarantee the conformity of models (both, of single models and between several models), analyze their contents, and automatically create parts of the software product to be developed. Such an automated MDD build process removes the necessity of manual synchronization between models and source code. After adapting the models, the corresponding source code can be automatically recreated without any additional effort. Hence, the use of MDD approaches aims for a more efficient and effective software development process [BCW12], ultimately, reducing manual effort, improving software quality through systematic translation of domain-specific models into the source code, and lowering the development costs.

In MDD projects, not only a large number of different artifacts caused by the complexity of the software exist, but also a multitude of different artifact types. Examples are models, templates, and grammar files, which are required by special technologies to create MDD tools. Between these specific types of artifacts, heterogeneous and complex relationships exist, thus understanding them is vital for MDD tool developers. Examples for such relationships are imports between model artifacts, artifacts that are used as input to automatically generate a set of target artifacts, or artifacts containing each other.

The high amount and complexity of these dependencies create a number of new challenges for MDD projects [GHR17]: (1) poor maintainability in case of necessary changes due to unnecessary or unforeseen artifact relationships, (2) inefficient build processes that perform unnecessary process steps or cannot be executed incrementally, (3) long development times because poorly organized artifact dependencies cause errors, and (4) prevention of reuse of individual tool components or parts of the generated product caused by unnecessary artifact relations. Due to the large number of different artifacts and relationships, the multitude of artifact and relationship types, the use of special MDD tools and the high degree of automation, a systematic approach is necessary to handle the complexity in MDD projects. For the existing projects where no such approach was used, reverse engineering techniques can help to reconstruct how artifacts are

structured. Furthermore, relations between artifacts can be reconstructed, which makes it easier to understand the dependencies as well as provide a way to analyze the potential of process optimization. Such analyses of artifacts and their relations are presented as artifact-based analyses [Gre19].

This paper gives an overview on the artifact-based analysis [Gre19] and, thus, presents content that is already published. It cannot cover all the contributions in all detail, which leads to presenting some of the topics in a simplified way or omitting some details in discussions. Moreover, this work focusses just on a single kind of artifact-based analysis: architecture consistency checking of MDD projects. Compared to the original thesis [Gre19], an additional case of application succeeding the former work is described in this paper in Sect. 4.2.

The main contribution of the original thesis are the methods and concepts for the artifact-based analysis of model-driven software development projects [Gre19]. The contribution consists of:

- A modeling technique for modeling artifact relationships and for the specification of associated analyses.
- A concrete model, which describes artifact relationships in MontiCore-based development projects.
- A methodology for using the modeling technique.
- A tool chain that supports artifact-based analyses.

2 Foundations

In this section, the modeling languages and model-processing tools used in this approach are presented. Using these to describe artifacts and artifact relationships is explained in Sect. 3.

2.1 UML/P

The UML/P language family [Rum16, Rum17] is a language profile of the Unified Modeling Language (UML) [OMG15], which is a modeling standard developed by the OMG [OMG17]. Due to the large number of languages, their fields of application, and the lack of formalization, the UML is not directly suitable for model-driven software development. However, this could be achieved by restricting the modeling languages and language constructs allowed as done in the UML/P language family. A textual version of the UML/P, which can be used in MDD projects, was developed [Sch12]. The approach for the artifact-based analysis of MDD projects uses the languages Class Diagram (CD), Object Diagram (OD), and the Object Constraint Language (OCL).

2.1.1 Class Diagrams

Class diagrams serve to represent the structure of software systems. They form the central element for modeling software systems with the UML and are therefore the most important notation. Using class diagrams primarily, classes and their relationships are modeled. In addition, enums and interfaces, associated properties such as attributes, modifiers, and method signatures as well as various types of relationships and their cardinalities can be modeled.

In addition to being used to represent the technical, structural view of a software system, i.e., as the description of source code structures, class diagrams can be used in analysis to structure concepts in the real world [Rum16]. Especially for this use case, an even more restrictive variant of the UML/P Class Diagrams was developed: the language Class Diagram for Analysis (CD4A) [Rot17]. As a part of this approach, CD4A is used to model structures in model-based development projects.

2.1.2 Object Diagrams

Object diagrams are suitable for the specification of exemplary data of a software system. They describe a state of the system at a concrete point in time. ODs can conform to the structure of an associated class diagram. A check, whether an object diagram corresponds to the predefined structure of a class diagram is in general not trivial. For this reason, an approach for an Alloy [Jac11]-based verification technique was developed [MRR11].

In object diagrams, objects and the links between objects are modeled. The modeling of the object state is done by specifying attributes and assigned values. Depending on the intended use, object diagrams can describe a required, represent prohibited or existing situation of the software system. In addition to the concepts described in the original thesis [Sch12], the current version of the UML/P OD language allows the definition of hierarchically nested objects. This has the advantage that hierarchical relationships can also be displayed as such in the object diagrams. Listing 1 shows an example of a hierarchical OD with two nested objects. In this work, CDs are not used to describe the classes of an implementation, but 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 this approach, object diagrams are used to describe analysis data, i.e., they reflect the current state of the project at the conceptual level.

OD

```
objectdiagram Students {

  Max:Person {
    id = 1;
    firstName = "Max";
    name = "Mustermann";
    address = address1:Address {
      street = "Ahornstraße";
      city = "Aachen";
      houseNumber = 55;
      country = "Germany";
    };
   };
}
```

Listing 1 Example OD with two hierarchically nested objects

2.1.3 Object Constraint Language

The OCL is a specification language of the UML, which allows to model additional conditions of other UML languages. For example, the 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. This approach uses the OCL/P variant only. OCL is only used in conjunction with class diagrams throughout this approach. OCL expressions are modeled within class diagram artifacts.

2.2 MontiCore

MontiCore [GKR+06, KRV08, GKR+08, KRV10, Kra10, Völ11, HR17] is a Language Workbench for the efficient development of compositional modeling languages. Figure 1 gives an overview of the structure and workflow of MontiCore-based generators.

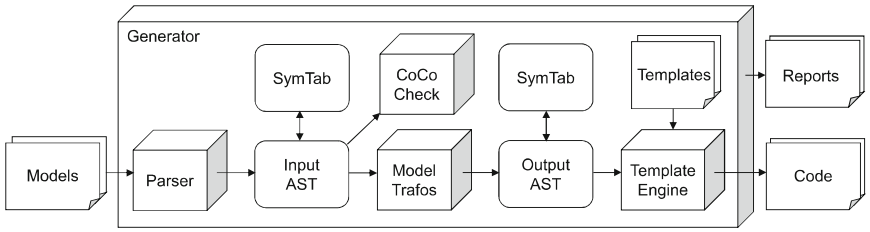

Fig. 1 Structure and execution of MontiCore-based generators

MontiCore languages are primarily specified using context-free grammars. MontiCore uses a grammar file as input to generate a parser, a data structure for abstract syntax trees (AST), and an implementation of the visitor design pattern [GHJV95] for the AST data structure. The AST data structure can be extended by a symbol table data structure that can be partially generated. Symbol tables act as a model interface and offer a way to dynamically load other model artifacts. Thus, symbol tables provide an integrated processing of multiple models of different modeling languages. By creating well-formedness rules, the so-called context conditions (CoCo), additional conditions can be added to the rules given by the grammar. These conditions are checked on the coherent data structure consisting of AST and symbol table.

Prior to the generation, model-to-model (M2M) transformation can be used to prepare AST and symbol table. These can either be implemented in the form of Java code or defined using special transformation languages [Wei12, Höl18]. A methodology for creating new transformation languages is also available [HRW15].

Next, the source code of the system in development can be generated by template-based code generation. As a by-product, additional reports are generated, which include relevant events of the last generation run or summarize the generation process.

MontiCore, thus, generates an infrastructure for processing models, checking them according to their well-formedness rules, transforming them using M2M transformations, as well as for generating source code artifacts. Moreover, MontiCore offers a runtime environment (RTE) providing functions and data structures which can be used by both the generated and handwritten parts of MontiCore-based tools. MontiCore has already been used to create languages and related tools in various domains including Automotive [RSW+15], robotics [AHRW17], and cloud applications [NPR13].

2.3 Architecture Consistency Checking

Architectural reconstruction and architectural consistency checking (ACC) are techniques that are used in the area of software architecture. Their goal is to compare the current state of a software architecture (the actual architecture or descriptive architecture) with the planned architecture (also called target architecture or prescriptive architecture). The following definitions of basic terms are taken from [TMD09]. The term architecture degradation is defined as

> The resulting discrepancy between a system's prescriptive and descriptive architecture.

Further, the architectural drift is defined as

> The introduction of principal design decisions into a system's descriptive architecture that (a) are not included in, encompassed by, or implied by the prescriptive architecture, but which do not violate any of the prescriptive architecture's design decisions.

Another term is the architectural erosion defined as

> The introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture.

In this work, the result of the calculation of architectural degradation is called difference architecture. Such a difference architecture, thus, contains all differences between the target and the actual architecture. Furthermore, it combines the parts of the architectural drift with those of architectural erosion. Both, the absence of expected relationships considered as architectural drift and unindended relationships considered as architectural erosion, are examined as part of this work.

In order to retrieve the difference architecture, the actual architecture must be compared to the target architecture. For this comparison, the target architecture must be specified manually, while the actual architecture is reconstructed (semi-) automatically from the source code. This process is called architectural reconstruction.

The process of architectural reconstruction distinguishes between top-down and bottom-up approaches. Bottom-up approaches are also referred to as architecture recovery [DP09]. They work fully automatically and can therefore be used even if no target architecture is available. However, they suffer from a lack of accuracy [GIM13]. By combining two different bottom-up approaches, the quality of the actual architecture during architecture recovery could be increased, which also increased the quality of the reconstructed architecture [vDB11, PvDB12]. In contrast to bottom-up approaches, top-down approaches rely on the existence of a modeled target architecture. Top-down approaches are also known as architecture discovery [DP09].

The calculation of the difference architecture based on a target architecture and the source code is also known as architecture consistency checking [PKB13]. In a study [ABO+17], different ACC approaches from research were compared. The study classified the approaches according to the following characteristics:

- The type of the underlying extraction method (static, dynamic). All approaches of architectural reconstruction use extraction techniques to reveal the relationships between the atomic architectural units (e.g., types, files). Moreover, there is an approach that combines both extraction methods [SSC96]. This approach utilizes information of a dynamic extraction to determine the frequency and, thus, relevance of architecture violations, which are detected based on static extraction.
- The technique for the evaluation of the architectural consistency. Particularly, interesting techniques in this area are:
  - Reflection Modeling (RM) [MNS01], a technique in which source code artifacts are assigned to architectural units of the target architecture. The

relationships between modules (actual architecture) are then reconstructed by investigating the relationships of the source code.
– DSL-based approaches [HZ12, GMR15] allow a comfortable definition of the target architecture. In addition to the specification of architectural units and expected relationships, information such as complex conditions for architectural violations or the severity of architectural violations in the event of an occurrence can be defined. DSL-based approaches can be combined with RM-based approaches.

These approaches can also be used for architectures outside the target product. For this purpose, the scope of the relationships under consideration must be extended. In MDD projects, the architecture of the target product as well as the architecture of individual MDD tools, the interaction between tools and artifacts, and the composition of tools into tool chains are of interest for such analyses. In this work, a flexible, RM-based approach for checking architecture consistency for MDD projects is presented. Depending on the application area, static, dynamic, or static and dynamic (in combination) extracted relationship types are considered.

3 Artifact-Based Analysis

This section gives an overview over the developed solution concept to perform artifact-based analyses. Before presenting details, the basic definition of an artifact is given.

Definition 1 An artifact is an individually storable unit with a unique name that serves a specific purpose in the context of the software engineering process.

In this definition, the focus is on the physical manifestation of the artifact rather than on the role in the development process. This requires that an artifact can be stored as an individual unit and then be referenced. Nevertheless, an artifact should serve a defined purpose during the development process because the creation and maintenance of the artifact would otherwise be unnecessary. However, no restrictions are made about how the artifact is integrated into the development process, i.e., an artifact does not necessarily have to describe the target system, for example, in the form of a model, but can also be part of the MDD tool chain instead. The logical content of artifacts thus remains largely unnoticed. This level of abstraction was chosen to effectively analyze the artifact structure, while considering the existing heterogeneous relationships and thus meeting the focus of this work.

An important part of the overall approach is to identify the artifacts, tools, systems, etc., and their relationships present in the system under development. Here, modeling techniques are used, which allow to make these concepts explicit and, thus, enable model-based analyses. Figure 2 gives an overview of the model-based solution concept.

First, it is explicitly defined which types of artifacts, tools, other elements, and relationships in the MDD project under development are of interest. This task will be covered by the architect of the entire MDD project through modeling an artifact model (AM). Such a model structures the corresponding MDD project. As the AM defines the types of elements and relationships and not the concrete instances, this model can stay unchanged over the lifecycle of the entire project unless new artifact or relationship types are added or omitted.

Definition 2 The artifact model defines the relevant artifact types as well as associated relationship types of the MDD project to be examined.

Artifact data reflect the current project state. They can ideally be automatically extracted and stored in one or more artifacts.

Definition 3 Artifact data contain information about the relevant artifacts and their relationships, which are present at a certain point in time in the MDD project. They are defined in conformance to an artifact model.

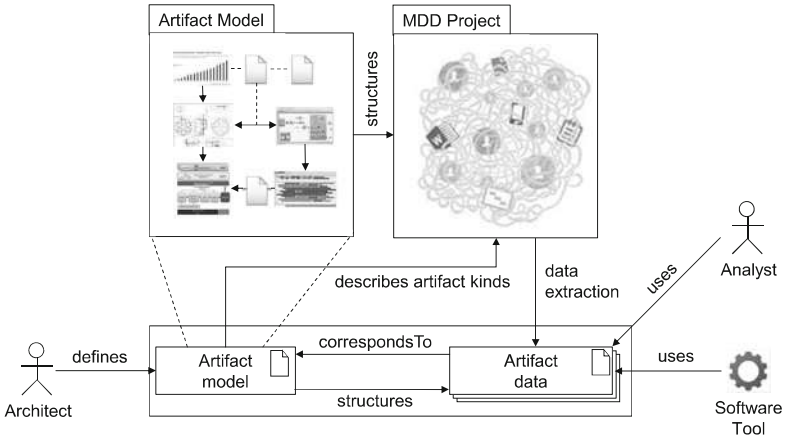

Fig. 2 Solution concept for artifact-based analyses

Artifact data are in an ontological instance relationship [AK03] to the AM, whereby each element and each relationship from the artifact data confirm to an element or relationship type of the AM. Thus, the AM determines the structure of the artifact data. Figure 3 shows how this is reflected in the used modeling technique.

Artifact data represent the project state at a certain point in time. Analysts or special analysis tools can use the extracted artifact data to get an overview of the project state, to check certain relations, create reports, and reveal optimization potential in the project. The overall goal is to make the model-driven development process as efficient as possible.

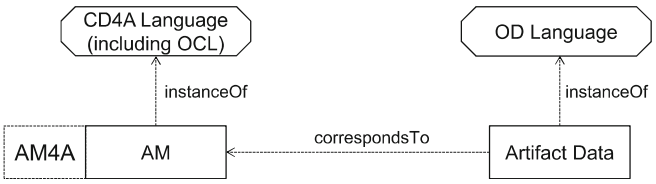

Fig. 3 Overview about the languages and models used for artifact modeling

Beside other kinds of analyses [Gre19], this approach is suited to define and execute architecture consistency checking of model-driven software development projects taking into account input models, MDD tools, which themselves consist of artifacts, and handwritten or generated artifacts that belong to the target product of the development process. A corresponding AM depends on the languages, tools, and technologies used in the project under development. Thus, it must usually be tailored specifically for a given project. However, such a model can be reused as a whole or partially for similar projects.

Fig. 4 Required steps to perform an artifact-based analysis

As shown in Fig. 4, the first step before the execution of artifact-based analysis is to create a project-specific AM. Subsequently, artifact data analyses are specified based on this AM. Based on these two preceding steps, the artifact-based analysis can finally be executed.

3.1 Create Artifact Model

The first step of the methodology is to model an AM. The AM determines the scope for project-specific analyses, explicitly defines the relations between the artifacts, and specifies pre-conditions for the analyses. Furthermore, by using the languages CD and OCL (see Sect. 2.1), it is possible to reuse the existing MDD tools to perform the analyses. An AM core as well as an extensive AM for MontiCore-based MDD projects has already been presented [Gre19]. If a new AM needs to be created or an existing AM needs to be adapted, the AM core and possibly parts of the existing project-specific AMs should be reused. A methodology for this already exists [Gre19].

The central elements of any AM are the artifacts. All project-specific files and folders are considered artifacts. Artifacts can contain each other. Typical examples

of artifacts that contain other artifacts are archives or folders of the file system, but database files are also possible. In this paper, we focus on those parts of the AM that are used to perform ACC. Figure 5 shows the relevant part of the AM core.

Since the composite pattern [GHJV95] is used for this part of the AM core, the archives and folders contain each other in any order. Each artifact contained in one artifact container at most. If all available artifacts are modeled, there is exactly one artifact that is not contained in a container: the root directory of the file system. Furthermore, artifacts can contribute to the creation of other artifacts (`produces` relationship), and they can statically reference other artifacts (`refersTo` relationship). These artifact relations are defined as follows:

Definition 4 If an artifact needs information from another artifact to fulfill its purpose, then it refers to the other artifact.

Definition 5 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.

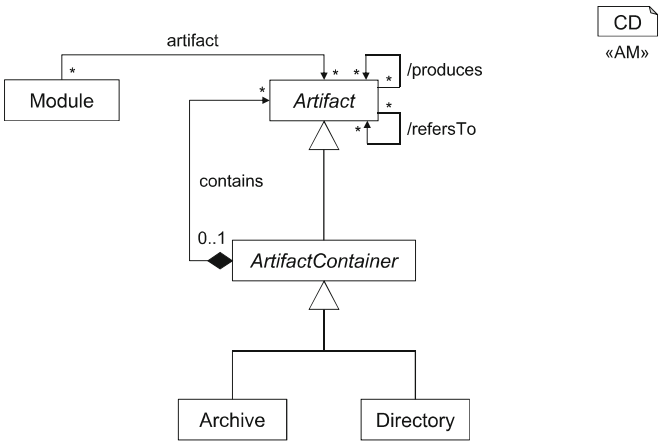

Fig. 5 Part of the AM core relevant for architecture consistency checking

Both relations are defined as derived association in the AM. Thus, these relations must be further specified in project-specific AMs (see Sect. 4), while the definition of artifact data analyses can already be done based on the derived associations (see Sect. 3.4). The specialization of associations is defined using OCL constraints [Gre19], because the CD language does not offer a first class concept here.

Modules represent architectural units based on artifacts and their relationships. The artifacts relation between modules and artifacts must be explicitly defined by the architect. A top-down approach (see Sect. 2.3) is used for the architecture reconstruction part of the analysis.

3.2 Specify Artifact Data Analyses

The second step of the methodology is the specification of project-specific analyses based on the previously created AM. Artifact-based analyses should be repeatable and automated. For this reason, an implementation of the analyses is necessary. This implementation can be either direct in which case the person performing the analysis would have both roles, analyst and analysis tool developer, or the analyst specifies the analyses as requirements for the analysis tool developer, who can then implement a corresponding analysis tool. In this work, analyses are specified using OCL. The reasons for this are:

1. The use of the CD language in the specification of the AM makes it possible to specify analyses using OCL, since the two languages can be used well in combination.
2. The OCL is already used to define project-specific AMs. The reuse of familiar languages reduces the learning curve for analysts, makes analysis specifications readable for architects, and enables reuse in tool development.
3. The OCL has mathematically sound semantics, allowing analyses to be described precisely. OCL expressions are suitable as input for a generator that can automatically convert them into MDD tools. Thus, this tool implementation step can be automated, reducing the effort for the analysis tool developer.

As this paper focuses on ACC, Sect. 2.3 defines architecture consistency checking as artifact data analysis in detail. Other kinds of artifact-based analyses are presented in the original thesis [Gre19].

3.3 Artifact-Based Analyses

As third step of Fig. 4, the artifact-based analysis is executed. This step is divided into five sub-steps, which are supported by automated and reusable tooling. Figure 6 shows these steps and their corresponding tools.

When performing artifact-based analyses, the first step is the extraction of relevant project data. If the data are stored in different files, a merge of the data takes place. The entire data set is then checked for conformity to the AM. In the next step, the data are accumulated based on the specification of the AM, so that the derived properties are present when performing the last step, the artifact data analysis. To support the steps of performing analyses, the MontiCore Artifact Toolchain (MontiArT) was developed [Gre19]. MontiArT is a tool chain that can be used to collect, merge, validate, accumulate, and finally analyze artifact data. Thus, all sub-steps of the artifact-based analysis are supported. The individual steps are each performed by one or more small tools, which can be combined in a tool chain, for example, by using a script. The tools shown in Fig. 6 are arranged according to the tool chain's execution order. The architecture as tool chain is modular and

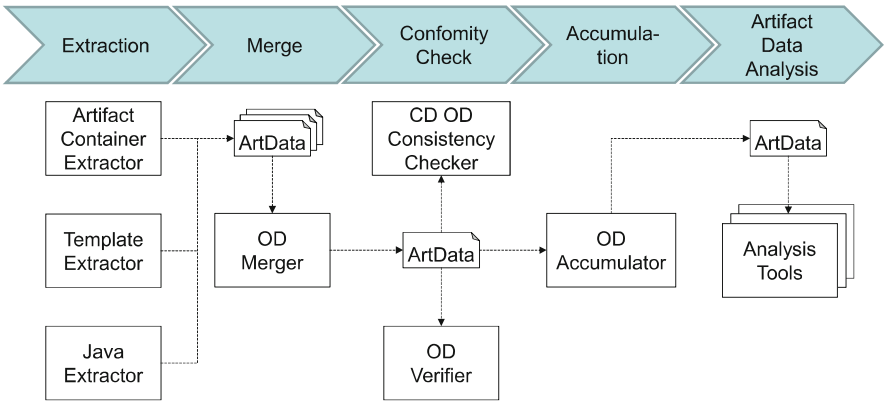

Fig. 6 Sub-step of the artifact-based analysis

adaptable. New tools can be added without the need to adapt other tools. Existing tools can be adapted or removed from the tool chain without having to adapt other tools. This is advantageous, since this work is based on the assumption that there is no universal AM for all MDD projects [Gre19]. For this reason, when using the tool chain in a new project, it is usually necessary to make project-specific adjustments. The selected architecture supports the reuse and adjustments of individual tools.

3.4 Artifact-Based Analysis for Architecture Consistency Checking

In architecture reconstruction (see Sect. 2.3), the relationships between the architectural units of the system are reconstructed, which are represented here by modules (see Sect. 3.3). Modules can be defined for the models and tools used in the project as well as for the product to be developed. By explicitly modeling modules and assigning artifacts of the project to modules, an abstract project view can be created that gives stakeholders (architects, developers, or tool developers) a good overview of the MDD project. Modules can often be identified by the existing packages or folders. However, due to architectural erosion (see Sect. 2.3), this alignment can be lost. It is also possible to define different architectures for a single project, each with a different focus [Lil16]. Furthermore, different types of modules can generally be defined for predefined elements of a software architecture, such as components, layers, or interfaces. In general, it is recommended that the specified modules are reflected as well as possible by an appropriate structure in the project. The relationships between modules can be derived based on the assigned artifacts as shown in Listing 2.

```
1  association /containedArtifact                                    CD
2    [*] Module -> Artifact [*];                                     «AM4A»
3
4  context Module m, ArtfiactContainer c inv:
5    m.containedArtifact.containsAll(m.artifact) &&
6    m.artifact.contains(c) implies m.artifact.containsAll(c.
       contains**);
7
8  association /externalReferredArtifact
9    [*] Module -> Artifact [*];
10 association /externalProducedArtifact
11   [*] Module -> Artifact [*];
12
13 context Module inv: (*@ \label{lst:applications_module:
       reliesonartifacts} @*)
14   externalReferredArtifact ==
15     containedArtifact.refersTo.removeAll(containedArtifact);
16
17 context Module inv: (*@ \label{lst:applications_module:
       producedartifact} @*)
18   externalProducedArtifact ==
19     containedArtifact.produces.removeAll(containedArtifact);
20
21 association /reliesOnModule [*] Module -> Module [*];
22 association /contributionModule [*] Module -> Module [*];
23
24 context Module inv: (*@ \label{lst:applications_module:
       reliesonmodules} @*)
25   reliesOnModule == externalReferredArtifact.module;
26
27 context Module inv: (*@ \label{lst:applications_module:
       contributionmodule} @*)
28   contributionModule == externalProducedArtifact.module;
```

Listing 2 Specification of the artifact data analysis for architecture consistency checking: actual architecture

A distinction is made between two different types of module relationships, which can be calculated based on the assigned artifacts and their relationships. The `reliesOnModule` relationship indicates that a module refers to another module, while the `contributionModule` relationship indicates the participation of a module in the creation of another module similar to the corresponding relations for artifacts defined in Sect. 3.1. To calculate the relationships, further derived auxiliary associations were defined. Each module can contain a set of artifacts, which is defined as the set of artifacts directly assigned to the module together with all artifacts transitively contained by assigned artifacts. Thus, when assigning a folder to a module, contained artifacts are also regarded to be part of that module. Furthermore, for each module, the external artifacts can be calculated, which are those artifacts that are related to artifacts of the module, but are not assigned to

the module themselves. Depending on the type of the underlying artifact relationship, the relationship is represented by the `externalReferredArtifact` or the `externalProducedArtifact` relationship in the shown AM extension. With the help of these associations, the calculations of `reliesOnModule` and `contributionModule` are specified, which results in the actual architecture of the system under development.

Based on the actual architecture, the difference architecture between the target and the actual architecture can also be calculated with the help of an AM extension. The first required extension part for this analysis is shown in Listing 3. It allows the specification of the target architecture.

```
29  association intendedReference [*] Module -> Module [*];
30  association intendedContribution
31    [*] Module -> Module [*];
```

CD «AM4A»

Listing 3 Specification of the artifact data analysis for architecture consistency checking: target architecture

With the help of the two associations `intendedReference` and `intendedContribution`, the intended relationships between modules can be specified. These can then be used to calculate the differences between the target and actual architecture. The second extension part for this analysis is shown in Listing 4. It allows the representation of the difference architecture and specifies its calculation.

```
32  association /unintendedReference
33    [*] Module -> Module [*];
34  association /unintendedContribution
35    [*] Module -> Module [*];
36
37  association /missingReference [*] Module -> Module [*];
38  association /missingContribution [*] Module -> Module [*];
39
40  context Module inv:
41    unintendedReference ==
42      reliesOnModule.removeAll(intendedReference);
43
44  context Module inv:
45    unintendedContribution ==
46      contributionModule.removeAll(intendedContribution);
47
48  context Module inv:
49    missingReference ==
50      intendedReference.removeAll(reliesOnModule);
51
52  context Module inv:
53    missingContribution ==
54      intendedContribution.removeAll(contributionModule);
```

CD «AM4A»

Listing 4 Specification of the artifact data analysis for architecture consistency checking: difference architecture

To be able to capture the results of the calculations, four derived associations were introduced. For each type of relationship to be compared, two associations are specified here. One represents the unexpected, additional relationships of the actual architecture in comparison to the target architecture, and the other contains the missing, but intended relationships. The unintended, additional relationships can be calculated by removing the intended relationships from the set of existing relationships. In contrast to this, missing, intended relationships can be calculated by removing the existing relationships from the set of intended relationships.

4 Applied Analyses

This section describes two different applied artifact-based analyses for architecture consistency checking. For both analyses, only step one (*Create Artifact Model*) and step three (*Artifact-based Analysis*) are described. Step two (*Specify Artifact Data Analyses*) was already presented in Sect. 3.4. As the defined analysis works completely on the concepts of the presented AM core, there is no need to define different analyses for the two different applications.

4.1 DEx Generator

The MontiCore Data Explorer (DEx) [Rot17] is a generator based on MontiCore that transforms an input CD into an executable system for data exploration. In addition to the generation of data classes, a graphical user interface and a persistence layer to connect the application to a database can be generated.

DEx offers a mechanism for the integration of handwritten code and a mechanism for the exchange and addition of templates by the generator user making the generator very flexible and the generation process highly configurable. Furthermore, internal M2M transformations take place when the generator is executed. The configuration functionality and the execution of internal model transformations lead to the fact that generation processes in different DEx-based development projects can differ greatly. For this reason, an analysis of DEx-based projects is particularly interesting.

DEx is delivered together with an exemplary CD model for generating an application for a social network. A target architecture was created together with the architect of DEx using the exemplary CD model as input for the generator. The architecture consists of eleven modules, which can be part of the generator, the generated product, or the runtime environment. As preparation for the analysis, the project-specific AM shown in Figs. 7 and 8 was modeled.

Figure 7 presents the project-specific `refersTo` relations of the DEx generator. The reflexive `reliesOnJavaArtifact` association indicates that two Java artifacts are dependent on each other. One Java source code file is dependent on

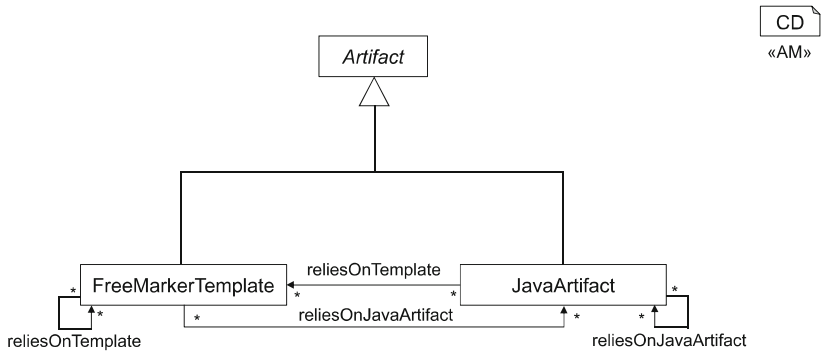

Fig. 7 Specific `refersTo` relationships between template and Java artifacts in the DEx project

another Java artifact iff the other artifact must be loaded when compiling the source code file. A .class file depends on another artifact iff something (type, method, constant, ...) from the other file is used when executing the part of the program defined by the .class file.

Templates can call each other during their execution [Sch12], which means that templates are dependent on other templates. This relationship is modeled by the `reliesOnTemplate` relationship of the AM. Furthermore, it is possible to create Java objects from templates and to store them temporarily, provide them as parameters to other templates, and call their methods. Hence, templates can depend on the existence of the corresponding Java artifacts. This relationship is represented by the `reliesOnJavaArtifact` relationship. Java artifacts can rely on the existence of templates when replacing standard templates with project-specific templates [Rot17].

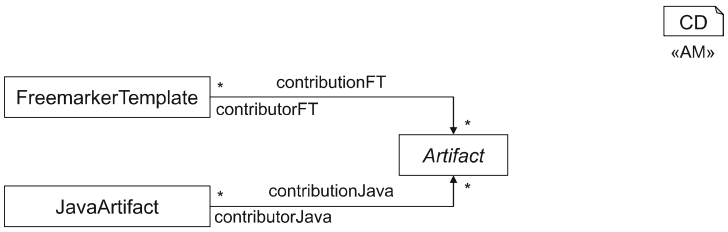

Fig. 8 Template and Java artifacts contribute to the creation of artifacts

The relationships `contributionFT` and `contributionJava` are specializations of the produces relation. Whenever a Java or template artifact of the generator contributes to the generation of a target file, such a relation is recorded in the artifact data.

To extract the data of the shown AM, several reusable as well as project-specific extractors were used. In the following step, the artifact data of each extractor are merged into a single artifact data file, which acts as the first input for the calculation of the actual architecture. The second input for the calculation of the actual architecture is the module definition, i.e., the assignment of artifacts to the modules of the target architecture defined by the project's architect. Based on the calculated actual architecture and the modeled target architecture, the difference architecture of the DEx project was determined, which is shown in Fig. 9.

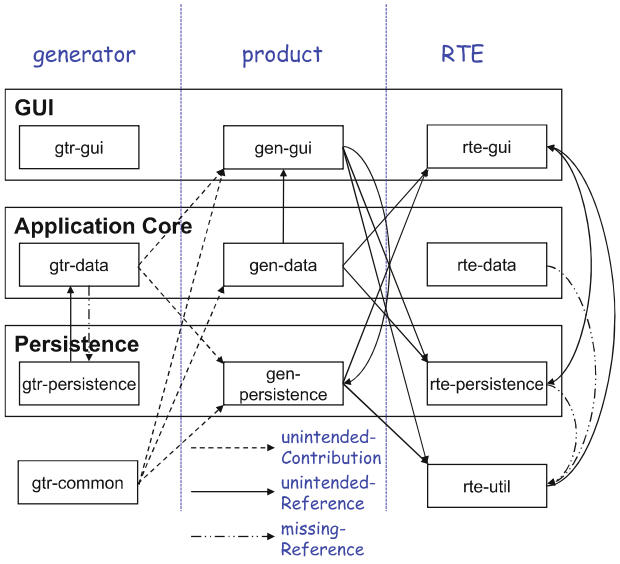

Fig. 9 Result of the artifact-based analysis: The difference architecture of the DEx project

The number of relationships present in the difference architecture reveals that a major architectural degradation has occurred. An architectural degradation leads to the fact that per time unit less and less functionality can be implemented and bugs can be fixed, which ultimately leads to frustrated and demotivated developers [Lil16]. Based on the analysis performed, the revealed architecture degradation can be eliminated, thus counteracting these problems. For this purpose, it must be decided individually for each relationship whether it is a violation of the target architecture or whether the target architecture is outdated at this point. The cause for unwanted relationships of the difference architecture can be traced using reporting files created as by-product by the tooling. To update an outdated target architecture, missing required relationships must be added and required relationships that have become obsolete must be removed. In the case that the target architecture is still valid, the corresponding relationship indicates a problem in the source code of the project. When remedying such a problem, missing and additional, unintended relationships of the actual architecture must be treated differently. Unintended

Table 1 Examined artifacts of the DEx project

Module	JavaSourceFile	FreeMarkerTemplate	∑
gtr-gui	19	96	115
gtr-data	27	92	119
gtr-persistence	14	93	107
gtr-commons	17	5	22
gen-gui	121	0	121
gen-data	44	0	44
gen-persistence	121	0	121
rte-gui	119	0	119
rte-data	10	0	10
rte-persistence	17	0	17
rte-util	8	0	8
Nicht zugewiesen	38	15	53
∑	555	301	856

Table 2 Examined relations between artifacts of the DEx project

Relation	Source artifact	Amount
contains	Directory	940
imports	JavaSourceFile	1815
reliesOnJava	JavaSourceFile	3807
reliesOnTemplate	JavaSourceFile	324
reliesOnJava	FreeMarkerTemplate	88
reliesOnTemplate	FreeMarkerTemplate	59
contributionJava	JavaSourceFile	270
contributionFT	FreeMarkerTemplate	2760
∑		10,063

relationships must be eliminated by refactoring the source code in any case, while a missing relationships can be either also fixed by a refactoring or it indicates that the desired, corresponding functionality was not implemented at all or in a fundamentally different way.

Finally, Tables 1 and 2 summarize some numbers to give an impression of the scope of the project and the regarded artifact and relationship types.

4.2 MontiCar Repositories

While the artifact-based analysis of DEx focuses on a single development project, we also applied our methodology to multiple projects located in different repositories. Although these projects reside in different locations, i.e., version control repositories, they are still related to each other. More specifically, our goal was to perform architectural compliance checking for the MontiCar family, a family

of projects developed in the context of autonomous driving and other automotive related topics [KRRvW17, KRSvW18, KNP+19].

In case of multiple projects, we change our approach in such a way that we define modules for the architectural compliance check based on the projects rather than single parts of each project. For example, the project containing common functionality for all other projects forms a single module or all functionality used for visualization purposes is comprised in a single module. Next, the target architecture for the set of projects is defined. Listing 5 shows an excerpt of this definition.

The *EmbeddedMontiArc* module has an intended relation to the commons and visualization modules as well as to the *EmbeddedMontiArcView* module. For the analyses of the MontiCar family, in general, we used the same specifications as defined for DEx in Listings 2, 3, and 4. Furthermore, we assigned the *src* folder of each project to the respective module since each project is a Maven project. The transitive closure of the *src* folder defined the set of artifacts considered for the ACC as described in Listing 2, ll. 3-5.

For the analysis of MontiCar, we only considered intended references between the modules, since contribute relations only exist within each project. The primary goals were to first check if all intended relations between modules actually exist and second to get an overview of all unintended relations within the family. With the help of the respective results, we then improved the current architecture of the family to adhere to the target architecture.

```
1  objectdiagram ModuleRelations {                                    OD
2                                                                     «MD»
3    EmbeddedMontiArc:Module{};
4    EmbeddedMontiView:Module{};
5    languagescommon:Module{};
6    visualisation:Module{};
7
8
9    link intendedReference EmbeddedMontiArc -> EmbeddedMontiView;
10   link intendedReference EmbeddedMontiArc -> languagescommon;
11   link intendedReference EmbeddedMontiArc -> visualisation;
12 }
```

Listing 5 Excerpt of the module definition for the MontiCar project family

Table 3 shows the results of the ACC for the MontiCar family. The first column displays the defined modules, and the following columns contain the total number of artifacts analyzed, the number of intended references for each module defined in the target architecture, and the found numbers of unintended and missing references as part of the difference architecture. Looking at the results, it is interesting to see that for most of the modules either a unintended or missing reference was discovered. Moreover, the analysis found unintended references for more than half of the modules. Unintended references are references not contained in the target

Table 3 Results of analysis containing the number of artifacts of each module, the number of intended references of the target architecture, the number of unintended references, and the number of missing references

Module	# Artifacts	# Intended	# Unintended	# Missing
EmbeddedMontiArc	151	6	0	1
Enum	110	0	3	0
languagescommon	136	0	3	0
EmbeddedMontiArcMath	261	4	2	0
MontiMath	119	5	1	1
TaggingExamples	31	2	2	0
Struct	51	4	0	0
EmbeddedMontiView	120	4	2	2
EMAM2Cpp	357	9	1	0
ViewVerification	181	5	2	0
reporting	510	4	1	1
Tagging	121	1	0	0
NumberUnit	26	1	0	0
visualisation	188	6	3	3
$\sum$	2362	51	20	8

architecture, however, present in the actual architecture. In general, these deviations illustrate the architectural erosion within the MontiCar family. Depending on the module, one deviation is considered more critical than others. For example, for the *languagescommon* project, the number of intended references is zero, since it contains common functionality, which must not relate to any other project. However, three unintended references to other projects of the family were found. The impact of other occurrences in the difference architecture is less critical, since these were classified as architectural drift. Especially, missing references between modules fall into this category for the MontiCar project. As a consequence, the target architecture was adjusted in these cases accordingly.

Using the results of the artifact-based analysis, we were able to look at each of the references found in the current state of the MontiCar family in more detail. Besides currently existing references between modules, the analysis also provided a detailed list of the artifacts and their relations that cause a reference between modules. For each of the unintended module references not contained in the target architecture, we had the options to reduce the architectural drift:

- Declare the reference to be unintended and as a consequence refactor the artifacts causing the reference.
- Declare the reference to be intended and add it to the target architecture.

Finally, we were able to reduce the architectural drift within the family of projects using the artifact-based architectural consistency checking and continue to perform it regularly to keep the architectural drift as small as possible at all times.

5 Conclusion

MDD helps to master the complexity of software development and to reduce the conceptual gap between the problem and the solution domain. By using models and MDD tools, the software development process can be at least partially automated. Especially when MDD techniques are used in large development projects, it can be difficult to manage the huge number of different artifacts, different artifact kinds, and MDD tools. This can lead to poor maintainability or an inefficient build process. The goal of the presented approach is the development of concepts, methods, and tools for artifact-based analysis of model-driven software development projects. The term artifact-based analysis is used to describe a reverse engineering methodology, which enables repeatable and automated analyses of artifact structures. To describe artifacts and their relationships, artifact and relationship types, and for the specification of analyses, a UML/P-based modeling technique was developed. This enables the specification of project-specific artifact models via CD and OCL parts. As part of the modeling technique, a reusable AM core is defined. In addition, analysis specifications can also be defined by CDs and OCL, while artifact data that represent the current project situation are defined by ODs. The choice of modeling technique allows you to check the consistency between an AM and artifact data. The models are specified in a human readable form but can also be automatically processed by MDD tools. The artifact-based analysis consists of five sub-steps, which, starting from an MDD project and an associated AM, allows the execution of specified analyses.

While this paper gave an overview of the contributions of the original thesis [Gre19], not all contributions could be covered in detail. Especially, this paper focused on only a single kind of artifact-based analysis, architecture consistency checking, whereas the original thesis presents several different analysis kinds. Moreover, Sect. 4.2 of this work described an additional case of application succeeding the former work.

Due to the results obtained, this work contributes to handling the increasing complexity of large MDD projects by explicit modeling artifact and relationship types, which can be used for manual and automated analysis of MDD projects. Hidden relationships can be revealed and checked immediately, which opens up the possibility for corrections and optimizations in a given project.

References

ABO+17. Nour Ali, Sean Baker, Ross O'Crowley, Sebastian Herold, and Jim Buckley. Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, pages 1–35, 2017.

AHRW17. Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

AIM10. Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54:2787–2805, 2010.

AK03. C. Atkinson and T. Kuhne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20:36–41, 2003.

BCW12. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

CCF+15. Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe. On the Globalization of Domain Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 1–6. Springer, 2015.

DP09. Stephane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35:573–591, 2009.

EF17. C. Ebert and J. Favaro. Automotive Software. *IEEE Software*, 34:33–39, 2017.

FR07. Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, pages 37–54, May 2007.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

GHR17. Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

GIM13. Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A Comparative Analysis of Software Architecture Recovery Techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 486–496. IEEE Press, 2013.

GKR+06. Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

GKR+08. Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10–18, 2008, Companion Volume*, pages 925–926, 2008.

GMR15. Timo Greifenberg, Klaus Müller, and Bernhard Rumpe. Architectural Consistency Checking in Plugin-Based Software Systems. In *European Conference on Software Architecture Workshops (ECSAW'15)*, pages 58:1–58:7. ACM, 2015.

Gre19. Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte*. Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019.

Höl18. Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

HR17. Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

HRW15. Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

HZ12. Thomas Haitzer and Uwe Zdun. DSL-based Support for Semi-automated Architectural Component Model Abstraction Throughout the Software Lifecycle. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '12. ACM, 2012.

Jac11. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2011.

KNP+19. Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

Kra10. Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

KRR14. Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

KRRvW17. Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

KRSvW18. Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language Embedded-MontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447–457. ACM, October 2018.

KRV08. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

KRV10. Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

Lee08. Edward A. Lee. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

Lil16. Carola Lilienthal. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. dpunkt, 2016.

MHDZ16. Markus Müller, Klaus Hörmann, Lars Dittmann, and Jörg Zimmer. *Automotive SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren*. dpunkt.verlag, 2 edition, 2016.

MNS01. G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27:364–380, 2001.

MRR11. Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

NPR13. Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, volume 1118 of *CEUR Workshop Proceedings*, pages 15–24, 2013.

OMG14. Object Management Group. Object Constraint Language (OCL, 2014. http://www.omg.org/spec/OCL/2.4.

OMG15. Object Management Group. Unified Modeling Language (UML), 2015. http://www.omg.org/spec/UML/.

OMG17. Object Management Group. OMG Systems Modeling Language (OMG SysML), 2017. http://www.omg.org/spec/SysML/1.5/.

PKB13. L. Pruijt, C. Köppe, and S. Brinkkemper. Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support. In *2013 IEEE International Conference on Software Maintenance*, 2013.

PvDB12. M. C. Platenius, M. von Detten, and S. Becker. Archimetrix: Improved Software Architecture Recovery in the Presence of Design Deficiencies. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 255–264, 2012.

Rot17. Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

RSW+15. Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

Rum16. Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

Rum17. Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

Sch12. Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

SSC96. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring Compliance of a Software System with Its High-level Design Models. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 387–396. IEEE Computer Society, 1996.

TMD09. R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

vDB11. Markus von Detten and Steffen Becker. Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, pages 23–32. ACM, 2011.

Völ11. Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

Wei12. Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

WHR14. J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.