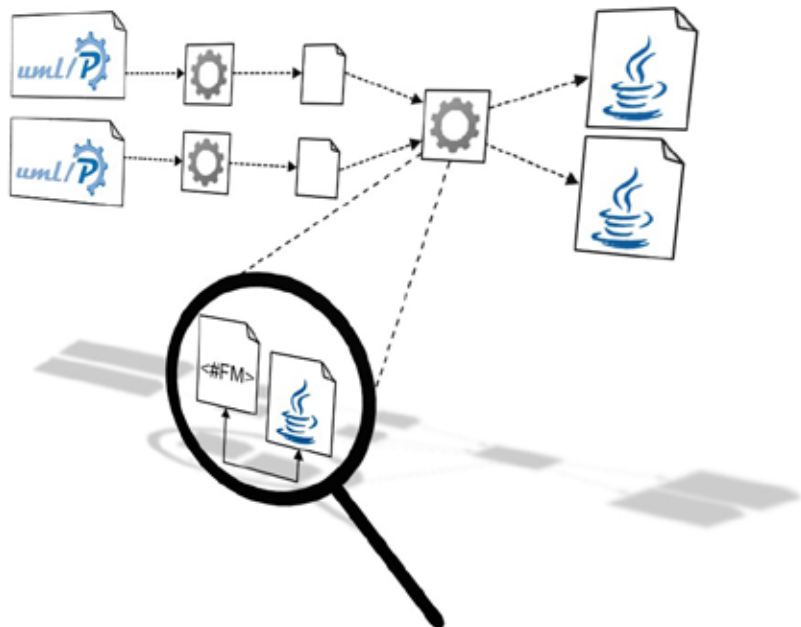


Timo Greifenberg  
Steffen Hillemacher  
Bernhard Rumpe

# Towards a Sustainable Artifact Model

Artifacts in Generator-Based  
Model-Driven Projects



Aachener Informatik-Berichte,  
Software Engineering

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Band 30

# **Aachener Informatik-Berichte, Software Engineering**

herausgegeben von  
Prof. Dr. rer. nat. Bernhard Rumpe  
Software Engineering  
RWTH Aachen University

Band 30

**Timo Greifenberg**  
**Steffen Hillemacher**  
**Bernhard Rumpe**  
RWTH Aachen University

## **Towards a Sustainable Artifact Model**

Artifacts in Generator-Based Model-Driven Projects

Shaker Verlag  
Aachen 2017



**Bibliographic information published by the Deutsche Nationalbibliothek**

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <http://dnb.d-nb.de>.

Copyright Shaker Verlag 2017

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publishers.

Printed in Germany.

ISBN 978-3-8440-5678-5

ISSN 1869-9170

Shaker Verlag GmbH • P.O. BOX 101818 • D-52018 Aachen

Phone: 0049/2407/9596-0 • Telefax: 0049/2407/9596-9

Internet: [www.shaker.de](http://www.shaker.de) • e-mail: [info@shaker.de](mailto:info@shaker.de)

# Abstract

Model-driven development (MDD) is an enabler for the automatic generation of programming language files for products or for tests from explicitly defined models. MDD projects manage a large magnitude of artifacts (files, etc.) with various relationships.

A large class of artifact relations comes from artifacts *using* others, e.g., via importing types and signatures. This form of usage strongly differs from *generation* dependencies, where one artifact is generated, compiled, and transformed from or to other artifacts.

An MDD project usually entails a number of potentially dependent process steps, where a chain of artifact generations, compilations, and packagings arises. During these steps a multitude of artifacts are created, read or even executed. Those artifacts are thus related to each other in various ways.

The number and complexity of occurring dependencies and other relationships between development artifacts can lead to several problems, such as poor maintainability and long development times of both, MDD tools and the product, in an MDD process. To tackle these problems, it is important to understand which artifacts are involved and how these artifacts are related to each other in MDD projects.

In this report, we (1) develop an abstract and rather general artifact model and (2) apply the artifact model by examining in detail the kinds of artifacts and related concepts relevant for a form of wide-spread projects, namely Java projects. We also dive into the core of generative projects, by looking at the generator as a set of artifacts executed at design time. Artifacts are regarded as storable and explicitly named elements of MDD projects, such as model files, directories, libraries, and source code files. Thus, artifacts are the physical manifestation of all information in an MDD project.

For a precise definition of all relevant concepts, we introduce the *Artifact Model* (AM), which allows the precise, model-based specification of involved kinds of artifacts, corresponding concepts, and their relations. The AM can also be considered as a specific form of *meta-model* for models representing the concrete elements and relations between these in MDD projects.



# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. How to Read the CD4A and OCL Specification . . . . .	4
1.2. Contents of the Report . . . . .	5
1.3. Acknowledgements . . . . .	5
<b>2. Essence of Artifact Models</b>	<b>7</b>
2.1. Artifacts and Artifact Containers . . . . .	7
2.1.1. Artifacts . . . . .	8
2.1.2. Relations between Artifacts . . . . .	9
2.1.3. Artifact Containers . . . . .	10
2.1.4. Directories . . . . .	12
2.1.5. Archives . . . . .	12
2.2. Systems and Tools . . . . .	13
2.2.1. Systems . . . . .	13
2.2.2. Modules . . . . .	14
2.3. How to Use the Artifact Model . . . . .	15
<b>3. Extending the Artifact Model to Java</b>	<b>17</b>
3.1. Java Source and Class Files . . . . .	17
3.1.1. Java Artifacts . . . . .	18
3.1.2. Java Source Files . . . . .	18
3.1.3. Java Class Files . . . . .	19
3.1.4. Java Archives . . . . .	20
3.2. Relation Between Java Artifacts and Types . . . . .	20
3.2.1. Packages . . . . .	21
3.2.2. Java Types . . . . .	21
3.3. Detailed Examination of Java Artifacts and Types . . . . .	25
3.3.1. Case 1: Java Compiles All Files It Relies On . . . . .	26
3.3.2. Case 2: Where Java Looks for Types . . . . .	27
3.3.3. Case 3: Where Java Looks for Inner Types . . . . .	29
3.3.4. Case 4: How Java Handles Archives . . . . .	30
3.3.5. Summary . . . . .	31
<b>4. Modelling Languages and Their Definitions</b>	<b>33</b>
4.1. Languages . . . . .	34

4.2. Grammar-Based Definitions . . . . .	34
4.3. ModelFiles Conform to Languages . . . . .	35
<b>5. Class Diagrams in the Artifact Model</b>	<b>37</b>
<b>6. Generators and their Artifacts in MontiCore</b>	<b>41</b>
6.1. Static Artifact Structures . . . . .	41
6.1.1. Templates . . . . .	42
6.1.2. Generators . . . . .	44
6.2. Dynamic Monitoring of Tool Executions . . . . .	45
6.2.1. Representing Actions and Events . . . . .	45
6.2.2. Actions in a Generation Process . . . . .	47
6.2.3. Tools Read and Create Artifacts . . . . .	49
6.2.4. Template and Java Files Contribute to Artifacts . . . . .	50
<b>7. Artifacts in Maven-managed Java Projects</b>	<b>53</b>
7.1. Maven Modules . . . . .	54
7.2. Relations between Maven Modules . . . . .	55
7.3. Target Directories and Target Artifacts . . . . .	56
7.4. POM and VCSRootDir . . . . .	57
7.5. Maven Phases . . . . .	59
7.6. Executing Maven . . . . .	60
<b>8. Applications of the Artifact Model</b>	<b>65</b>
8.1. Analyses based on Tool Monitoring . . . . .	66
8.2. Understanding the Module/Artifact Architecture . . . . .	66
8.3. Generated Systems . . . . .	69
8.4. Template Relations Induced by Generated Artifacts . . . . .	70
8.5. Incremental Toolchain Execution . . . . .	71
8.6. Unused Imports . . . . .	73
<b>9. Conclusion</b>	<b>75</b>
9.1. Considerations on the Artifact Model . . . . .	75
9.2. Multi-level considerations on CD4A and OCL . . . . .	76
<b>Bibliography</b>	<b>79</b>
<b>A. Merged Artifact Model</b>	<b>91</b>
<b>B. Entire Application Model</b>	<b>109</b>
<b>Index</b>	<b>113</b>

# Chapter 1.

## Introduction

A project usually develops, modifies, and uses a large number of artifacts. These artifacts may be documentations, e.g., in word or excel files, models of various kinds, test definitions, but especially code files. For us, an artifact is an independently storable and editable unit of information, and therefore usually a file in the file system. Managing artifacts and their various dependencies is a major issue in larger projects.

In Model-Driven Development (MDD) projects, model artifacts are used to capture various information needed during the development process on various abstraction levels. Thus, MDD aims at employing models as primary development artifacts to abstract from technological and reoccurring details.

In MDD projects, not only the number of artifacts increases but the kinds of artifacts and their relationships become more complex as well. Some of these relationships should lead to automatic re-execution of generative or compiling tools when the source artifacts change, and thus these relations need to be precisely understood. Model transformations and code generators perform model-to-model and model-to-text transformations [CH03, HRW15, GMR<sup>+</sup>16, Rum17] to generate the source code of a complex software system. For the realization of such a process, a multitude of different elements such as models, templates, source code, transformations, directories, languages, and generators are involved. They participate in different states of the MDD process such as design, language and tool development, and product development.

The mentioned elements are related to each other in several ways: (1) elements can use each other statically, e.g., a model file imports another model file when parts of the other model are needed for the definition of the former one. (2) Elements may be generated using other elements within the MDD process. For example files are generated only when the generator exists and has been executed. Furthermore, generated files relate to the source files that the generator uses. (3) Elements can also contain each other allowing to structure projects. For example, archives contain other artifacts.

The number and the complexity of these dependencies lead to a number of challenges in MDD projects. Such challenges are: (1) poor maintainability due to unnecessary and unforeseen impacts of required changes, (2) inefficient processes performing unnecessary process steps, (3) long development times as inadequate organized dependencies constitute a source of errors, (4) hindering the reuse of single components of software engineering



tools or (partly) generated target systems caused by unnecessary dependencies preventing the extraction of components.

We believe that, due to the large number of involved artifacts, their various kinds of dependencies, the number of employed software engineering tools, and the high degree of automation in such projects, it is crucial to provide systematic solutions to manage the complexity of MDD projects. As a first step to tackle these problems it is important to understand which artifacts are involved and how these artifacts are related to each other in MDD projects.

Only then we are able to analyze the structure of a project in terms of artifacts and dependencies, understand how well structured and modularized a project is, and package only the necessary artifacts leaving out unused artifacts. Refactoring [Fow99] allows to improve the structure of the system as well as the project containing all source artifacts, generators, and other tools used.

A number of attempts have been made to capture, analyze, and visualize MDD project related elements and their dependencies such as:

- Dependencies between models for the evolution of models and corresponding strategies for model evolution [VKB13, DRDRIP14, Wen14, DRELHE15, KEK<sup>+</sup>15, SPBS15].
- Dependencies between models as part of an integrated mega model [FLV12, HSG12, VJBB13, BSS14, SPBS15].
- Dependency analysis of executable programs as well as their source code [SJSJ05, WL08, Die12, NWi15, Son15, Sta15, Str15].
- Tracing of MDD process dependencies and program execution traces [DS10, CHGZ12, LM12, ALB<sup>+</sup>14, LvdA15].

All these approaches provide solutions for parts of MDD projects. However, not all relevant elements of MDD projects have been taken into account sufficiently, such as templates or handwritten source code files. These elements introduce additional unconsidered relations to MDD projects and thereby increase the project's complexity. These relations occur between different kinds of model elements such as *relies on* relations between templates and generator source code as well as between elements that participate in different parts of the MDD process (e.g., the *generates* relation between templates, which are part of the generator, and the generated source code files, which are part of the target product). Moreover, none of the mentioned approaches is capable to cover complete MDD projects, as they focus only on specific parts. We believe that by taking the data of the overall MDD project into account and learning from it, the performance of the process as well as the quality of both the target product and MDD byproducts such as reusable models, languages, and tools can be improved.

In this report, we present the notion of *Artifact Models* (AMs), which enable to precisely model element and relation types of MDD projects. AMs serves as a basis for tackling

---

the mentioned problems. [BGRW17] already presents a first idea of such a model and discusses foreseen challenges for its application. In an AM, different types of MDD project elements and dependencies between these are considered. This includes *contains* relations, static *refers to* relations and dynamic *produces* relations as well as involved elements relevant for the different phases of an MDD process. To show the applicability of the approach, an AM for a specific code generator is provided in this report. Parts of this AM can be directly reused within other types of projects. Other parts need to be adapted or can serve as an example on how to model MDD projects by AMs.

A concrete AM for a specific set of MDD projects allows the precise definition of the involved element types and the relation types that can exist between elements. Moreover, derived properties can be specified within or based on the AM. Furthermore, data of MDD projects can be directly extracted in the format of the AM. Thus, existing modeling tools can be reused to create the AM, check the conformance of the data to the AM, perform the calculation of derived properties automatically, and provide capabilities for analyzing and visualizing the data in a comfortable way.

By taking the integrated project data of MDD projects into consideration, the following capabilities can be obtained:

- Evolution among all parts of MDD projects, such as model evolution, tool evolution, and product evolution, can be planned and performed, as the relations between the relevant parts of a project are made explicit and thus, can be easily taken into account when performing the necessary actions.
- The optimization and correction of the overall MDD process is possible based on the extracted data to ensure a correct order of process execution, leading to deterministic and repeatable code generation results.
- Incremental code generation becomes possible because of the explicit specification of all necessary dependencies. The resulting speed up allows for a more agile development, due to short generation, compilation, and testing cycles.
- Consistency of tools and processes to a desired architecture can be ensured by automatically analyzing the extracted data.

The approach improves the overall understanding of MDD projects, i.e., of their involved elements and relations. This may include the usage of generator customization points of a configurable generator [GMR<sup>+</sup>16, Rot17] or the integration between handwritten and generated code [GHK<sup>+</sup>15a, GHK<sup>+</sup>15b] as the product developer can investigate the data to evaluate if the generator is utilized adequately. Moreover, the model-based representation facilitates the communication between the different stakeholders, such as language engineers, tool developers, product developers, and product operators [KRV06]. Thus, the contributions of this report are:

- The notion of *Artifact Models* describing the structure of a project in terms of its artifacts and their various relations.

- An artifact model for generator-based model-driven projects, consisting of core part (cf. Chapter 2) that can be reused for various kinds of projects and extended by adding specific forms of artifacts and relations. The artifact model for generator-based model-driven projects consist, among others, of a part to describe plain Java projects and a part required to describe MDD projects, where not only the final product is developed, but also generators are used and potentially adapted.
- Several initial ideas for the application of an AM are presented (cf. Chapter 8). These kinds of analyses serve as a basis for increasing the quality of the overall project and its structure, e.g., by enhancing the generation process (faster, more reliable) or decoupling sub-systems.

## 1.1. How to Read the CD4A and OCL Specification

In the following, we are describing an AM as a structure of files, directories, archives, etc. Therefore, we are using the notion of class diagrams, in particular *CD4A* as defined in [Rot17], to precisely define this structure. *CD4A* is a textual notation for class diagrams, which restricts the usage of some language concepts, such as methods and modifiers, compared to the *UML/P* class diagram language [Sch12, Rum16]. Listing 2.2 is the first part of the AM described using the *CD4A* modeling language. It is marked with *AM* to identify it as part of the artifact model.

The first AM part using *CD4A* associations is presented in Listing 2.3. An association starts with a keyword, defines its name (e.g., `refersTo`) and the source and destination classes (e.g., `Artifact`). Additional elements are the navigation arrow (here `->`), optional multiplicities on each side (here `[*]`), role names and modifiers (such as `/` for derived).

Furthermore, where relations and constraints are more detailed, the logic language *OCL* is used as precise technique to further specify the properties of the AM. The *OCL* variant used, e.g., in Listing 2.2 is defined in [Rum16] and uses a Java-like syntax for property description, but is otherwise conceptually rather similar to the *OCL* standard [OMG14]. When referencing *OCL* constraints of a specific listing only the first line of the constraint is referenced in this report. An example of an *OCL* constraint can be found in Line 10 of Listing 2.2.

To give the reader a better overview, the report furthermore uses graphical illustrations of the core classes and associations of the AM like in the Figure 2.1. However, these figures are redundant to their textual counterparts and do not contain extra information but usually omit all attributes.

Starting with Section 6.2, we also use *CD4A* and *OCL* to describe an execution of the tools that generate a system's source code. A tool execution is a sequence of actions (that are furthermore hierarchically structured). In principle, actions are described in

a behavioral language. In this case a UML sequence diagram's like-mechanism would usually be used.

However, we encode behavior in form of a protocol (cf. Subsection 6.2.1) that is defined as ordered sequence of action objects and store start and end times within each action. We have therefore "objectified" the actions. This has two advantages: (1) For describing the protocol, we can use CD4A and OCL in the same form as for the structure of the AM, and (2) we can even use OCL to relate the AM structure and a tool execution that is generating artifacts.

## 1.2. Contents of the Report

The remainder of the report is structured as follows: First, Chapter 2 introduces the essence of the AM defining the most general parts with a high potential of direct reuse. Afterwards, Chapter 3 examines Java artifacts and types and extends the AM to Java. Chapters 4 to 6 make further extensions taking into account MDD related concepts such as languages, models, and MDD tools. Furthermore, Section 6.2 deals with dynamical process relations, which allow to describe the execution of MDD build processes. The modeling of Maven as an exemplary build tool is presented in Chapter 7, before Chapter 8 shows beneficial possibilities to apply the AM. Last, Chapter 9 concludes this report.

## 1.3. Acknowledgements

It is a pleasure to thank Pedram Mir Seyed Nazari and Andreas Wortmann for useful discussions. Moreover, Arvid Buttin, David Schmalzing, Filippo Grazioli, and Matthias Markthaler gave valuable hints on draft versions of this text.

This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.



# Chapter 2.

## Essence of Artifact Models

In large projects, it is very helpful to structure the information and program modules in appropriate forms to be able to develop, manage, evolve, or test individual, encapsulated parts of the system. Furthermore, the reuse of parts from other systems, such as frameworks or components, relies on a good capability for modularization of the system into *artifacts*. In MDD projects quite a number of forms of artifacts occur and their relationships become more complicated.

It is therefore worthwhile to classify the forms of artifacts and their relationships in an *Artifact Model (AM)*.

The specific AM defines basic artifacts as well as relations between these in the context of a software development project. It is capable of linking some elements within the artifacts (such as types, signatures, etc.) to their defining and importing artifacts, but also allows to look at higher-level structures, commonly called modules, components, packages, or subsystems.

In this report an AM is introduced which was mainly created for Java-based MDD projects, more specifically MontiCore-based projects. However, the essentials presented in this chapter are widely reusable to model other projects. The most specific model parts presented in this report serve at least as a basis for the way MDD projects can be modeled by AMs.

In the following, the AM is introduced incrementally. The most general elements, which therefore have a high potential for direct reuse in other software development projects, are introduced first. This core part of the AM for MontiCore-based projects serves as an extension point for other projects with different artifact types in use. A complete version of the AM is depicted in Appendix A. During the introduction of the different elements of the AM, the corresponding excerpts of the model are presented to give a better understanding of the way they are modeled. Henceforth, the AM presented in this report is simply referred to as *the AM*.

### 2.1. Artifacts and Artifact Containers

The core elements of the AM are artifacts. Any project-related file or directory can be seen as an artifact. We define an artifact as follows:

**Definition 1 (Artifact)** *An artifact is an individually storable and referenceable element serving a certain purpose in the context of a software engineering process.*

Typical artifacts are files, such as Java source or class files, but also model files or documentation units. However, in a database-oriented development setting, artifacts may have a different shape.

The AM allows that artifacts can contain other artifacts. Typical examples are archives, directories, but potentially also database files. Figure 2.1 gives an overview of the physical organization of artifacts.

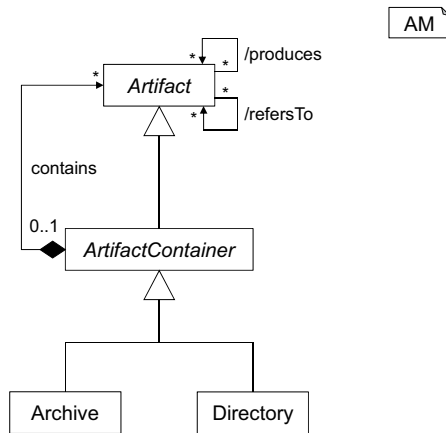


Figure 2.1.: Organizing artifacts in containers.

It should be noted that, because we use the composite pattern [GHJV95] for this part of the AM, archives and directories can contain each other arbitrarily, but each artifact is contained in at most one artifact container. If the artifacts are modeled completely, there is only one concrete element which is not contained in an artifact container: the root directory. This directory depicts the root of the file system (see Subsection 2.1.4). Moreover, artifacts can be used to produce other artifacts and can statically refer to other artifacts. A definition for these relations can be found in Subsection 2.1.2.

### 2.1.1. Artifacts

Artifacts are the core elements of the AM. There are many different types of artifacts, some of which are introduced in later chapters. In Listing 2.2 the common properties of all artifacts are modeled in the abstract class *Artifact*. Each artifact has a simple name and a name extension. This allows artifacts to be used from other artifacts.

In addition, each artifact contains the date of the last time it was modified (attribute *modified*). The Attributes *name*, *fullName*, and *isRoot* can be derived from

```

1  abstract class Artifact {
2      String simpleName;
3      String nameExtension;
4      Date modified;
5      /String name;
6      /String fullName;
7      /boolean isRoot;
8  }
9
10 context Artifact inv:
11     name == ((nameExtension == "") ? simpleName :
12         simpleName + "." + nameExtension);

```

Listing 2.2: AM: Artifact.

the information available in the artifact structure and are therefore specified by OCL constraints.

The constraint shown in Listing 2.2 describes how the attributes `name`, `simpleName`, and `nameExtension` are related. If the name extension of an artifact is empty, the name is equal to the simple name. The full name of an artifact resembles the absolute path of the file in the file system and is specified later.

## 2.1.2. Relations between Artifacts

```

1  association /refersTo [*] Artifact -> Artifact [*];
2
3  association /produces [*] Artifact -> Artifact [*];
4
5  context Artifact a inv:
6      !a.produces**.contains(a);

```

Listing 2.3: AM: Core relations.

One core element of the AM is the `refersTo` relation between artifacts, as shown in Figure 2.1. Specific forms of `refersTo` relations between different kinds of artifacts are united in this relation.

**Definition 2 (Refers to Relations Between Artifacts)** *If an artifact in some form needs information from another artifact to fulfill its purpose, it refers to the other artifact.*

Referring to other artifacts is a binary relation, with many concrete incarnations. Typically, Java classes rely on other Java classes, when they use their types, methods, or



inherit from their superclasses. In Java this typically (but not exactly always) manifests as `import` or subclassing. Other ways would be, for example, the use of full qualified names. Because this form of relation is usually defined through names that act as references, we call it `refersTo` in the AM. Please note that artifacts may mutually rely on each other, which generally allows `refersTo` to be cyclic.

The third and last core element of the AM is the `produces` relation between artifacts that describes when information from one artifact are used to produce (generate, compile) the other.

**Definition 3 (Produces Relations Between Artifacts)** *An artifact can be produced by automatic application of a tool. The tool may use existing source artifacts for this production. Thus, an existing artifact contributes to the production of the new artifact if its existence or content has influence on the produced resulting artifact.*

The production of a new artifact does not destroy or modify the source artifacts. One or more artifacts are created containing all the information needed from the sources. This has two consequences: (1) the new artifact typically has no `refersTo` relationship to its sources. (2) The `produces` relation has no cycle. It may be that one artifact is produced from several sources and one source may be used several times, but the resulting graph structure is definitely acyclic (specified by constraint Line 5 using transitive closure (\*\*\*) of Listing 2.3).

As a tool itself consists of a set of artifacts, the tool's artifacts also have to be considered as production elements for generated artifacts. This is especially interesting, when tools are not considered as a black box, but can rely on artifacts, which are generated by meta-tools. The `produces` relation is therefore dedicated to capture all forms of artifacts necessary to successfully produce a resulting artifact.

As already mentioned, there exist many forms of *produce* relations caused by, e.g., a simple copy script, a compiler, a code generator, or an archive tool. These relations all have in common that the actions which caused them are executed automatically and can be re-executed at any time. No humans are involved, but usually a build script knows what to do – and in particular also, when to redo it.

The AM currently does not reflect human activities in all their detail. From its current version, we can derive if an artifact is created by humans or has been produced by tools. However, we do not capture artifacts that have been produced by tools originally and modified by humans afterwards. These are generally handcrafted artifacts. The reason behind this approach is that we generally believe in automatic production of artifacts only in repeatable form. Hence, humans should not modify any produced artifact.

### 2.1.3. Artifact Containers

**Definition 4 (Artifact Container)** *An artifact container is an artifact that contains a set of conjointly used artifacts.*

Technically `ArtifactContainers` are composed using the composite design pattern [GHJV95]. Thus, `ArtifactContainer` is a specialization of `Artifact`. We designed this composite in the AM, because `ArtifactContainers` are also handled as individual, reusable units in development projects. An artifact container only has the inherited properties and adds no further attributes. Artifact containers can contain other artifacts represented by the `contains` composition from `ArtifactContainer` to `Artifact` in Figure 2.1.

By default two forms of `ArtifactContainers` are provided, namely `Directory` and `Archive`.

```

1  abstract class ArtifactContainer extends Artifact {}
2
3  composition contains
4      [0..1] ArtifactContainer (parent) -> Artifact [*];
5
6  context Artifact inv:
7      isRoot <=> parent.isAbsent &&
8      !isRoot implies {p in parent** | p.isRoot}.size == 1;
9
10 context Artifact inv:
11     fullName == (isRoot ? "/" :
12                 (parent.isRoot ? "/" + name :
13                 parent.fullName + "/" + name));
14
15 context Artifact a, Artifact b inv:
16     a.name == b.name && a.parent == b.parent
17     implies a == b;

```

Listing 2.4: AM: `ArtifactContainer`.

In Listing 2.4 further constraints for the `ArtifactContainers` are given. The first constraint ensures that only the root has no parent and that there exists exactly one root for each file system.

The full name of artifacts is derived from the hierarchy structure of artifact containers. It serves as an identifier for artifacts within a file system. This even holds for archives as they are modeled as whitebox containers. That is, we assume that we can access artifacts within archives and also identify them via their full names. This is possible as the full name for all artifacts is unique including the full name of artifact containers even if they are of a different kind.

The last constraint of Listing 2.4 ensures that the full name of each artifact is unique within a file system (Line 15). Note that the presented constraints allow to capture more than one file system, as there is no restriction for the number of root directories and the name of an artifact must only be unique in the scope of its parent. That is, the full name of artifacts is only unique in the scope of a single file system.

### 2.1.4. Directories

```

1  class Directory extends ArtifactContainer {}
2
3  context Directory inv:
4    nameExtension == "";
5
6  context Artifact inv:
7    isRoot implies (this in Directory);
8
9  context Artifact inv:
10   isRoot <=> simpleName == "/";
11
12 context Artifact inv:
13   name.contains("/") implies isRoot;

```

Listing 2.5: AM: Directory.

Directories of file systems can be modeled by the `Directory` class. Nowadays, projects tend to organize their artifacts in complete directories, which are version controlled, generated, copied, etc. Therefore, we have modeled in Listing 2.5 that `Directory` is also an `Artifact`. As a consequence, however, we have to set the `nameExtension` attribute to be always empty (Line 3). Furthermore, we demand that the root `Artifact` must be a `Directory` (Line 6) and, if present, its simple name is “/” (Line 9). Lastly, the constraint in Line 12 specifies that only the root directory may contain a “/” in its name thus having directory names conform to a typical file system.

### 2.1.5. Archives

Archives are used to collect several artifacts in a single file, which makes storage and versioning easier. We ignore that files in archives can also be compressed. In general, the content of an archive cannot be accessed directly, but the application of an external program is necessary to extract the contained elements. However, in the AM only the content of archives and not the compression or extraction algorithms is of interest. For this reason, archives are modeled as artifact containers, which means that their content can be accessed without any restrictions. In the AM, archives are modeled as subtypes of artifact containers as shown in Listing 2.6. In contrast to directories no additional constraints are given.

```

1  class Archive extends ArtifactContainer {}

```

Listing 2.6: AM: Archive.

## 2.2. Systems and Tools

In order to understand how systems are modeled in the AM, first the general definition of a software system is considered.

### 2.2.1. Systems

**Definition 5 (System)** *A system is a set of cooperating and connected artifacts that can be executed to fulfill a desired purpose.*

In the AM, the term system is used to describe executable software systems. In C-based projects, a system is often only one artifact, namely the `.exe` file, produced by the linker. In Java projects a system may be contained in a jar archive or actually consist of a set of class files in the directory.

The AM distinguishes between two kinds of considered systems: *tools* and *products*. Tools are systems used during the development process. This is in particular interesting, when the tool itself is modified and thus has to be compiled (or generated) in the same development project as the final product. Furthermore, several tools and products may be of interest. The situation becomes even more tricky, when tools and products share parts of their artifacts.

Figure 2.7 gives an overview over the elements related to systems.

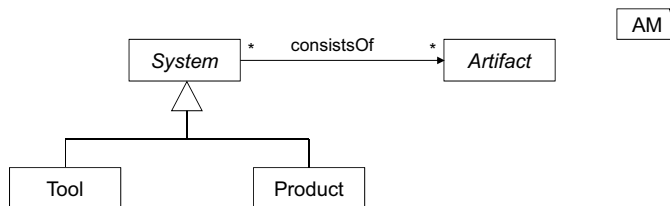


Figure 2.7.: A system consists of artifacts and is either a tool or a product.

A system can be identified by its name and usually also has a version number. The `version` attribute is of type `String` to permit arbitrary version definition (e.g. `1.0.0-SNAPSHOT`). As displayed by the `consistsOf` relation in Figure 2.7 and Listing 2.8, each system consists of any kind of artifacts. Even though a system consists of artifacts, their existence does not depend on the existence of the system they belong to and artifacts can be used in many systems.

As described above, there are two well-known kinds of `Systems` namely `Product` and `Tool`. Both are described in the following Listing 2.9 and Listing 2.10 respectively.

A product is a system that is the main outcome of a project made available to the user with its sources (partly) generated by a generator.

A tool is also a special kind of system. Compared to a product, however, a tool is an executable system involved in the MDD process that reads artifacts as input and produces

```

1  abstract class System {
2      String name;
3      String version;
4  }
5
6  association consistsOf [*] System -> Artifact [*];

```

Listing 2.8: AM: System.

```

1  class Product extends System {}

```

Listing 2.9: AM: Product.

```

1  class Tool extends System {}

```

Listing 2.10: AM: Tool.

output artifacts when executed (see Section 6.2). Tools are not necessarily available to the user. They may be readily available (e.g., like a Java compiler) or must be developed within the project, such as a specific code generator.

## 2.2.2. Modules

In order to model the architectural substructure of a system, a system in the AM can be divided into modules. Modules can be further divided into submodules consisting of artifacts. By using modules, a high level overview over the system's architectural units and their relations can be modeled, as shown in Figure 2.11.

**Definition 6 (Module)** *A module is a set of cooperating and connected artifacts that fulfill a desired purpose, but are not necessarily complete and executable.*

Typical modules are ranging from subsystems, developed by a subset of the developers, to reusable frameworks or class libraries, or even single Java source files. The Java package concept is a possibility to structure modules, but this depends on how developers use their package structures. Besides that, jar archives are often used to organize modules.

In the literature the term *module* is frequently used interchangeably with the terms *component* or *subsystem*.

A module represents an architectural substructure of a system. The architecture represented by modules is based on artifacts. Ideally, the physical structure of artifacts (i.e., their organization in artifact containers) matches the structure of architectural modules. In other approaches, such as [PW15], types are used instead of artifacts to define and analyze the system's architecture.

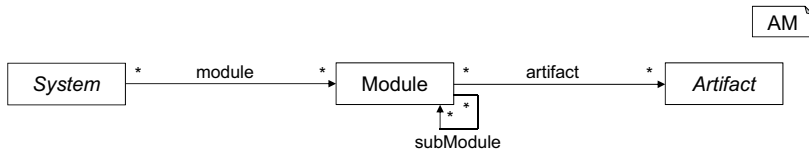


Figure 2.11.: Modules represent the architectural substructure of a system. They consist of artifacts and can be composed of submodules.

```

1  class Module {
2      String name;
3  }
4
5  association module [*] System -> Module [*];
6
7  association subModule [*] Module -> Module [*];
8
9  association artifact [*] Module -> Artifact [*];

```

Listing 2.12: AM: Module.

A set of modules constitutes a system. The `subModule` association enables a way to divide modules into submodules and therefore provides a way to compose modules hierarchically. Besides an arbitrary number of submodules, a module comprises a set of artifacts (defined by the `artifact` relation). Artifacts can be assigned to multiple modules and modules to multiple systems.

While it is possible that a module is reflected by a concrete artifact, such as an archive, it may also be that modules are only used as a conceptual form of structure and are not directly reflected in the artifact structure.

## 2.3. How to Use the Artifact Model

So far, the essentials of the artifact model are rather abstract. They describe general concepts enabling to structure large projects. However, specific projects require specific forms of artifacts, modules, etc.

The idea of the presented AM core is to be extendable by introducing subclasses. The subclasses may be special forms of artifacts, systems, or modules. Along with these special forms of artifacts come specializations for the corresponding associations. While object orientation is well suitable for classification using subclasses, there is no generally accepted form of refining associations.

We therefore use a specific technique for association specialization: We introduce a new association and relate the new one with the existing, refined association using an OCL

constraint. We have borrowed this technique from the field of mathematics. A typical specialization follows the principle used to specify the association `reliesOnJavaArtifact`, defined in the forthcoming Listing 3.2. It refines the `refersTo` association defined in Listing 2.2, because it is connected via an OCL constraint which is basically a `containsAll` statement (see Line 6).

As an alternative, it would have been possible to introduce the AM core as a form of meta-meta-model that should be instantiated to concrete projects, where, e.g., meta-association `refersTo` would then be instantiated accordingly. However, we felt this technique would not be simpler to understand and master. Furthermore, we can imagine situations where a hierarchy of specializations is needed which cannot easily be handled in a meta-setting yet.

In the following chapters, we will discuss several extensions of the AM core to demonstrate its usage.

# Chapter 3.

## Extending the Artifact Model to Java

Java, like many other programming languages, uses files to store individual pieces of code. Therefore Java source and class files are the primary artifacts to deal with. Because Java has a smart technique to identify the artifacts in which external types (classes) are defined with the import statement, we additionally investigate how the Java import statement actually works and how this is modeled in the AM.

### 3.1. Java Source and Class Files

In this section, Java artifacts and relations between different Java artifacts are introduced as a refinement of the AM. These artifacts can be found in Java-based software projects. Figure 3.1 gives an overview of the involved classes of the AM and their relations.

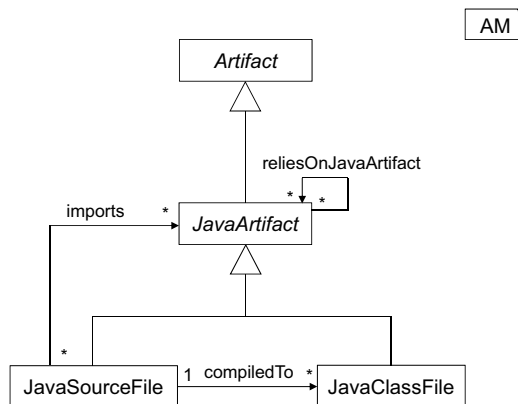


Figure 3.1.: Java artifacts, related concepts and relations.

A Java artifact is either a `JavaSourceFile` or a `JavaClassFile`. Java source files contain source code written in Java [GJS<sup>+</sup>15], while Java class files contain the compiled code to be executed by the Java Virtual Machine [LYBB15]. Class files also contain the complete symbol information for external use. Therefore, importing source



files do not necessarily rely on the source of the imported classes. Symbols are defined as follows:

**Definition 7 (Symbol)** “A symbol definition (or short symbol) contains all essential information about a named model element. It has a specific kind depending on the model element it denotes. A symbol is defined exactly once.” [MSN17]

### 3.1.1. Java Artifacts

The `JavaArtifact` class is a specialization of the `Artifact` class. It is abstract and unifies its two subclasses, the Java source and Java class artifacts. In Listing 3.2, the definition of a Java artifact is given.

```

1  abstract class JavaArtifact extends Artifact {}
2
3  association reliesOnJavaArtifact
4    [*] JavaArtifact -> JavaArtifact [*];
5
6  context JavaArtifact inv:
7    refersTo.containsAll(reliesOnJavaArtifact);

```

Listing 3.2: AM: JavaArtifact.

The homogenous `reliesOnJavaArtifact` relation shown in Figure 3.1 is used to display that Java artifacts can rely on each other. We define that a Java *source* file *relies* on another Java artifact iff the other artifact has to be loaded when compiling the Java source file. A Java *class* file *relies* on another Java class file iff something (type, method, constant, ...) of the other class file is used during execution of the part of the program defined by the class file.

As described in Section 2.3, the `reliesOnJavaArtifact` relation is defined as specialization of the `refersTo` relation (Line 6).

We could already further specialize the `reliesOnJavaArtifact` relation, based on a distinction whether it is defined through explicit import, use of a fully qualified name, inheritance, or other mechanisms. We use this distinction in Section 8.6, e.g., to identify unused imports.

### 3.1.2. Java Source Files

Java source files denote source code artifacts written in Java. They are modeled as a specialization of the abstract element `JavaArtifact`. The specification of the `JavaSourceFile` element is given in Listing 3.3.

A Java source file can import Java artifacts corresponding to the import statements defined within the file. This import relation is represented by the `imports` association as

```

1  class JavaSourceFile extends JavaArtifact {}
2
3  association imports [*] JavaSourceFile -> JavaArtifact [*];
4
5  association compiledTo
6    [1] JavaSourceFile -> JavaClassFile [*];
7
8  context JavaSourceFile inv:
9    reliesOnJavaArtifact.containsAll(imports);
10
11 context JavaSourceFile inv:
12   produces.containsAll(compiledTo);
13
14 context JavaSourceFile inv:
15   nameExtension == "java";

```

Listing 3.3: AM: JavaSourceFile.

shown in Figure 3.1 and does not distinguish source or class imports. The import statement allows to use symbols defined in the imported artifacts, such as the types, methods, etc. Thus, it contributes to the `reliesOnJavaArtifacts` relation (Line 8).

However, an import can be unused (cf. Section 8.6) and foreign references may be fully qualified and thus do not need imports. While it is a good engineering practice that `imports == reliesOnJavaArtifacts`, the relation is not necessarily a subset in any of both directions, since the Java programming language allows for a Java artifact to import other Java artifacts using their full qualified names. Then, no explicit import statements is required.

Furthermore, a Java source file is compiled into multiple Java class files, one for each class defined in the source file. This is modeled by the `compiledTo` relation, which is a refinement of the `produces` relation (see Line 11).

### 3.1.3. Java Class Files

According to [LYBB15], a Java class is the compiled code to be executed by the Java Virtual Machine. It represents a hardware- and operating system-independent binary format, typically stored in the class file.

In the AM, it is assumed that the compiled code is stored in such files, which are represented by the `JavaClassFile` element in the AM. Its AM definition is given in Listing 3.4.

Similar to Java source files, Java class files are modeled as specializations of the abstract `JavaArtifact` class, such that import statements can refer to both at the same time.

```
1  class JavaClassFile extends JavaArtifact {}
2
3  context JavaClassFile inv:
4    nameExtension == "class";
```

Listing 3.4: AM: JavaClassFile.

### 3.1.4. Java Archives

To complete the Java artifacts, we also consider the Java archives, also called `jar` files, because of their names extensions. Figure 3.5 illustrates the embedding of the `jar` file within the AM.

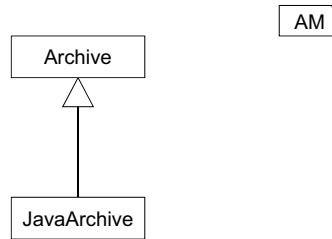


Figure 3.5.: Java archives.

```
1  class JavaArchive extends Archive {}
2
3  context JavaArchive inv:
4    nameExtension == "jar";
```

Listing 3.6: AM: JavaArchive.

The Java archive acts like any other archive. While we expect that it mainly contains Java source and class artifacts it is generally allowed to add any other kind of artifacts as well. Furthermore, the archive internally resembles a directory structure, which is used for example by the Java compiler to quickly find artifacts by their fully qualified name.

## 3.2. Relation Between Java Artifacts and Types

For Java projects, the AM does not only take Java artifacts into account but also Java types and packages. Both form individual relations among themselves and to Java artifacts. Figure 3.7 gives an overview of the different elements and the relations between these within the AM.

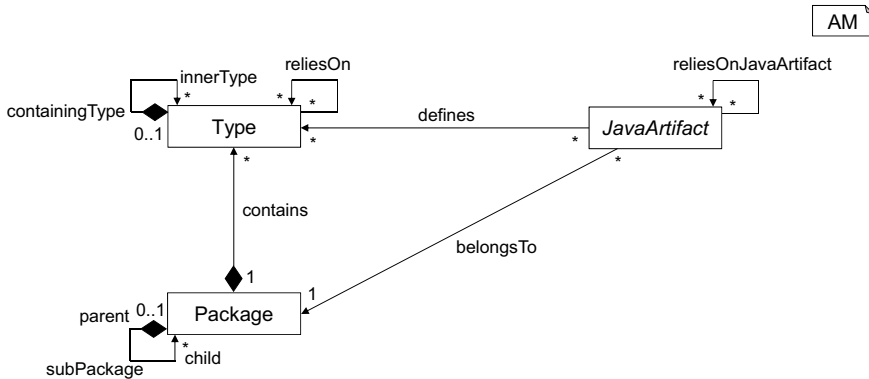


Figure 3.7.: Java types and their relations to Java artifacts.

### 3.2.1. Packages

We start with the `Package` class of Figure 3.7. A package organizes a set of related classes and interfaces [Ora16].

The `Package` class contains a name attribute as well as a `fullName`. Furthermore, each Java source as well as class file belongs to exactly one package. This relation is modeled by the `belongsTo` association in Figure 3.7.

As mentioned earlier, the naming structure of packages suggest a package hierarchy, which is reflected in the `subPackage` relation. This hierarchical definition is based on the Java package specification [GJS<sup>+</sup>15]. While the package hierarchy is a helpful form of structure for humans, Java itself does not capitalize on this hierarchy. In particular, visibilities between parent and subpackages are the very same as between two arbitrary packages.

Furthermore, it should be noted that packages may or may not be identical to modules. Thus, we do not presuppose a given relation between `Module` and `Package`.

Several constraints shown in Listing 3.8 ensure some correctness, such as names of a package must not contain dots, how the full names of a package and a subpackage are related, that the full name of a package is unique, and that there exists exactly one root for each package hierarchy.

We only enforce packages to be present if an artifact is included. Therefore, several packages may not have a parent or no root package is present.

### 3.2.2. Java Types

The relationships between Java artifacts are needed, because some Java artifacts use types and their signatures defined in another Java artifact. We therefore make the type structure explicit in the AM. Thus, the AM contains the `Type` class to describe where types can be found in artifacts. We chose to not distinguish between any specific kind of types, but to

```

1  class Package {
2      String name;
3      String fullName;
4      /boolean isRoot;
5  }
6
7  association belongsTo [*] JavaArtifact -> Package [1];
8
9  composition subPackage
10     [0..1] Package (parent) -> (child) Package [*];
11
12  context Package inv:
13     !name.contains(".");
14
15  context Package inv:
16     isRoot <=> parent.isAbsent &&
17     !isRoot implies {p in parent** | p.isRoot}.size == 1;
18
19  context Package inv:
20     !isRoot implies
21     fullName == parent.fullName + "." + name;
22
23  context Package a, Package b inv:
24     a.parent == b.parent && a.name == b.name implies a == b;

```

Listing 3.8: AM: Package.

use the general class `Type` to represent any form of type system. Henceforth, when we talk about Java types, these types are always represented by instances of the `Type` class of the AM. In the AM, Java types are content elements of a Java artifact. Listing 3.9 shows the corresponding excerpt of the AM.

In general, a Java type is either a Java class, interface, or enum. Within the AM, however, we do not differentiate between them, as the level of detail of the AM abstracts from any specific model element.

The `Type` class describes the types. Each type has a simple name (such as `Person`) and the three derived attributes `name`, `fullName`, and `isInnerType`. The `name` attribute also contains the name of the containing type. Lastly, the `fullName` attribute is equal to the full qualified name of a type. Listing 3.9 further defines the general type structure used in the AM.

Each Java artifact can define an arbitrary number of Java types. Every type can be defined by an arbitrary number of Java artifacts. This relation is modeled by the `defines` association (Line 8).

```

1  class Type {
2      String simpleName;
3      /String name;
4      /String fullName;
5      /boolean isInnerType;
6  }
7
8  association defines [*] JavaArtifact -> Type [*];
9
10 composition contains [1] Package -> Type [*];
11
12 composition
13     [0..1] Type (containingType) -> (innerType) Type [*];
14
15 association reliesOn [*] Type -> Type [*];
16
17 context Type inv:
18     isInnerType <=> !containingType.isAbsent &&
19     isInnerType implies
20         {t in containingType** | t.isInnerType}.size == 1;
21
22 context Type inv:
23     name == (isInnerType ?
24         containingType.name + "." + simpleName : simpleName);
25
26 context Type inv:
27     fullName == (package.isRoot ? name :
28         package.fullName + "." + simpleName);

```

Listing 3.9: AM: Type.

Furthermore, a type is a member of exactly one package, whereas a package can contain an arbitrary number of types [GJS<sup>+</sup>15]. This is modeled by the `contains` composition (Line 10).

The homogeneous relation of the `Type` class models that a type can define multiple inner types. More specifically, a type can have an arbitrary number of inner types, while it can only be contained by one other type (Line 12). This relationship is important when inner types are used externally in a qualified form, because the outer type then acts as a part of the qualifier.

Finally, a type `A` relies on a type `B` iff `B` itself or one of its elements (method, constant, ...) is syntactically used to define `A`. This also includes the encapsulated use of, e.g., type `B` for a local variable or private attribute in `A`. The `reliesOn` relation summarizes all relations introduced in [PKB13, PW15] (Line 15).

The first constraint shown in Listing 3.9 states that inner types have a `containing-Type` (Line 17). Moreover, the name of an inner type is composed by the name of the containing type and its own simple name separated by a dot (Line 22). Finally, the full name of a type located in the root package is the type's name. The full name for any other type composed of the name of its containing package and its own name separated by a dot (Line 26).

```

1  context JavaSourceFile inv:
2     defines == compiledTo.defines;
3
4  context JavaSourceFile inv:
5     {t in defines | !t.isInnerType}.size == 1;
6
7  context JavaArtifact a1, a2,
8     Type t1 in a1.defines, Type t2 in a2.defines inv:
9     t1.reliesOn.contains(t2) implies
10    a1 == a2 || a1.reliesOn.contains(a2);
11
12 context JavaArtifact a, Type t in a.defines inv:
13    !t.isInnerType implies t.simpleName == a.simpleName;
14
15 context JavaArtifact inv:
16    parent.fullName.replaceAll("/", ".")
17    .endsWith(belongsTo.fullName);
18
19 context JavaSourceFile inv:
20    forall n in { t.fullName.replace(".", "/") |
21    t in defines && !t.isInnerType }:
22    fullName.endsWith(n + ".java");
23
24 context JavaClassFile inv:
25    parent.fullName.replaceAll("/", ".")
26    .endsWith(defines.package.fullName);
27
28 context JavaClassFile inv:
29    simpleName == defines.name.replaceAll(".", "$");

```

Listing 3.10: AM: Relation between Java artifacts and types.

Listing 3.10 concludes this subsection with a set of OCL statements that further refine the relations between Java artifacts, Java types, packages, and directories as modeled in the AM.

Line 1 focuses on Java source files and the Java class files to which they are compiled to. The types defined by a Java source file and the ones defined by the corresponding Java class files must be the same.

Line 4 enforces that each Java source file defines exactly one top level Java type. Note, that this is modeled for simplicity and is not enforced by the Java compiler. It is possible to compile a Java source file without defining any public type. Multiple top level types can be handled by the Java compiler as well. Properties and behavior of the default Java compiler are discussed in more detail in Section 3.3 by taking a closer look at some corner cases of the Java programming language.

Line 7 states that if a type  $A$  relies on a type  $B$ , and both are defined by Java artifacts, they are either defined in the same Java artifact or their Java artifacts rely on each other in the same way.

All further OCL statements constrain the relations of names as known from Java. Constraint Line 12 demands that the simple name of a Java artifact and the non-inner type it defines are equal.

Constraint Line 15 enforces that the full name of the directory containing the Java source file must end with the full name of the package declared by the Java source file. Nevertheless, the directory may be deeper nested. This constraint is not enforced by the Java language definition (see Section 3.3), but by many tools including the Eclipse IDE [Ecl15].

Line 19 enforces that the full name of a Java source file corresponds to the full name of its defined top level type modulo separator replacement.

Line 24 defines that the name of the package declared by the Java source file, which is compiled to Java class files, must be reflected in the file system, i.e., the full name of the folder containing these Java class files must end with the full name of the package declared by the Java source file (see Section 3.3). Lastly, Line 28 demands that the simple name of a Java class file corresponds to the name of its defined type. If the type is an inner type, dots are replaced by “\$” chars.

## 3.3. Detailed Examination of Java Artifacts and Types

Many Java tools capitalize on the following rule:

**Definition 8 (Java Type Artifact Correspondence Rule)** *A type with name  $T$  is stored in an artifact of the same name  $T$ . `java`. This also holds for full qualified types  $q$ .  $T$  that are stored in `q/T.java`*

This rule is not enforced by the Java language definition, but by many tools, because it greatly simplifies the lookup mechanism for types. Based on this rule import statements basically refer to other types as well as to artifacts. In contrast to this, other languages like `C` or `C++` explicate that their import statements refer to artifacts exclusively.

This Java correspondence rule is also very helpful for developers, because it allows them to widely ignore the difference between classes and artifacts. They can write the desired types into the import statement and thereby automatically refer to their defining artifacts.



However, the AM clearly separates both concepts and their relationships. For further motivation of this separation, in the rest of this section we demonstrate some special corner cases that occur in Java and make this distinction necessary.

The presented corner cases will help to understand the programming language Java and its import statement. Moreover, these cases are used to give a better understanding of the way Java-based projects are modeled in the AM. For each of the special cases, first, its setup is presented. Next, it is explained, which actions are performed, i.e., the way the Java default compiler was used. Afterwards, the results of the specific compilation attempts are discussed in detail in addition to the direct consequences for the AM. As a final remark, if not explicitly stated otherwise, we assume for all the cases that the directory structure on the file system matches the package structure.

### 3.3.1. Case 1: Java Compiles All Files It Relies On

The first case in Listing 3.11 shows a standard situation with four Java classes D, E, F, and G, where each relies on the previous one in a different form.

```
1 // artifact p/D.java
2 package p;
3
4 public class D {}
```

Java

```
1 // artifact p/E.java
2 package p;
3 import p.D;
4
5 public class E {}
```

Java

```
1 // artifact p/F.java
2 package p;
3
4 public class F {
5     E e;
6 }
```

Java

```
1 // artifact p/G.java
2 package p;
3
4 public class G extends F {}
```

Java

Listing 3.11: Java source artifacts D, E, F and G for Case 1.

Each of these classes may be compiled individually. However, if we only compile class G by using

```
1 javac p/G.java
```

we get the other four classes compiled as well. This doesn't happen if we compile E, where only two classes are produced.

```
1 javac p/E.java
```

We learn: Java includes its own dependency management. It knows which artifacts it has to consider when compiling an artifact. So an import statement serves at the same time as reference to both, the imported type and the artifact where the type is stored.

As an aside, if the superclass artifact F changes and G is recompiled, then Java detects a potential need for recompilation. However, this need is not detected along the transitive closure. Modifying D.java and then recompiling only F.java does not lead to a recompilation of E.java and D.java. So it is unsafe to rely on Java's dependency management when building a system incrementally.

#### 3.3.2. Case 2: Where Java Looks for Types

The setup of the second case, an unusual situation with three Java classes A, B, and AlsoDefinesA, is shown in Listing 3.12.

```
1 // artifact A.java
2 package p;
3
4 public class A {}
```

Java

```
1 // artifact B.java
2 package p;
3 import p.A;
4
5 public class B {}
```

Java

```
1 // artifact p/AlsoDefinesA.java
2 package p;
3
4 class A {}
5
6 class AlsoDefinesA {}
```

Java

Listing 3.12: Java source artifacts A, B, and AlsoDefinesA for Case 2.

The situation is unusual as `A` claims it belongs to package `p`, but is not stored inside the directory `p`. Furthermore, the artifact `AlsoDefinesA` defines two types, because it also produces a version of class `A`.

```
1 javac A.java
```

compiles well, but unfortunately creates the class file `A.class` in the same (current) directory, which does not correspond to package `p`. This is why the compilation of `B` fails, pretending not to find the desired symbol `A`:

```
1 javac B.java
```

This failure is independent of whether `A` has been compiled before. However, if we compile `A` at the wrong place and move it to the desired directory, then the compilation is successful:

```
1 javac A.java
2 mv A.class p
3 javac B.java
```

So Java does not look in all directories, not even in the current directory, for potential places where an artifact can be, but uses the concatenation of package and class name as artifact identification and thus as destination to look at.

As an aside, file `A.class` is also created by the unusual file `p/AlsoDefinesA` that contains two top-level type definitions. When we compile this in the following order, the compilation is successful:

```
1 javac p/AlsoDefinesA.java
2 javac B.java
```

This example also shows that the import statement looks for both the class files and the source files, but is already satisfied if it finds one of these artifacts. However, if both files are compiled together like in

```
1 javac B.java A.java
```

then the compilation is successful, because the compiler uses the compiled version of `A` in the internal buffer and does not look externally at all.

We learn: The Java compiler only looks at certain destinations for certain artifacts. If an artifact is in the wrong place, then it is not used even if it would contain the correct type.

If, however, the artifacts are compiled together, then the internal buffer stores types with their qualified names and not artifacts.

As a note, we would like to add that the smartness of the Java compiler managing dependencies between Java files has a lot of advantages for standard Java projects. Nonetheless,

it also introduces problems, when a larger number of files are generated and the dependency management, as very well done by *Make* [SMS15], has to include artifacts that serve as sources for generated Java files.

### 3.3.3. Case 3: Where Java Looks for Inner Types

The third case, whose setup is shown in Listing 3.13, is related to the second case. It also demonstrates how Java manages packages and inner types.

```

1 // artifact M.java
2 package x.y;
3
4 public class M {}
5
6 class P {
7     public class Q {}
8 }

```

Java

```

1 // artifact N.java
2 package x.y;
3
4 import x.y.P.Q;
5
6 public class N {}

```

Java

Listing 3.13: Java source artifacts M and N for Case 3.

The artifact M contains the class M and two more classes P and P.Q. Both artifacts M and N belong to package x.y. Compiling with

```

1 javac M.java
2 javac N.java

```

is not successful. The compiler complains that package x.y.P does not exist. It does not recognize that the type P exists and contains the inner type Q in the package x.y. It only looks at location x/y/P. However, again the internal buffer has precedence, because

```

1 javac M.java N.java

```

successfully compiles and produces four artifacts for M, N, P, and P.Q.

Again, Java is smart and efficient, but it considers each element of the path in an import statement, such as x.y.P, as directory as well as file artifact. The following setting in Listing 3.14 is very similar to Listing 3.13, but this time it stores the Java source artifacts in the appropriate directory and includes the inner type in the official type (identical to the filename).

```
1 // artifact p/y/R.java
2 package p.y;
3
4 public class R {
5     public class T {
6         public class U {}
7     }
8 }
```

```
1 // artifact p/y/S.java
2 package p.y;
3
4 import p.y.R.T.U;
5
6 public class S {}
```

Listing 3.14: Java source artifacts R and S for Case 3.

Because directory `p.y` exists, artifact `R.java` is found and identified as the next step towards finding the inner type `R.T.U`. Thus, calling the compiler with

```
1 javac p/y/S.java
```

leads to the compilation of both Java files, resulting in four class files `p/y/R$T$U`, `p/y/R$T`, `p/y/R`, and `p/y/S`.

We learn: Java tools and especially the compiler use the import statements to infer the artifacts and directories where the definitions of the types that are of interest are assumed. The import statement, however, combines a lookup of the artifact in the directory with a lookup for inner types within the artifact. Thus, when a type contains inner types, the type itself shares similarities to an artifact container.

### 3.3.4. Case 4: How Java Handles Archives

The fourth case focuses on the use of archives in the Java programming language. We reuse Java source artifact `B.java`, as defined in Listing 3.12. Additionally, we use several versions of libraries containing directories and versions of Java class files for class `A`, as defined in Listing 3.12. All artifacts are located in the same folder. Listing 3.15 shows the contents of the libraries.

The first two of the following three commands are successful, while the third fails:

```
1 javac -cp lib-classes.jar B.java
2 javac -cp lib-sources.jar B.java
3 javac -cp lib-broken.jar B.java # fails
```

```
1 :> jar -tf lib-sources.jar
2   |_ p/
3   |_ p/A.java
4
5 :> jar -tf lib-classes.jar
6   |_ p/
7   |_ p/A.class
8
9 :> jar -tf lib-broken.jar
10  |_ A.class
```

Listing 3.15: Content of the Java libraries.

While the first command produces artifact `B.class` only, the second command additionally produces `A.class`, because only the sources were in `lib-sources.jar`.

The failure of the third is due to the fact that in the `lib-broken.jar` the artifact for class `A` is stored in the wrong directory and thus is not found in package `p`. This indicates that the Java compiler looks only in desired directories within archives and does not search archives completely.

We learn: To use the Java compiler successfully, it has to be ensured that the directory structure as well as the naming of directories, types, and artifacts is correct within a `jar` file as well.

We have reflected these considerations in the AM using a number of constraints to ensure that packages, directories, types, and inner types are named correctly and consistently (cf. Line 15 of Listing 3.10).

### 3.3.5. Summary

In this section, a number of cases covering different aspects of the Java programming language were presented. Each of these cases focuses on different properties of the programming language and the behavior of the Java compiler.

The first case provided a simple yet informative example, which showed that Java compiles not only the artifacts given in the arguments, but also artifacts these rely on. This case can be seen as a strong hint that the `import` statement is regarded as a statement between artifacts by the Java compiler.

The second and third case demonstrated where the Java compiler looks for types. With the given examples it was shown that the Java compiler uses the `import` statements to infer the artifacts and directories in which it assumes the definitions of certain types.

The fourth case took a closer look at archives and showed how they are handled by the Java compiler. More specifically, the provided example demonstrated that the Java compiler assumes the package structure to match the internal directory structure of a library. This case also showed that this assumption by the default Java compiler is modeled in the

AM. This way artifact data of a given MDD project that conforms to the AM can be used to validate the project, or parts of it.

We also discussed that recompiling a previously modified artifact does only lead to a recompilation of the modified artifacts it directly relies on. However, when the modified artifact transitively relies on further modified artifacts, these artifacts are not recompiled, as shown in the example of Subsection 3.3.1. Therefore, it is easily possible that this can lead to errors. Thus, the Java compiler is not fully reliable for incremental compilation.

The case also demonstrated that the AM provides an useful way of giving an overview of the artifacts relying on each other. Such an overview can be used to evaluate the impact of any modification done to an artifact or for an efficient incremental compilation.

In conclusion, the different cases showed that our artifact centric approach to model Java-based development projects not only describes the behavior of the Java compiler, especially concerning the different relations on the artifact level, but also provides a good opportunity to evaluate and analyze the artifact structure of Java projects.

As a final remark, we repeat our assumption that there is only one outer type in each Java source file with the same simple name as the defining Java artifact. This is not enforced by the compiler, but many tools rely on it. The assumption ensures that the default class loading of the Java compiler works, since the logical structure matches the physical structure of these.

Nonetheless, we have demonstrated that there exist other special corner cases for which the logical and physical structure do not exactly match. Yet, it is still possible to successfully use the Java compiler if all the sources are explicitly given as parameters. In these cases the class loading step must work without file based search. It is, for example, possible to replace the default `ClassLoader`. However, for the AM this possibility is not taken into consideration, since we are convinced that for the majority of Java-based development projects replacing the default `ClassLoader` is not necessary.

## Chapter 4.

# Modelling Languages and Their Definitions

A modeling language, such as the UML/P defined in [Sch12, Rum16, Rum17], needs a proper definition that is useful for humans but also processable by computers. The extension of the AM defined up until now serves as a preparation for the definition of tools. It is again generic in the sense that it does not deal with a concrete language, but covers the range of language definitions.

On the other hand, we do not try to cover all potential language definitions, but concentrate on textual languages as they are defined by tools like MontiCore [GKR<sup>+</sup>06, KRV08, GKR<sup>+</sup>08, KRV10, Kra10, Völ11]. Because of the textual nature of these languages, we use grammars for language definition [KRV07a, KRV07b]. An alternative would be, for example, to use meta-models, like MOF [OMG16] or Ecore [SBMP08].

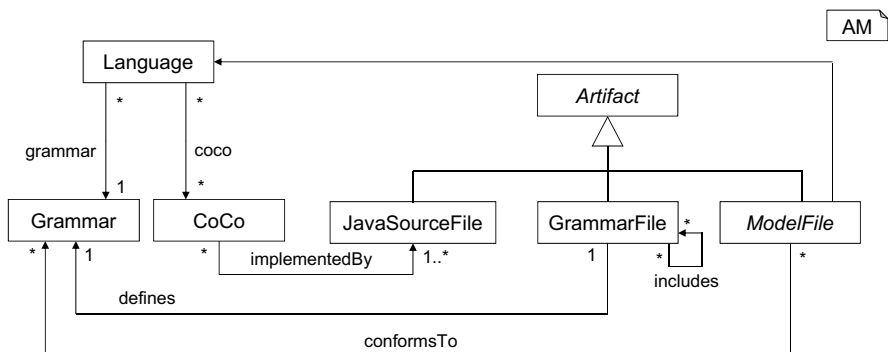


Figure 4.1.: A language is described by a grammar and several context conditions (CoCos), which describe the language’s well-formedness rules.

In Figure 4.1 we have defined the core concepts for modeling languages, including the specialized artifact kinds `JavaSourceFile`, `GrammarFile`, and `ModelFile`. Moreover, logical concepts reflecting the part of modeling languages that we are currently interested in are shown.



## 4.1. Languages

Let us first capture the essence of a language in terms of the abstract concepts `Language`, `Grammar`, and context condition (`CoCo`) in Listing 4.2. These are not directly reflected as artifacts, neither are they part of artifacts. Thus, they belong to the area of semantics definitions, which help us to structure and discuss concepts around the AM, but do not (necessarily) find a counterpart in any form of tools.

```

1  class Language {
2      String name;
3  }
4
5  class Grammar {}
6
7  class CoCo {
8      String name;
9  }
10
11 association grammar [*] Language -> Grammar [1];
12
13 association coco [*] Language -> CoCo [*];
14
15 context Language inv:
16     name != "";

```

Listing 4.2: AM: Language.

With the concept `Language` we describe any form of modeling, programming or other language that is of interest for us. Various forms of automata, statecharts, class diagrams, the full UML, etc. are candidates for a language.

In the pure textual setting that we have chosen, a language is defined by exactly one `Grammar` to capture the context-free part. To define the set of well-formed models in all the details, a set of context conditions (short `CoCos`) act as further constraints. Hence, for a model of a language to be well-formed none of these constraints must be violated.

Please again note that the `Language`, `Grammar`, and `CoCo` are not artifacts themselves, but abstract concepts that are defined using artifacts.

## 4.2. Grammar-Based Definitions

In MontiCore and other language work benches, such as Xtext [EB10], spoofax [KV10], or MPS [Voe13], it is possible to store a grammar used to describe a language in several artifacts. We reflect this by a separation between the actual grammar as a logical concept and the `GrammarFile` as a kind of reusable artifacts that contains a grammar, but also

includes other grammar definitions to complete the grammar. MontiCore, for example, allows to extend and replace nonterminals of a grammar as well as to compose or refine grammars. It offers the concept of a component grammar, which is basically dedicated for reuse.

```

1  class GrammarFile extends Artifact {}
2
3  association defines [1] GrammarFile -> Grammar [1];
4
5  association includes [*] GrammarFile -> GrammarFile [*];
6
7  context GrammarFile inv:
8      nameExtension == "mc4";
9
10 context GrammarFile inv:
11     refersTo.containsAll(includes);

```

Listing 4.3: AM: GrammarFile.

Listing 4.3 therefore introduces the `GrammarFile`, which defines a grammar while including a number of other grammars. This inclusion is again a form of reference and thus the association `includes` is a subset of `refersTo`. In MontiCore, grammar files have always the file extension `".mc4"`.

```

1  association implementedBy
2      [*] CoCo -> JavaSourceFile [1..*];

```

Listing 4.4: AM: CoCo.

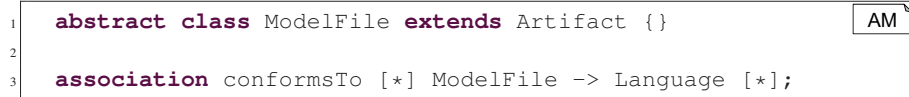
Context conditions (`CoCos`) describe well-formedness rules of a language. Without looking deeper into details, we very generally allow a context condition to be implemented by a set of Java classes. Association `implementedBy` in Listing 4.4 models this without further details.

### 4.3. ModelFiles Conform to Languages

Languages allow developers to write models. These models are stored in artifacts. We therefore introduce in Listing 4.5 the artifact of kind `ModelFile`, which conforms to a `Language`.

Of course, many model files conform to a language, but it may also be that a single model file conforms to several languages. This is in particular the case, if languages exist in different versions, such as Java 7, 8, etc. and their models do not explicitly contain

```
1  abstract class ModelFile extends Artifact {}
2
3  association conformsTo [*] ModelFile -> Language [*];
```



Listing 4.5: AM: ModelFile.

a version number. This is, for example, the case in programming languages, but not necessarily for XML dialects, where the XML document is forced to store its version and thus only belongs to one version of the language.

At this point, we could go much further into semantic details, for example, by describing models as a concept and relating them to a language as well as to the `ModelFiles`. In practice it may be that there is a complex relationship, because one model is decomposed and stored in several individual artifacts (`ModelFiles`). However, with this chapter we only intended to give a short introduction into the basic concepts of languages and how they can be modeled by an AM.

We could also take a deeper look into tooling issues, by investigating the parsing of `ModelFiles` and the creation of abstract syntax trees (ASTs). This would require to take a more detailed look at internals of artifacts, which is out of scope of this report and increasingly specific.

## Chapter 5.

# Class Diagrams in the Artifact Model

In Chapter 3, we have seen how to extend the AM to describe the typical artifacts of a programming language. However, the AM is not only designed for programming artifacts, but also for models written in specification and modeling languages. In Chapter 4, we prepared the meta-level describing how a modeling language can be defined. In this chapter, we demonstrate how to represent the set of models of a concrete language, namely class diagrams, in the AM.

UML/P Class diagrams (CD) [Rum16, Rum17] are structure diagrams, which can be utilized for various purposes. Common applications are conceptual modeling or the modeling of the system's design. In conceptual modeling [ET14] the main concepts of the problem domain are modeled as result of an analysis activity, whereas when used for the system design [Rum16], the structure of the system to be implemented is specified. Thus, classes in a CD can represent both, the main concepts of the problem domain as well as the technical classes of the target system.

In many MDD projects, CDs are used as conceptual models to describe the concepts of the problem domain and their relations [Rei16, Rot17, Loo17]. The files containing the model, namely CD model files, serve as input for the generator (cf. Subsection 2.2.1), which generates a data management system including a GUI and persistence functionality. Thus, CD model files can serve as source for the generation of the target product. In this section, only CD related artifacts, elements and corresponding relations are explained. An overview of these is given in Figure 5.1.

The relationship between source models and generated artifacts is investigated in Chapter 6.

A `CDModelFile` is a file artifact, which defines a CD in a textual notation [Sch12]. CD model files import other artifacts corresponding to their import statements. The import statement allows to use symbols defined in imported artifacts for the definition of the model under consideration. MontiCore provides a complex infrastructure to allow the import of symbols from foreign modeling languages [HLMSN<sup>+</sup>15, MSN17]. So the imported symbols are not necessarily defined in CD artifacts.

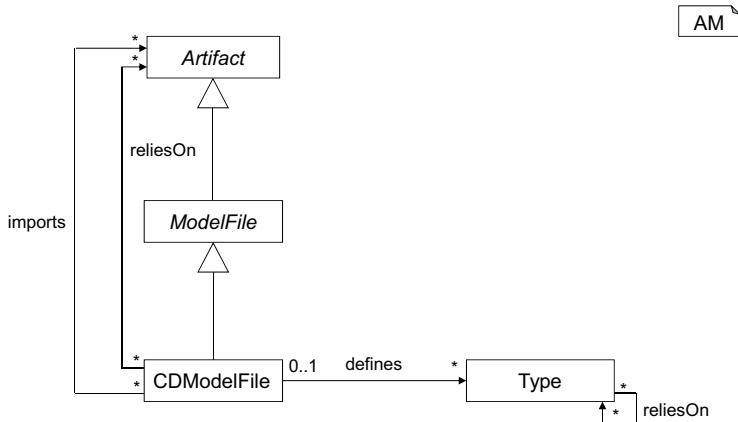


Figure 5.1.: Class diagram model files and their defined types.

We could look deeply into the syntactic structure of CD model files, exhibiting the concepts of classes, associations, etc. However, we do not look into the syntactic structure of CDs here, as the focus of the report lies on the artifact level. What we do, however, is to describe the semantics of a class diagram in terms of the types it introduces.

In this section, we only talk about the types that are introduced but not about the artifacts that implement these types. Probably, these artifacts will be Java artifacts generated from the class diagram or handcoded by a developer using the class diagram as source of information. Furthermore, the instances of class diagrams are located at the "object" level here, as we use the language of class diagrams to model the AM.

A CD model file defines types. The `Type` class subsumes any form of types from Java, CDs, etc. Its properties have already been defined in Listing 3.9.

As shown in Listing 5.2, several constraints are defined to ensure the consistency of the corresponding project data. The `imports` relation contributes to the `reliesOn` relation and the `reliesOn` relation is modeled as specializations of the `refersTo` relation in the AM, similar to the corresponding relations between Java artifacts (cf. Listings 3.2 and 3.3). Moreover, Lines 18 and 22 ensure that all types that are required in a CD model file are defined in the artifacts that the CD model file relies on.

Note that the last two OCL constraints of Listing 5.2 could be simplified as shown in Listing 5.3 if there was a `defines` relation from `Artifact` to `Type`.

This `defines` relation, however, is not present in the AM. The decision to leave out this relation was made since only a small number specialized artifact kinds in the AM, namely `CDModelFiles` and `JavaArtifacts`, actually define `Types`. Thus, the shown alternative cannot be chosen as valid specification of the AM, which is why the AM flag is not assigned to Listing 5.3.

The artifact model is defined in such a way that it is not possible for Java artifacts to rely on CD model files. However, this could, for example, be needed when handcoded

---

```

1  class CDModelFile extends ModelFile {}
2
3  association imports [*] CDModelFile -> Artifact [*];
4
5  association reliesOn [*] CDModelFile -> Artifact [*];
6
7  association defines [0..1] CDModelFile -> Type [*];
8
9  context CDModelFile inv:
10     nameExtension == "cd";
11
12 context CDModelFile inv:
13     reliesOn.containsAll(imports);
14
15 context CDModelFile inv:
16     refersTo.containsAll(reliesOn);
17
18 context CDModelFile m1, m2,
19     Type t1 in m1.defines, Type t2 in m2.defines inv:
20     t2 in t1.reliesOn implies m1 == m2 || m2 in m1.reliesOn;
21
22 context CDModelFile m, JavaArtifact a,
23     Type t1 in m.defines, Type t2 in a.defines inv:
24     t2 in t1.reliesOn implies a in m.reliesOn;

```

Listing 5.2: AM: CDModelFile.

```

1  context CDModelFile m inv:
2     m.reliesOn.defines.containsAll(m.definesType.reliesOn);

```

Listing 5.3: Simpler version of the OCL specification based on a defines relation from Artifacts.

Java classes cooperate with generated Java classes that implement types from a CD. On the model level, only the CDs and the handcoded Java classes are taken into consideration and thus have complex relationships, which we could also handle with the AM. As the `refersTo` relation is defined between `Artifacts` and we use a common `Type` element within the AM, we already prepared the addition of a `reliesOn` relation from `CDModelFile` to `JavaArtifact`.

This requires the presence of generators that transform CD model files into Java artifacts such as those described in [Rei16, Loo17, Rot17]. Here, the generated Java artifacts define the same types as the input CD model file.

Instead of defining this extension directly for class diagrams only, we define the general mechanism of generators in the following Chapter 6, allowing to relate source models and generated artifacts in general.

# Chapter 6.

## Generators and their Artifacts in MontiCore

In general, a generator reads one or more existing artifacts and produces a number of new artifacts. A code generator reads model artifacts as input and produces source code artifacts as output, thus transforming abstract models into executable code. The input models conform to well-defined modeling languages. In our AM, the target language, i.e., the language of the generated artifacts forming the product, is Java. Moreover, we now concentrate mainly on MontiCore-based generation techniques, even though many of the existing generator technologies use very similar approaches.

### 6.1. Static Artifact Structures

MontiCore-based generators make use of template-based code generation. The template engine used by MontiCore-based generators is FreeMarker<sup>1</sup>. FreeMarker is a flexible tool providing a powerful template language that, for example, allows templates to call other templates as well as to execute arbitrary Java expressions within the templates.

Thus, each generator consists, among others, of a set of FreeMarker templates and Java artifacts. Figure 6.1 shows the relevant elements and relations of the AM regarding generators.

Figure 6.1 mainly introduces relations between FreeMarker templates and Java artifacts. These relations occur as Java is not only used as target language for generation but also as language for the implementation of the generator itself. The details of those relations are given in Subsection 6.1.1.

All relations shown in Figure 6.1 can be inferred by a static analysis of the artifacts that are involved. These relations are therefore especially useful to identify the part of a development project that needs to be included in the generator, while unused templates and Java classes can be stripped.

The part of the AM presented in this chapter is meant for reuse, which is important as specific forms of MontiCore-based projects use different generators. Moreover, the AM

---

<sup>1</sup>[www.freemarker.org](http://www.freemarker.org)



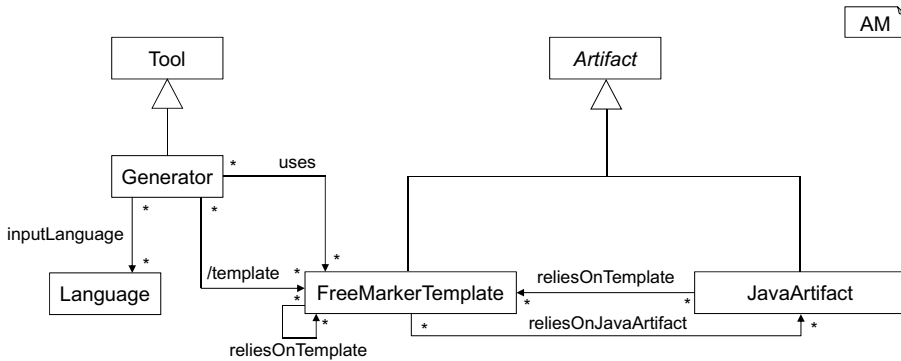


Figure 6.1.: Static structure of a generator as a tool that uses templates.

enables to describe projects using several generators and to model generators with several input languages. Nevertheless, it is restricted to FreeMarker-based template generation and Java as target language.

### 6.1.1. Templates

Before describing the generator as a whole, templates are introduced in detail in this section as they constitute an important part of the generator.

*“The basic idea behind Templated Generation is to write the output file you desire, inserting callouts for all the bits that vary. You then use a template processor with the template file and a context that can fill the callouts to populate the real output file.”* [Fow10]

Thus, the purpose of a template is to comfortably generate source code. The corresponding dynamic relation is introduced later on in Section 6.2 as the occurrence in a project can only be observed during the execution of the generator. As mentioned, in this section we concentrate on the relations, which can be statically determined in contrast to those observable during the execution of the generator. The specification of the `FreeMarkerTemplate` element of the AM is presented in Listing 6.2.

As mentioned, FreeMarker templates can invoke each other [Sch12]. A corresponding invocation statement induces a `reliesOnTemplate` relation between templates.

Templates can call methods from Java objects or create new objects. In this case the template relies on the Java artifact that defines the method or creation mechanism that is being used. This is indicated by the `reliesOnJavaArtifact` relation.

The `reliesOnTemplate` relation forms the counterpart to the aforementioned relation as it connects Java artifacts to FreeMarker templates. Java artifacts can rely on FreeMarker templates, for example, for the purpose of replacing default templates by more specific ones [Rot17]. Such a reference is implemented using the qualified template name in form of a string.

```

1  class FreeMarkerTemplate extends Artifact {}
2
3  association reliesOnTemplate
4      [*] FreeMarkerTemplate -> FreeMarkerTemplate [*];
5
6  association reliesOnJavaArtifact
7      [*] FreeMarkerTemplate -> JavaArtifact [*];
8
9  association reliesOnTemplate
10     [*] JavaArtifact -> FreeMarkerTemplate [*];
11
12 context FreeMarkerTemplate inv:
13     nameExtension == "ftl";
14
15 context FreeMarkerTemplate inv:
16     refersTo.containsAll(reliesOnTemplate);
17
18 context FreeMarkerTemplate inv:
19     refersTo.containsAll(reliesOnJavaArtifact);
20
21 context JavaArtifact inv:
22     refersTo.containsAll(reliesOnTemplate);

```

Listing 6.2: FreeMarkerTemplate and its associations.

The flexible management of templates, both within the template hierarchy and within the Java classes, contributes, among other techniques, to the configurability of the generator. In [GMR<sup>+</sup>16], a similar approach is presented, which uses a DSL instead of Java artifacts to define such replacements.

Whether the execution of such statements really occurs can only be determined at runtime of the generator. The corresponding part of the AM can be found in Section 6.2. It should also be noted that the dynamic relations of templates that call each other and the static knowledge about each other may differ in dynamic systems. The static knowledge is captured by the `reliesOnTemplate` association.

However, a static "type safe" analysis on the use of templates is not simple, because references to templates are based on strings that are potentially computed at generation time. We should not forbid this, but accept it as a well-known mechanism of parameterization and customization. Nonetheless, when template names are computed at generation time, then a static analysis of knowledge and usage becomes impossible.

In contrast, capturing corresponding calls through dynamic monitoring is easily doable. The generator, however, becomes more difficult to understand, when the strings referring to templates are manipulated, for example, by appending indexes or changing the path to the referenced template in the file system. In such cases, it is not possible to determine which

templates are actually in use by applying static analysis techniques. We thus suggest to avoid manipulating any string that refers to a template.

## 6.1.2. Generators

We already know that a generator is a system that consists of a set of artifacts. We refined this knowledge by connecting the generator with the templates it uses. Listing 6.3 shows the core elements of a generator.

```

1  class Generator extends Tool {}
2
3  association uses [*] Generator -> FreeMarkerTemplate [*];
4
5  association /template
6     [*] Generator -> FreeMarkerTemplate [*];
7
8  association inputLanguage [*] Generator -> Language [*];
9
10 context Generator g, JavaArtifact a in g.consistsOf inv:
11     g.template == g.uses.addAll(g.template.reliesOnTemplate**)
12         .addAll(a.reliesOnTemplate);
13
14 context Generator inv:
15     consistsOf.contains(template);
16
17 context Generator inv:
18     consistsOf.contains(template.reliesOnJavaArtifact);

```

Listing 6.3: AM: Generator.

The generator uses templates for the purpose of code generation depicted by the `uses` and the `template` relation. We define both relations because the `uses` relation gives us the set of directly known templates, whereas the `template` relation additionally contains the indirectly used templates. Indirectly used templates are templates that are included by other templates or templates on which Java artifacts of the generator rely. The `template` relation can therefore be used to identify the full set of necessary templates. The directly known templates can be identified through recognizing all names for templates that are provided as parameter when invoking the generator. In case of the MontiDEX generator [Rot17], the generator is invoked for each type defined in the CD. For each of the types *class*, *interface*, and *enum* a different template is used. Thus, the `uses` relation contains three templates in this case. Furthermore, only a smaller subset of the templates connected through the `template` relation may actually be called, because some may not be used in any execution.

We also specify the set of input languages of a generator, i.e., we define which kind of artifacts can be processed as the generator's input (cf. Section 6.2).

From Figure 2.1 we know that artifacts contribute to other artifacts when a generation process is executed. The AM provides a general relation called `produces` to capture this kind of contribution. Template and Java artifacts of a generator will contribute to the generated artifacts. We concentrate on static relations and thus cannot deduce a further refinement of the `produces` relation. For this, we use generator executions as described in Section 6.2.

## 6.2. Dynamic Monitoring of Tool Executions

The representation of dynamic processes, like a monitoring of a tool execution, is based on the notion of *actions*, which may be time-consuming. Therefore, the following subsection reflects our notion of monitoring, and the forthcoming sections refine actions and events for tool specific observations.

### 6.2.1. Representing Actions and Events

Figure 6.4 gives a general overview of how actions and their monitoring are modeled in the AM. The figure is detailed by Listing 6.5.

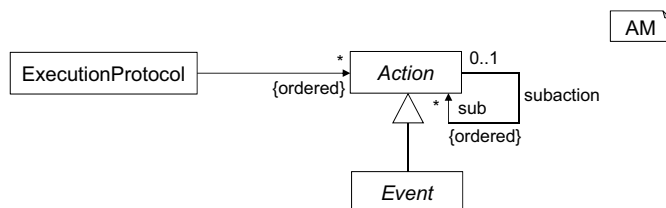


Figure 6.4.: Notion of action and their monitoring.

We generally add timestamps to *Actions*. An *Action* may have a duration and thus has a `start` and `end` time.

Line 23 ensures that timestamps are well defined in an *Action* and Line 26 states that a special form of *Action*, namely the *Event*, is instantaneous or just happens too fast for us to observe a duration.

An action may consist of subactions, which may be events as well. Thus, we model an *Action* hierarchy using a variant of the composite pattern with the `subaction` relation. This relation is ordered to reflect the order in which the actions occur. However, this needs to be compatible with the start and end dates of the actions. Line 29 ensures that subactions are actually contained in their parent's action and Line 32 ensures that two subsequent actions are in timing order.

```

1  class ExecutionProtocol {}
2
3  class Action {
4      Date start;
5      Date end;
6  }
7
8  class Event extends Action {}
9
10 association ExecutionProtocol -> Action [*] <<ordered>>;
11
12 association subaction
13     [0..1] Action -> (sub) Action [*] <<ordered>>;
14
15 context ExecutionProtocol p, int i,k inv:
16     (0 <= k && k <= p.action.size && 0 <= i && i <= k) implies
17     p.action[i].start <= p.action[k].start;
18
19 context ExecutionProtocol p inv:
20     p.action.nonEmpty implies (p.action.asSet ==
21     p.action[0].subaction**.asSet.add(action[0]));
22
23 context Action inv:
24     start <= end;
25
26 context Event inv:
27     start == end;
28
29 context Action a, Action s in a.subaction inv:
30     a.start <= s.start && s.end <= a.end;
31
32 context ExecutionProtocol p, int i,k inv:
33     (0 <= k && k <= p.action.size && 0 <= i && i <= k) implies
34     p.action[i].end <= p.action[k].start;

```

Listing 6.5: AM: Actions.

Please note that this form of action monitoring assumes that we have a sequential process for the generation of code. It does not reflect potential parallelization of independent generation steps, even though this might speed up the generation process.

An `ExecutionProtocol` consists of a list of actions. Because of Line 15, the order of the list is consistent with the order in which the actions start, and thus a flat list of all actions of an execution is available. This flat list can be derived from the hierarchy of actions by the constraint in Line 19. This constraint also enforces that the list additionally contains subactions of the starting action `action[0]`.

## 6.2.2. Actions in a Generation Process

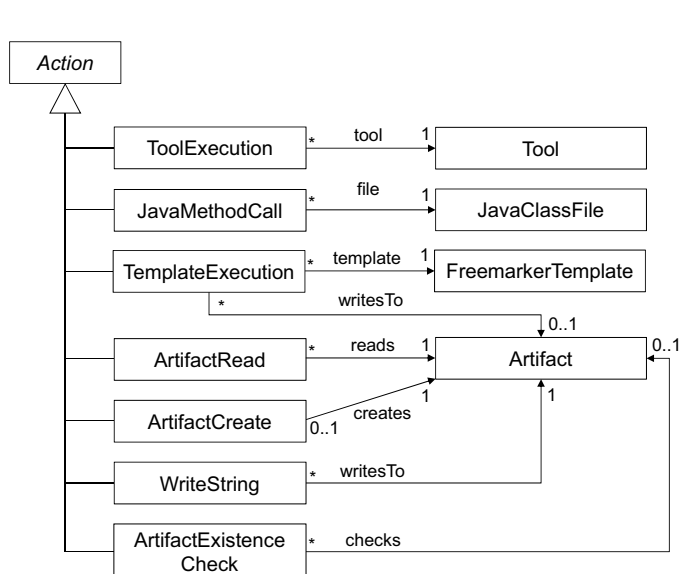


Figure 6.6.: Relevant actions in a Generation process.

Figure 6.6 and Listing 6.7 show the relevant kinds of actions that we want to observe when a generation process executes.

We are interested in observing all template executions, because we want to understand which templates are used and where they are contributing to. A template usually is copied completely into the artifact it writes to, thus a template execution is associated to an artifact. However, it may be that control templates do not directly write output into a file and subactions of a template, triggered, e.g., by related Java code, may contribute to further artifacts.

We are also interested in the artifacts created during the generation process and the artifacts that are read for that purpose. Some generators, especially MontiCore, are sensitive to the existence of certain hardcoded Files. To reflect this, we observe the check, whether an artifact exists.

We do not want to observe all method calls, but only the interesting ones, which includes method calls from templates to Java. A template may invoke Java methods to retrieve information. Additionally, it may invoke certain Java methods which then again start more generation processes. Furthermore, we subsume static methods, object creation, and attribute read and write as method calls.

In general, template executions and method calls may mutually and recursively call each other. In particular, it is also possible that a method call starts another template execution. We have reflected this in the action hierarchy in Listing 6.5. However, we allow

```

1  class ToolExecution extends Action {}
2  class TemplateExecution extends Action {}
3  class JavaMethodCall extends Action {}
4  class ArtifactRead extends Action {}
5  class ArtifactExistenceCheck extends Action {
6      String fullName;
7  }
8  class ArtifactCreate extends Action {}
9  class WriteString extends Action {
10     String content;
11 }
12
13 association tool [*] ToolExecution -> Tool [1];
14
15 association writesTo
16     [*] TemplateExecution -> Artifact [0..1];
17
18 association template
19     [*] TemplateExecution -> FreeMarkerTemplate [1];
20
21 association file [*] JavaMethodCall -> JavaClassFile [1];
22
23 association reads [*] ArtifactRead -> Artifact [1];
24
25 association checks
26     [*] ArtifactExistenceCheck -> Artifact [0..1];
27
28 association creates [0..1] ArtifactCreate -> Artifact [1];
29
30 association writesTo [*] WriteString -> Artifact [1];

```

Listing 6.7: AM: Actions.

`ArtifactRead` and `ArtifactCreate` actions to be time-consuming. In particular, it may be that an artifact creation uses a number of interesting and therefore observed methods calls.

Please note that there are a number of caveats and options: First, we might be interested in observing more forms of actions, such as the application of a transformation [Wei12, HRW15], creation of an abstract syntax tree or a symbol table [MSNRR16, MSN17], execution of a visitor [HMSNRW16], etc. Second, the temporal order (or even temporal containment) of two actions does not necessarily induce a causality. The actions could also be executed in a different order, because they are actually independent. Third, it may also be that an earlier action modifies the internal state (e.g. abstract syntax of the loaded models), such that a later action depends on the earlier one. In particular, it may be that all

artifacts read at a certain point in time, contribute to the artifacts created later on. Thus, observations on this level are potentially coarse-grained and could need further refinement.

We could add a number of additional constraints. These constraints, however, would not contribute to the overall specification. For example, a `JavaClassArtifact` can obviously not be called before it is created. It is most likely, that an artifact that is created during a `ToolExecution` will not be called later in the process, even though that is possible.

In the following, we derive some higher level relations but also add additional relations that allow us to understand in a more fine-grained way which methods and which templates contribute to which artifacts created.

### 6.2.3. Tools Read and Create Artifacts

One goal when monitoring the execution of a generator is to understand when to re-execute it (cf. Section 8.5). This is important if the goal is to use efficient and therefore incremental building processes. If such a process consists of several generation steps, where one generation step relies on the previous one, it is necessary to observe which artifacts have been used and which are produced. With our execution monitoring, we can derive this information and store it in the relations shown in Figure 6.8.

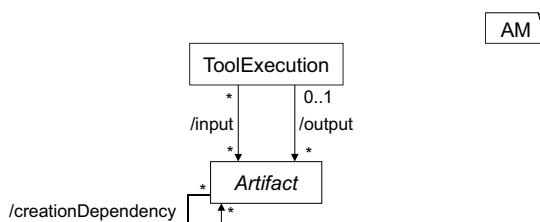


Figure 6.8.: Input and output dependencies of tool executions.

In Listing 6.9, we additionally describe the exact derivation.

Line 1 derives the total set of input artifacts that have been used by the tool execution. For this purpose all subactions that actually read an artifact are regarded.

Starting with Line 7, we derive the set of artifacts that is created during a tool execution.

Starting with Line 13, the `creationDependency` associations between artifacts is connected with the `output` relation. This relation is a conservative over approximation, which means that two `Artifacts` `M` and `N` can be related through this relation, although `M` has no real effect on `N`. Nonetheless, it is ensured that any building script that is based on artifact dependencies, such as *Make* (but not *Maven*), can utilize this to calculate necessary re-executions of generators.

Because of the conservative nature of `creationDependency`, it may be that the actual `produces` relation contains less links (see Line 18). For example, the Java



```

1  association /input [*] ToolExecution -> Artifact [*];
2
3  context ToolExecution inv:
4    input == {a in subaction** |
5              a in ArtifactRead || a in ArtifactExistenceCheck}.reads;
6
7  association /output [0..1] ToolExecution -> Artifact [*];
8
9  context ToolExecution inv:
10   output ==
11     {a in subaction** | a in ArtifactCreate}.creates;
12
13  association /creationDependency [*] Artifact -> Artifact [*];
14
15  context Artifact inv:
16   creationDependency == output.input;
17
18  context Artifact inv:
19   produces.containsAll(creationDependency);

```

Listing 6.9: Derivation of input and output.

compiler reads all source files and produces all class files in one execution. As a result, for Java files the `produces` relation forms a small subset of the `creationDependency` as we *know* what the tool does internally. However, as already discussed in Chapter 3, if only Java is involved the Java compiler manages its production dependencies itself.

### 6.2.4. Template and Java Files Contribute to Artifacts

When monitoring artifact dependencies using the `creationDependency` relation it is not possible to determine in a fine-grained form which artifacts of the generator tool contribute to which generated artifacts of the product. Therefore, we add the `contributesTo` relation described in Figure 6.10 and Listing 6.11 that stores more fine-grained contribution information.

Assuming that a generator is a `Tool` with a fixed set of `Artifacts` (see `consistsOf`), we can distinguish between variable input files and the fixed set of artifacts, both contributing to a generated artifact. Hence, the generator tool's internal contributions are monitored through the `contributesTo` relation.

This relation helps to understand which template or which specific Java class actually contributes to an artifact. This is especially interesting when the template can and should be modified to allow adaptation of the generation process. This way, the `contributesTo`

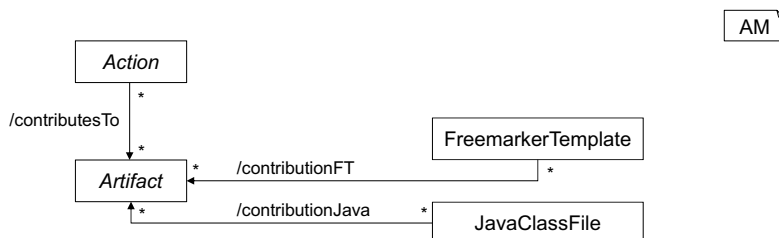


Figure 6.10.: Template and Java files contribute to artifacts.

```

1  association /contributesTo [*] Action -> Artifact [*];
2
3  context ToolExecution inv:
4    contributesTo == sub.contributesTo;
5
6  context ToolExecution inv:
7    contributesTo == output;
8
9  context TemplateExecution inv:
10   contributesTo == sub.contributesTo.add(writesTo);
11
12 context JavaMethodCall inv:
13   contributesTo == sub.contributesTo;
14
15 context ArtifactRead inv:
16   contributesTo == {};
17
18 context ArtifactExistenceCheck inv:
19   contributesTo == {};
20
21 context ArtifactCreate inv:
22   contributesTo == {creates};
23
24 context WriteString inv:
25   contributesTo == {writesTo};

```

Listing 6.11: Relevant action contribution associations.

relation determines which template to modify and what the impact of this modification will be.

In general, relation `contributesTo` must be derived by the concrete generator tool, e.g., through the protocol. This is the case for `TemplateExecution` and `JavaMethodCall`. For a detailed understanding of the actions, we also collect information on individual contributions through `write` actions. For other special forms of `Actions`,

the contribution can be defined as done by the constraints specified in Lines 3 to 24 pf Listing 6.12.

We assume that it is a minimally defined association in the sense that if an action is embedded in a super-action, the super-action must not know that it actually contributes to an artifact. However, we can derive the set of all artifact contributions by deriving the transitive closure of the subactions and their contributions.

Based on the `contributesTo` relation, we can derive which templates and which Java classes actually have a contributing impact on a generated artifact as shown in Listing 6.12.

```

1  association /contributionFT
2  [*] FreeMarkerTemplate -> Artifact [*];
3
4  context FreeMarkerTemplate inv:
5  contributionFT == templateExecution.contributesTo;
6
7  context FreeMarkerTemplate inv:
8  produces.containsAll(contributionFT);
9
10 association /contributionJava
11 [*] JavaClassFile -> Artifact [*];
12
13 context JavaClassFile inv:
14 contributionJava == javaMethodCall.contributesTo;
15
16 context JavaClassFile inv:
17 produces.containsAll(contributionJava);

```

Listing 6.12: Relevant artifact contribution associations.

In Lines 1 and 10, we define appropriate associations that capture this impact, and in Lines 4 and 13, we derive these relations from the monitoring relation `contributesTo`.

The relations `contributionFT` and `contributionJava` help to understand which artifacts of the generator need to be adapted or replaced to modify the content of a generated artifact. Furthermore, the relation `contributionFT` also helps to understand all potential side effects of template modification, which is in practice an underestimated problem.

Finally, both relations `contributionFT` and `contributionJava` are subsets of the `produce` relation (Line 7 and Line 16).

To demonstrate how these relations can be used to perform a variety of different analyses that are based on tool monitoring, Chapter 8 describes an exemplary application of the AM.

## Chapter 7.

# Artifacts in Maven-managed Java Projects

The configuration and execution of software development projects is usually done by build tools such as *Make*, *Ant*, *Maven*, or *Gradle*. This holds also for MDD Projects, where MDD tools have to be additionally executed. In this chapter we investigate on the example of Maven, “[...] a tool used to build deployable artifacts from source code.” [OMC<sup>+</sup>08], how the usage of build tools can be modeled by an AM. Figure 7.1 gives an overview over the involved artifacts, concepts and relations between them as they are modeled in our AM.

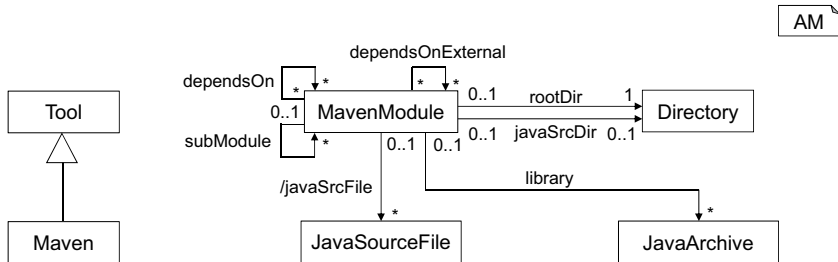


Figure 7.1.: Maven is a tool to build Maven modules, which have several directories with specific purposes. Maven modules make use of libraries, submodules, and modules they depend on.

Maven is a specialized form of `Tool` itself, which “[...] can now be used for building and managing any Java-based project.” [Mav17b]. Therefore, Maven modules containing Java source files are defined, which take Java archives into account when building the software. In addition several directories with special purposes are defined. In this section, we assume that Maven is used for building Java projects. Nevertheless, there are also approaches in which Maven is used to build software written in another programming language [NAR17].

```

1  class Maven extends Tool {}
2
3  class MavenModule {
4      boolean jarModule;
5      boolean project;
6  }
7
8  association rootDir [0..1] MavenModule -> Directory [1];
9
10 association javaSrcDir
11     [0..1] MavenModule -> Directory [0..1];
12
13 association /javaSrcFile
14     [0..1] MavenModule -> JavaSourceFile [*];
15
16 context MavenModule inv:
17     javaSrcDir.fullName ==
18         rootDir.fullName + "/src/main/java";
19
20 context MavenModule inv:
21     javaSrcFile ==
22         {JavaSourceFile f | f in javaSrcDir.contains**};
23
24 context MavenModule m, JavaSourceFile f in m.javaSrcFile inv:
25     f.parent != m.javaSrcDir implies
26         f.parent.fullName == m.javaSrcDir.fullName + "/" +
27             f.belongsTo.fullName.replaceAll(".", "/");
28
29 context MavenModule inv:
30     jarModule implies javaSrcDir.isPresent;

```

Listing 7.2: AM:Maven and MavenModule.

## 7.1. Maven Modules

Maven projects can either consist of a single module producing a single artifact or consist of several Maven modules organized in a module hierarchy. Listing 7.2 specifies the `MavenModule` concept of the AM that represents each of the modules. The root module of a module hierarchy is marked by the `project` flag. Only the leafs of the module hierarchy can contain source code. Other modules are used for the configuration of the build process. Leafs that contain source code files are marked by the `jarModule` flag, as they produces a jar file as output.

Each `MavenModule` has a `rootDir`, which is the directory that contains the overall content of the Maven module. Another directory with a special purpose regarding the

`MavenModule` is the `javaSrcDir`. This directory contains the Java source files of the Maven module to be considered when building the deployable artifact. In this directory only the source files of the target product are contained but not side products such as test cases. The set of all those Java source files can be derived, as realized by the `javaSrcFile` relation.

By default, the Java source directory of a Maven module is located in the subdirectory `"/src/main/java"` of the root directory (cf. Line 16). Moreover, it is ensured in Line 24 that each Java source file is located in the subdirectory of the Java source directory that corresponds to the file's package name.

## 7.2. Relations between Maven Modules

As shown in Listing 7.3, there are three kinds of relations between `MavenModules`. Multi-module projects consist of several modules organized in a module hierarchy. Submodules related by the `subModule` relation inherit some properties of their parent module.

The second relation is the `dependsOn` relation. Modules of the same module hierarchy can depend on each other, and thus artifacts defined in one Maven module can be used within another Maven module. Furthermore, Maven modules can depend on external Maven modules, which are modules that are part of another project.

Moreover, the `library` relation depicts that third party libraries, which are Java archives, can be used by Maven modules. These archives are not built by a module that is part of the same module hierarchy, but any form of artifacts, such as Java class files, templates, etc., within a library can be used by the artifacts of the module.

Furthermore, several interesting properties of Maven modules are expressed by the OCL constraints in Listing 7.3.

Line 18 defines that the root directory of each submodule is contained in the root directory of its parent module. This is a guideline enforced for MontiCore MDD Projects and the default configuration of Maven. Maven examines these directories and adds the submodules to the list of modules that are included in a build [OvZF<sup>+</sup>10]. Nevertheless, for other projects, this constraint could be relaxed, because the Maven default configuration can be adapted if desired.

Line 21 expresses that Maven modules can only depend on modules that are part of the same module hierarchy. Hence, dependent modules have the same root module marked by the `project` flag. Dependencies to other modules are comprised by the `dependsOnExternal` association.

Lines 25 to 35 describe that the `dependsOn`, the `dependsOnExternal`, and the `library` relations of a parent module are inherited from all submodules and that `dependsOn` relations cannot form a cycle.

```

1  association submodule
2      [0..1] MavenModule (parent) -> MavenModule [*];
3
4  association dependsOn [*] MavenModule -> MavenModule [*];
5
6  association dependsOnExternal
7      [*] MavenModule -> MavenModule [*];
8
9  association library [0..1] MavenModule -> JavaArchive [*];
10
11 context MavenModule inv:
12     (project <=> parent.isAbsent) &&
13     (!project implies {p in parent** | p.project}.size == 1);
14
15 context MavenModule inv:
16     jarModule implies submodule.isAbsent;
17
18 context MavenModule inv:
19     rootDir.contains(subModule.rootDir);
20
21 context MavenModule m1, MavenModule m2 in m1.dependsOn inv:
22     {m in m1.parent** | m.project} ==
23     {m in m2.parent** | m.project};
24
25 context MavenModule inv:
26     !project implies dependsOn.containsAll(parent.dependsOn);
27
28 context MavenModule inv:
29     !project implies dependsOnExternal
30     .containsAll(parent.dependsOnExternal);
31
32 context MavenModule inv:
33     !project implies library.containsAll(parent.library);
34
35 context MavenModule inv:
36     !(dependsOn**).contains(this);

```

Listing 7.3: AM: Relations between MavenModules.

### 7.3. Target Directories and Target Artifacts

During its execution Maven creates an output directory for each of its jar modules, where the produced output artifacts are stored. This relation is shown, among others, in Figure 7.4. Some of the shown elements are introduced later in Section 7.4.

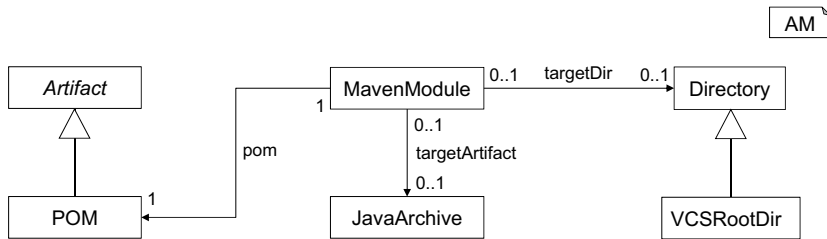


Figure 7.4.: Each Maven module is configured by its POM. The compiled Java class files of the Maven module are packaged to a target artifact contained in the target directory. The artifacts of Maven modules can be stored within a version control system (VCS).

One of these outputs is the `targetArtifact`, which is a deployable artifact packaged during the build process and is produced for each jar module. If there is only one jar module that is used to define the target product, the produced `targetArtifact` equals the target product. Nevertheless, in other cases the `targetArtifact` can either be part of the target product or form a library to be used in other projects. In the MDD domain, this target artifact can also be an executable generator, the run time environment needed to execute a generated product or even the run time environment needed to execute MDD tools as it is the case for MontiCore.

In addition to Figure 7.4, Listing 7.5 specifies how the target directory is named and where the target directory and the target artifact is located (cf. Line 6). Moreover, Line 13 describes that Java class files compiled from Java source files are located in the directory inside the target artifact, which corresponds to the package name of its source file.

The last constraint (Line 20) states, that target artifacts of dependent external modules are contained in the set of libraries that can be used for development.

## 7.4. POM and VCSRootDir

This section introduces further elements shown in Figure 7.4, which are detailed by the Listings 7.6 and 7.7.

As the sources of Maven modules are usually stored in a version control system (VCS) such as SVN [PCSF11] or git [CS14], a special kind of directory is introduced with `VCSRootDirectory`, which represents the working copy of the VCS content. Thus, this directory is a directory of the local file system. Moreover, Line 3 of Listing 7.6 describes that Maven root directories are contained in the directory hierarchy of exactly one `VCSRootDir`.

The configuration of Maven modules is contained in a special artifact, the Project Object Model (POM). Each Maven module has exactly one POM file. Within the POM artifact, the mentioned `subModules`, `dependendModules`, and `libraries` are defined. Moreover, bindings of tools can be defined within a POM file (see Section 7.5). Thus,



```

1  association targetDir [0..1] MavenModule -> Directory[0..1] AM
2
3  association targetArtifact
4    [0..1] MavenModule -> JavaArchive [0..1];
5
6  context MavenModule inv:
7    jarModule implies targetDir.isPresent &&
8                    targetArtifact.isPresent &&
9                    targetDir.parent == rootDir &&
10                   targetDir.simpleName == "target" &&
11                   targetArtifact.parent == targetDir;
12
13 context MavenModule m, JavaSourceFile f in javaSrcFile inv:
14   m.jarModule implies (f.compiledTo.parent.fullName ==
15   ((f.parent == m.javaSourceDir) ?
16   targetArtifact.fullName :
17   m.targetArtifact.fullName + "/" +
18   f.belongsTo.fullName.replaceAll(".", "/)));
19
20 context MavenModule inv:
21   library.containsAll(dependsOnExternal.targetArtifact);

```

Listing 7.5: AM: targetDir and targetArtifact relations.

```

1  class VCSRootDir extends Directory {} AM
2
3  context MavenModule inv:
4    {VCSRootDir d | d in (rootDir.parent**)}.size == 1;

```

Listing 7.6: AM: VCSRootDir.

POMs are the configuration points for the Maven build process. Furthermore, the necessary information to define a concrete project version, namely `groupId`, `artifactId`, and `version`, are specified within this artifact.

Listing 7.7 further specifies the name of POM artifacts in terms of their `simpleName` (cf. Line 9) and their `nameExtension` (cf. Line 12). This naming convention is used in MontiCore projects and thus the naming for POM files follows its default. Nevertheless, the naming convention is not enforced by Maven. To use a POM file with a different `simpleName`, Maven can be executed by the following command, where the file name is provided as argument [OMC<sup>+</sup>08]:

```

1  mvn -f <file>

```

```

1  class POM extends Artifact {
2      String groupId;
3      String artifactId;
4      String version;
5  }
6
7  association pom [1] MavenModule -> POM [1];
8
9  context POM inv:
10     simpleName == "pom";
11
12 context POM inv:
13     nameExtension == ".xml";
14
15 context MavenModule inv:
16     rootDir == pom.parent;

```

Listing 7.7: AM: POM.

However, the POM of each Maven module must be located in the module's root directory (Line 15).

## 7.5. Maven Phases

When Maven is executed, different `MavenPhases` are processed (see Section 7.6).



Figure 7.8.: Tools can be bound to Maven Modules via a Maven phase.

Each `MavenPhase` has a name and an `index`. Maven provides three lifecycles with a total of 30 Phases [Mav17a]. Here, we restrict the model to the seven main phases of the default lifecycle [Mav17a], which are *validate*, *compile*, *test*, *package*, *verify*, *install*, and *deploy*. The index of the first phase (*validate*) is 0 and the index of the last phase (*deploy*) is 6. The number of phases is restricted in the AM by Line 11 Listing 7.9.

When Maven is executed, different phases are processed (see Section 7.6). For each phase, several tools can be bound to a module. A bound tool will be executed when the phase to which the tool is bound is executed. Such bindings can be defined within the POM files. Bound tools of a parent module are inherited by all submodules (Line 14). "In Maven 2.0.5 and above, multiple goals bound to a phase are executed in the same order as they

```

1  class MavenPhase {
2      String name;
3      int index;
4  }
5
6  association executes
7      [*] MavenModule -> MavenPhase [*] <<ordered>>;
8
9  association binds [*] MavenPhase -> Tool [*] <<ordered>>;
10
11 context MavenModule inv :
12     executes.size == 7;
13
14 context MavenModule inv:
15     executes.binds.containsAll(parent.executes.binds);

```

Listing 7.9: AM: MavenPhase.

are declared in the POM [...]” [Mav17a]. Thus, the relations `executes` and `binds` are ordered.

## 7.6. Executing Maven

This section examines how Maven processes its phases in terms of its actions. Therefore, three additional `Action`s are introduced in Figure 7.10. The fourth action, `ToolExecution`, has already been introduced in Subsection 6.2.2.

The action `MavenExecution` represents the execution of the overall Maven build process, which requires the Maven module serving as root module for the build and a specific Maven phase provided as parameter. In contrast, `MavenModuleExecution` represents the part of the process building a single `MavenModule`, which is performed by several consecutive `MavenPhaseExecutions` in the AM. Listing 7.11 gives further details on the modeling of the Maven build process execution.

The `subActions` of each Maven-related `Action` kind are all of the same specialized action kind, as defined in Lines 18 to 24. Further properties of Maven related actions are specified by Listing 7.12.

The Maven related actions contribute to the artifacts their subactions contribute to, which is defined by Lines 1 to 7 of Listing 7.12. Moreover, Line 10 states that Maven always builds the module that is provided as root of the build first. Line 13 states that Maven also builds all transitively reachable submodules. All phases are executed for each module up to the phase provided as a parameter to a single Maven build (Line 16). The tools, which are executed during a single phase execution, are exactly those bound to the corresponding `MavenModule` of that phase (Line 21). Finally, a target artifact is

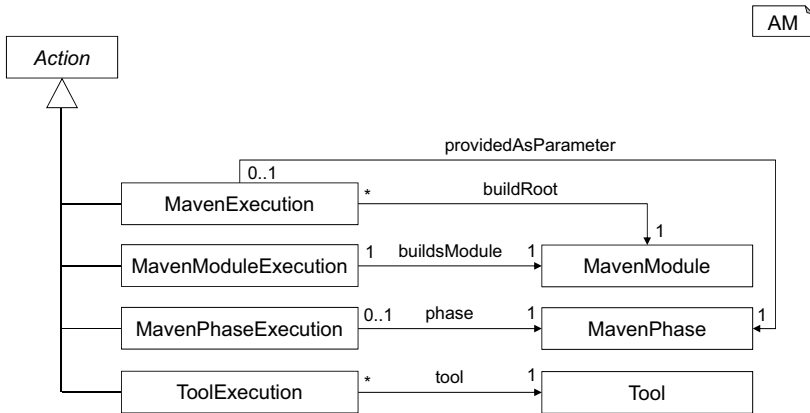


Figure 7.10.: Maven executions.

available after a successful Maven build, when one of its phase executions executed the *package* phase (Line 24).

It might seem that the `MavenExecution` class is not necessary, as we do not have a corresponding structural element in the AM. Moreover, allowing `MavenModuleExecutions` as subactions of `MavenModuleExecutions` and determining the start and end time of a single execution by looking at the subactions seems natural. We did not model the AM in this way as (1) each module needs time to build its own artifact, and (2) not all submodules of a parent module must be completed before building another module. An example for (2) is given in Figure 7.13, which is marked with the `ArtifactData` flag indicating that the shown object diagram conforms to the AM. In the example, the order of executions would be:  $R \rightarrow B \rightarrow B2 \rightarrow A \rightarrow B1$ . Thus, module *A* "interrupts" the building of the complete module hierarchy of module *B* consisting of *B*, *B1* and *B2*.

```
1  class MavenExecution extends Action {}
2
3  association buildRoot [*] MavenExecution -> MavenModule [1];
4
5  association providedAsParameter
6    [0..1] MavenExecution -> MavenPhase [1];
7
8  class MavenModuleExecution extends Action {}
9
10 association buildsModule
11   [1] MavenModuleExecution -> MavenModule [1];
12
13 class MavenPhaseExecution extends Action {}
14
15 association phase
16   [0..1] MavenPhaseExecution -> MavenPhase [1];
17
18 context MavenExecution inv:
19   sub in MavenModuleExecution;
20
21 context MavenModuleExecution inv:
22   sub in MavenPhaseExecution;
23
24 context MavenPhaseExecution inv:
25   sub in ToolExecution;
```

AM

Listing 7.11: AM: MavenExecution.

```

1  context MavenExecution inv:
2      contributesTo == sub.contributesTo;
3
4  context MavenModuleExecution inv:
5      contributesTo == sub.contributesTo;
6
7  context MavenPhaseExecution inv:
8      contributesTo == sub.contributesTo;
9
10 context MavenExecution inv:
11     buildRoot.mavenModuleExecution == sub[0];
12
13 context MavenExecution inv:
14     sub.containsAll(buildRoot.subModule**);
15
16 context MavenExecution me,
17     MavenModuleExecution mme in me.sub,
18     MavenPhaseExecution mpe in mme.sub inv:
19     me.providedAsParameter.index >= mpe.phase.index;
20
21 context MavenModuleExecution inv:
22     sub.containsAll(phase.binds.toolExecution);
23
24 context MavenModuleExecution me,
25     MavenPhaseExecution pe in me.sub inv:
26     pe.phase.name == "package" implies
27     me.buildsModule.targetArtifact.isPresent;

```

AM

Listing 7.12: Further properties regarding the execution of a Maven build process.

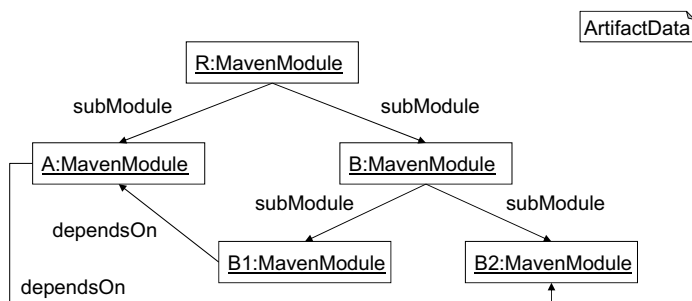


Figure 7.13.: Example of a Maven module structure.



## Chapter 8.

# Applications of the Artifact Model

With the last chapter we finished the introduction of our AM consisting of its core and a number of project specific extensions, for example, for Java and for MDD concepts. Based on our AM, we are now able to analyze concrete projects or add additional constraints that we expect the projects to fulfill. The actual number of these kinds of analyses is unlimited. In this chapter, we therefore demonstrate the usefulness of the AM based on a selected set of examples. Besides the ones presented in this chapter, further examples would be:

- Understand the execution dependencies in a Maven project and reduce the unnecessary re-execution of process steps. In the case of MDD projects, this would lead to a much more efficient generation and deployment process.
- Understand the complex multi-level generation process, where some artifacts are generated and then used in a generator to generate the next level of artifacts. Maven currently handles this by separating each generation level in its own module, as the compile phase is only executed once during a module execution. In this case, Maven re-processes all artifacts belonging to the same module, even though, the re-processing of only a few artifacts per module might be necessary. This considerably reduces efficiency and thus agility, especially when several levels of generation are modified at the same time.
- Understand the complexity of relationships as potential sources of errors, e.g., for review or automated testing.
- Understand the flow of symbols imported and exported from several artifacts, potentially modified or mapped into other kinds of symbols, which is heavily necessary in heterogeneous modeling language environments, such as assisted by MontiCore.
- Understand the impact of potential changes in some artifacts on dependent artifacts, which is usually referred to as *Change Impact Analysis* [Arn96]. Using the AM, such an analysis can be performed considering different kinds of artifacts and relations.
- Understand which artifacts have been generated and which are source artifacts. This can be especially useful to understand if exactly the source artifacts are version controlled.



- Understand which present artifacts do not contribute to the result. Examples for such artifacts are generated artifacts that are never used and are not part of any shipped product or archive. This indicates that the generator produces irrelevant artifacts. Another example for such artifacts are model artifacts that are not used as input within the generation process and are thus potentially irrelevant.

## 8.1. Analyses based on Tool Monitoring

In this section, we demonstrate how to define additional associations that describe certain forms of information derived from the AM. These analyses are mainly based on the model part defined in Subsection 6.2.1. Listing 8.1 contains these descriptions in form of associations and their derivations as OCL constraints.

The analyses are based on a number of execution protocols (*execs*). Line 7 describes which of the *tools* have never been used. Line 13 describes which of the available FreeMarker templates are not syntactically known to the tools and thus are not usable for generation. Those templates are candidates for removal. Line 25 describes which of the available FreeMarker templates are not used. They might be used in a different context but for the monitored generation processes they are not. It may be that an erroneous template did not executed those templates accidentally. Line 30 examines whether there are templates that have been used, but don't have any contribution to the resulting artifacts.

## 8.2. Understanding the Module/Artifact Architecture

Modules describe the architectural substructures of the systems that are present in a project. Modules can be defined for the target product as well as potential tools and generators in model-driven development projects. By explicitly modeling such modules and assigning artifacts to modules, stakeholders are able to get a view on the overall project from a more abstract perspective. Modules might be identified by packages or directories, but neither of these are actually sufficient to describe what the modules in a concrete project are. By defining modules explicitly, we are able to focus only on specific parts of the overall project, and different architectures, each with its own focus, can be defined for the same project. In [Lil16], for example, the following architectures are regarded:

- Technical architecture consisting of technical modules such as client, logic, and persistence.
- Domain architecture consisting of modules reflecting elements or processes of the target domain.
- Pattern architecture putting together artifacts with the same role within the project, such as all artifacts fulfilling the same role of, e.g., the model-view-controller pattern [Fow02].

```

1  class Analysis {}
2
3  association execs Analysis -> ExecutionProtocol [*];
4
5  association /unusedTools Analysis -> Tool [*];
6
7  context Analysis inv:
8      unusedTools == Tool.removeAll(
9          {te.tool | ToolExecution te in execs.action});
10
11 association /unknownTP Analysis -> FreeMarkerTemplate [*];
12
13 context Analysis inv:
14     unknownTP ==
15         {FreeMarkerTemplate t | !t in Generator.template};
16
17 association /usedTP Analysis -> FreeMarkerTemplate [*];
18
19 association /unusedTP Analysis -> FreeMarkerTemplate [*];
20
21 context Analysis inv:
22     usedTP ==
23         {te.template | TemplateExecution te in execs.action};
24
25 context Analysis inv:
26     unusedTP == FreeMarkerTemplate.removeAll(usedTP);
27
28 association /noContrib Analysis -> FreeMarkerTemplate [*];
29
30 context Analysis inv:
31     noContrib ==
32         {Template t in usedTP | contributionFT.isEmpty};

```

Listing 8.1: Defining analyses based on the AM.

Moreover, there can be different kinds of modules in an architecture description such as components, layers, interfaces, etc. In the AM, we concentrate on a single substructure to define the architecture, which can be hierarchically composed: `Module`. To maintain the required flexibility when defining architectures, the assignment from artifacts to modules must be done manually. With the artifacts and their relations at hand, the `reliesOnModule` relation between modules can be derived as shown in Listing 8.2. The shallower this relation, the better the degree of modularization in the project is.

To calculate the `reliesOnModule` relation between modules, three other assisting derived associations are defined. First, modules are structured hierarchically, i.e., each module may contain a set of submodules. Moreover, each module contains a set of artifacts

```

1  association /containedModule [*] Module -> Module [*];
2
3  context Module inv:
4      containedModule == {this}.addAll(subModule**);
5
6  association /containedArtifact [*] Module -> Artifact [*];
7
8  context Module inv:
9      containedArtifact == containedModule.artifact;
10
11 association /externalArtifact [*] Module -> Artifact [*];
12
13 context Module inv:
14     externalArtifact ==
15         containedArtifact.reliesOn.removeAll(containedArtifact);
16
17 association /reliesOnModule [*] Module -> Module [*];
18
19 context Module inv:
20     reliesOnModule == externalArtifact.module;

```

Listing 8.2: Application of AM: Actual architecture.

that consists of the artifacts contained directly by the module itself and those contained by its submodules. Then, for each module hierarchy a set of external artifacts exists consisting of all the artifacts on which the artifacts of the whole module hierarchy rely and which are not contained in the module hierarchy themselves. With those relations at hand, the mentioned `reliesOnModule` relation between modules can be derived. Here, all other modules that are no submodules of the given module and which contain artifacts the module or its submodules rely on are related. This relation finally allows us to inspect the actual architecture of the project in terms of modules and relations between them.

As we choose to manually assign artifacts to modules, the concept `Module` cannot be derived from the underlying set of artifacts automatically. In our view, the modeled architecture should be well aligned with the actual structure of the artifacts. This way, stakeholders are enabled to orient themselves easily in the sources of the projects, as they can find and follow the architectural decomposition they already have in mind. Even when the actual architecture follows the desired architecture, a gap between the organization of artifacts and the module organization can exist. This gap should be as small as possible. Therefore, the analyses shown in Listing 8.3 are useful.

First, the relationship between `Directory` and `Module` is pretty interesting. In many projects, modules are organized in such a way that artifacts belonging to one module (and its submodules) are stored in one directory (and its subdirectories), while packages are distributed over both, modules and directories. Whether the modules and directories are aligned well can be assessed by looking at the `moduleDirectory` relation. It relates

```

1  association /moduleDirectory [*] Module <-> Directory [*];
2
3  context Module m, Directory d in m.containedArtifact inv:
4      m.moduleDirectory ==
5          {Directory sub | sub in d.contains**}.add(d);
6
7  association /modulePackage [*] Module <-> Package [*];
8
9  context Module m, JavaArtifact j in m.containedArtifact inv:
10     m.modulePackage == j.package;
11
12 association /packageDirectory [1] Package <-> Directory [*];
13
14 context Package inv:
15     javaSourceFile.parent in Directory implies
16     packageDirectory == javaSourceFile.parent;

```

Listing 8.3: Application of AM: Align modules to the physical structure.

all directories assigned to the module and its submodules as well as all subdirectories of those to the module (cf. Line 3).

Another interesting question is how `Modules` and `Packages` are related. This can be examined through the `modulePackage` relation illustrating, how many packages belong to a module and also in how many modules a package participates. It is an interesting question whether this should be a shallow relation, potentially even a 1 – 1 relation. All packages of contained Java artifacts are constituted in this relation (cf. Line 9).

Lastly, also the `Directory` and `Package` concepts are related, but not identical, because it is not required to locate all artifacts of one package in the same directory (and also not in the same archive). Although it is not mandatory, but ensured by the constraints of the AM, to locate an artifact of a certain package in a directory that reflects the package name. Nevertheless, packages can be related to several directories and thus the relation `directoryPackage` also gives interesting hints about the structure of the project. All directories containing the artifacts belonging to a given package are related to that package (cf. Line 14). Note that Java source files can also be located in archives without having a parent directory. Hence, to restrict the `directoryPackage` relation to directories an implication is used.

## 8.3. Generated Systems

In a model-driven development project at least one system is (partly) generated: the target product. To get an overview of the generators directly participating in the generation of different systems, a relation between `Generators` and `Systems` can be calculated as shown in Listing 8.4

```

1  association /generatedSystem [*] Generator -> System [*];
2
3  context Generator inv:
4    generatedSystem ==
5    {System s | s.consistsOf(toolExecution.contributesTo)};

```

Listing 8.4: Application of AM: Generated systems.

Again, a derived association is introduced representing the result of this analysis: `generatedSystem`. A generator is linked to a system if the generator participated in the system's creation, i.e., the generator generated at least one artifact of each linked system. As tools and generators are systems themselves, this analysis also covers cases where a generator generates another tool participating in the creation of the target system. An example for such a multi-step build process is given in [AHRW17], where transformation tools are generated based on transformation models. Then, these tools are used to perform model-to-model transformations as a preparation step before code generation.

## 8.4. Template Relations Induced by Generated Artifacts

When template-based generation is used, the following problem occurs quite often: large parts of the templates are written in the target language and only small parts are written in the template language itself. Thus, templates often do not rely on each other directly, but the artifacts generated by the templates do. In these cases, changing only one of the templates may result in the generation of corrupted target code by other templates. Therefore, it is useful to know this kind of indirect relation between templates, which can be calculated as shown in Listing 8.5.

```

1  association /contributionsRelyOn
2    [*] FreeMarkerTemplate -> FreeMarkerTemplate [*];
3
4  context FreeMarkerTemplate inv:
5    contributionsRelyOn == contributionFT.contributionFT
6    .addAll(contributionFT.refersTo.contributionFT);

```

Listing 8.5: Application of AM: Indirect template relations.

By adding the association `contributionsRelyOn` between `FreeMarkerTemplates`, the results of this analysis are available when needed. Two template artifacts are linked if either both templates contributed to the generation of the same artifact or the generated source files of these templates rely on each other (e.g., through an import). Note that it also relates each template to itself.

The presented analysis can give a hint whether a change of a template might induce changes of other templates. There are two threats: (1) As the analysis depends on one execution of the tool chain and the available generated code depends on the input used in one execution, it cannot be guaranteed that the relation is complete for all possible inputs. (2) For both cases of the analysis, it may be that the change of a related template is not really necessary. This depends on the kind of change, i.e., if the generated code of one template interacts somehow with the generated code of another template. A more detailed analysis would be beneficial, but requires a more detailed extraction of relations between generated code parts. As the focus of the AM lies on relations between artifacts, the hint that a change may be necessary is sufficient.

## 8.5. Incremental Toolchain Execution

Executing a tool chain should be as fast as possible. Otherwise developers can loose their development flow [Csi96], which results in a lack of productivity. One effective way to reduce the execution time of the tool chain is to only execute the relevant parts of the tool chain affected by the last changes. A flag indicating that a `ToolExecution` should be repeated can be calculated as shown in Listing 8.6.

```

1  abstract class Artifact {
2      //...
3      boolean removed;
4      boolean isHandcoded;
5      /boolean toBeRecreated;
6  }
7
8  class ToolExecution extends Action {
9      /boolean toRepeat;
10 }
11
12 context Artifact inv:
13     isHandcoded implies !toBeRecreated;

```

Listing 8.6: Application of AM: Incremental toolchain execution.

Listing 8.6 introduces some "runtime" extensions to the AM necessary to determine the part of the tool chain that needs to be executed again. While the structure of the AM is in principle static, this information can change during tool executions and development activities.

First, with the Boolean flag `removed`, we indicate that an artifact was deleted, which can be observed, for example, by monitoring the underlying file system. Artifacts with flag `removed=true` are thus only virtually existing (in the monitoring logs), but there are no real artifacts corresponding anymore. Together with the `modified` date attribute (cf.

Listing 2.2), we model modifications and deletions of files that happen when developers change models. This includes the creation of a new artifact, which can also be detected by its modification date.

Another attribute describes whether the artifact is subject to be generated or whether it is a handcoded artifact (`isHandcoded`). This information is usually retrieved from build files such as makefiles or Maven POMs. Finally, the `toBeRecreated` flag indicates whether some action on the artifact is necessary.

The `ExecutionProtocol` contains a list of actions, including `ToolExecutions`, that all have start and end dates. Furthermore, we know for each `ToolExecution`, which artifacts it used as input and produced as output. However, we have already combined this information into the `creationDependency` association between artifacts directly. This allows us to describe the `toBeRecreated` value, without explicit naming of the tools necessary in Listing 8.7.

```
1  context Artifact inv:
2    toBeRecreated == !isHandcoded && (
3      removed ||
4      lastChange < max(createDependency.lastChange) ||
5      creationDependency.toBeRecreated ||
6      output.output.toBeRecreated);
```

Listing 8.7: Application of AM: Artifacts to be recreated.

A file is subject to re-creation when it was generated and then has been removed (Line 3), or one of the modification dates of one dependent artifacts is newer (Line 4), or one of the dependent artifacts themselves are marked for re-creation (Line 5).

Unfortunately, when executing a tool, it usually re-creates or at least touches all its outputs, not only those that are necessary. This is some kind of "collateral damage" that could be optimized if tools were more fine-grained. To capture this, the last part of the specification (Line 6) ensures that re-creation is also propagated downward the tool chain for files that are re-generated as collateral damage.

This is not a complete definition but a recursive characterization. Indeed, setting all `toBeRecreated` flags of all generated files to true would fulfill the specification. However, we are interested in the minimal solution, where as many `toBeRecreated` flags as possible remain false. This form of calculation is done, e.g., by Make but unfortunately not by Maven.

Finally, Listing 8.8 describes which tool executions need to be repeated.

```
1  context ToolExecution e, Artifact a in e.output inv:
2    a.toBeRecreated implies e.toRepeat;
```

Listing 8.8: Application of AM: Tool executions to be repeated.

## 8.6. Unused Imports

Code smells generally reduce the readability of sources and increase maintainability costs. One specific kind of code smell is the unused import. For Java, modern IDEs can remove unused imports automatically. Based on the AM, different relations indicating that unused imports for corresponding artifact kinds (including Java artifacts) exist, which can be calculated as shown in Listing 8.9.

```

1  association /typeUsage [*] JavaArtifact -> JavaArtifact [*];
2
3  association /unusedImports
4    [*] JavaSourceFile -> JavaArtifact [*];
5
6  context JavaSourceFile inv:
7    typeUsage == defines.reliesOn.javaArtifact.remove(this);
8
9  context JavaSourceFile inv:
10   unusedImports == imports.removeAll(typeUsage);
11
12 association /typeUsage [*] CDModelFile -> Artifact [*];
13
14 association /unusedImports [*] CDModelFile -> Artifact [*];
15
16 context CDModelFile inv:
17   typeUsage == defines.reliesOn.javaArtifact
18     .addAll(defines.reliesOn.cdModelFile).remove(this);
19
20 context CDModelFile inv:
21   unusedImports == imports.removeAll(typeUsage);

```

Listing 8.9: Application of AM: Unused imports.

Unused imported artifacts are those artifacts that are imported, but whose types are not required by the importing artifact. For Java, only Java source files can import other artifacts. Thus, the relation `unusedImports` (Line 3) from Java source files to Java artifacts is specified. Unused Java imports can be derived from the set of imported Java artifacts by removing Java artifacts whose types do not rely on each other. Similar to the relation for Java, the relation `unusedImports` (Line 14) is specified containing imported but unused artifacts for CD model files. As introduced in Chapter 5, CD Model files can import several kinds of artifacts. Those kinds are not further constrained, which is why the target of this association is `Artifact`. Nevertheless, the derivation of the unused imports for CD model files relies on the concrete kinds of artifacts that define relevant types for CD model file (see Line 16).





# Chapter 9.

## Conclusion

We conclude this report by summing up the considerations on the presented version of the artifact model, which are followed by multi-level considerations on CD4A and OCL based on our experiences.

### 9.1. Considerations on the Artifact Model

In this report, we presented the first version of our artifact model for generator-based model-driven projects with precise definitions for the core elements and relations relevant for software development projects and some extensions useful for model based projects that use generators.

For a precise specification, we used the CD4A dialect [Rot17] describing class diagrams and the OCL/P constraint language [Rum16] to capture relations in all necessary details. The main advantage of these notations is that they can be directly used for tool development, e.g., for an analysis of concrete projects with, e.g., derivatives of the Data Explorer [MSNRR15, Rot17].

The AM was developed in an extensible way. Building subclasses and refining associations allows to adapt the meta-classes defined in the AM. Several extensions of the later chapters have demonstrated how to do this.

We assume that our AM can not only be used as foundation for tooling but also helps to clarify terminology. For example the term *artifact* has now a clear definition. The term *dependency* is also used very often, but different stakeholders have a very different understanding of what a dependency is. Already at the early definitions, we have clarified that artifacts can either "depend" on each other, because they *refer to* each other, e.g., through import statements and thus are used together, or because a generator retrieves information from one artifact to *produce* another one, and then only the newly produced artifact is of further use.

The presented model is actually much more than only an artifact model. It contains abstract, organizational concepts, such as *modules*, but also partially internals of the artifacts, retrieving, e.g., imports and fully qualified use of external artifacts. It also describes a part of the type systems whose types are typically passed around between artifacts.

The AM contains static relations that can be inferred through artifact analysis but also dynamic relations that need to be monitored when a build process is executed. The monitoring protocol gives useful hints on which of the static relations are actually in use at construction time and which may be used. We might speak of *construction smells*, for example, when static knowledge between artifacts prevents a decoupling of the development, modularization, or leads to a re-execution of generation processes even if unnecessary.

With the AM and corresponding tooling, we can also capture the overall picture of the situation in an MDD project. This helps to improve the communication between all involved stakeholders and constitutes a substantial step towards an integrated, model-based analysis of MDD projects.

These contributions will serve as a starting point for further research. We are interested in exploiting the big picture of development projects. The AM and corresponding data will serve as a basis for the improvement of software development projects in general and MDD projects in particular, such as the optimization and correction of MDD processes, incremental code generation, and ensuring consistency between the actual state of the project and its desired architecture. Moreover, further evaluations and improvements of the approach are planned in software development projects including in particular Cyber Physical System developments projects [Lee06, Lee08, HRR12, KRS12, RRW14, KRRvW17, RRSW17], where mechanical, electrical, and software parts are modeled together using various modeling languages. The same holds for many other domains, such as flight control [ZPK<sup>+</sup>11], cloud computing [NPR13, HHK<sup>+</sup>14, KRR14, HHK<sup>+</sup>15] building information systems [FLP<sup>+</sup>11, FPPR12, KPR12], robotics [RRW12, RRW13, THR<sup>+</sup>13, RRRW15], or automotive development [BBR07, GHK<sup>+</sup>08, GKPR08, BR12b, BR12a].

## 9.2. Multi-level considerations on CD4A and OCL

In Section 1.1 we introduced the specification language CD4A [Rot17] in combination with a Java-like version of the OCL [Rum16]. Throughout this report, we used this combination consequently to describe the structure and partly structural manifestation of execution behavior (we can call that protocol of action sequences) of a model-based development project.

The effort undertaken was partly due to the available tooling infrastructure that enables to directly generate a monitoring system based on this specification language. But to some extent, we also wanted to understand how well a textual notation for class diagrams and for additional constraints works.

The reader will probably concur that the illustrating graphical descriptions of the class diagram gave a better overview of the structure than the textual representation could do. Unfortunately, OCL constraints cannot easily be visualized but to a large extent need to be textual. We would also like to point the reader to the idea of integrating object diagrams for specific situations into the OCL, as discussed in [Rum17].

Other alternatives would be to use a graphical notation only for class diagrams or to use a combination between graphical visualization and a textual master artifact, quite like we did in this report. However, this form of tooling is still to be improved and not part of the consideration here.

Both CD4A and the OCL can be improved by variety of operators to better cope with multi-level modeling. It could be interesting, for example, to introduce several levels of classes that can be instantiated into other forms of classes, similar to [Atk97, AK01, CHE05, AGK09, dLG10, Fra16].

As an alternative, we could remain in a single level structure, but allow at least better assistance for singletons, and especially for refinement of complete structures including several classes and associations. In particular, we had to describe two associations and specify with OCL that one is a subset of the other, instead of directly refining the association.

Although we presented CD4A in modular chunks, the whole class diagram is actually one flat model. To refine structures, it would be helpful to modularize class diagrams and provide explicit operators for their composition and refinement. Composition of class diagrams means that the diagrams are merged on elements (especially classes and associations) with the same name. Refinement of a class diagram means that existing classes and associations are individually refined by adding attributes, specializing multiplicities, and similar operations.

Finally, the OCL also could provide more elaborate functions, for example, for a reflective, transitive closure, handling of sequences in time, or string manipulation for directory and package names.



# Bibliography

- [AGK09] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering*, 35(6):742–755, 2009.
- [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.
- [AK01] Colin Atkinson and Thomas Kühne. The Essence of Multilevel Metamodeling. In *International Conference on the Unified Modeling Language*, pages 19–33. Springer, 2001.
- [ALB<sup>+</sup>14] Hamoud Aljamaan, Timothy C. Lethbridge, Omar Badreddin, Geoffrey Guest, and Andrew Forward. Specifying Trace Directives for UML Attributes and State Machines. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 79–86, Jan 2014.
- [Arn96] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
- [Atk97] C. Atkinson. Meta-modelling for Distributed Object Environments. In *Proceedings First International Enterprise Distributed Object Computing Workshop*, pages 90–101, Oct 1997.
- [BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.
- [BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. Jordi Cabot, Richard Paige, and Alfonso Pierantonio, editors, Part of the *Grand Challenges in Modeling (GRAND'17) Workshop*. <http://www.edusymp.org/Grand2017/>, 2017.

- [BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.
- [BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.
- [BSS14] María Cecilia Bastarrica, Jocelyn Simmonds, and Luis Silvestre. Using Megamodeling to Improve Industrial Adoption of Complex MDE Solutions. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, pages 31–36. ACM, 2014.
- [CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, Anaheim, California, USA, 2003.
- [CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software process: improvement and practice*, 10(2):143–169, 2005.
- [CHGZ12] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman. *Software and Systems Traceability*. Springer, 2012.
- [CS14] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2014.
- [Csi96] Mihaly Csikszentmihalyi. *Creativity: Flow and the Psychology of Discovery and Invention*. HarperCollins, 1996.
- [Die12] Jens Dietrich. Upload your Program, Share your Model. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 21–22. ACM, 2012.
- [dLG10] Juan de Lara and Esther Guerra. *Deep Meta-modelling with MetaDepth*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [DRDRIP14] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Dealing with the coupled evolution of metamodels and model-to-text transformations. In *Proceedings of the Workshop on Models and Evolution (ME)*, 2014.

- [DRELHE15] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E. Lopez-Herrejon, and Alexander Egyed. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software*, 2015.
- [DS10] Anna Derezinska and Marian Szczykalski. Tracing of state machine execution in the model-driven development framework. In *Proceedings of 2nd International Conference on Information Technology (ICIT)*, pages 109–112, June 2010.
- [EB10] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '10*, pages 307–309, New York, NY, USA, 2010. ACM.
- [Ecl15] Eclipse Website <http://www.eclipse.org>, visited: 16.11.2015.
- [ET14] David W. Embley and Bernhard Thalheim. *Handbook of Conceptual Modeling*. Springer, 2014.
- [FLP<sup>+</sup>11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.
- [FLV12] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. *Modeling the Linguistic Architecture of Software Products*. Springer, 2012.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
- [Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley, 1st edition, 2010.
- [FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.
- [Fra16] Ulrich Frank. Designing Models and Systems to Support IT Management: A Case for Multilevel Modeling. In *3rd International Workshop on Multi-Level Modelling (MULTI 2016)*, pages 3–24, 2016.



- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [GHK<sup>+</sup>08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.
- [GHK<sup>+</sup>15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.
- [GHK<sup>+</sup>15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development, Communications in Computer and Information Science 580*, pages 112–132. Springer, 2015.
- [GJS<sup>+</sup>15] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle America, Java SE 8 edition, 2015.
- [GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.
- [GKR<sup>+</sup>06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.
- [GKR<sup>+</sup>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.

- [GMR<sup>+</sup>16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modelierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.
- [HHK<sup>+</sup>14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.
- [HHK<sup>+</sup>15] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.
- [HKR<sup>+</sup>11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.
- [HLMSN<sup>+</sup>15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.
- [HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.
- [HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.
- [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.
- [HSG12] Regina Hebig, Andreas Seibel, and Holger Giese. On the Unification of Megamodels. *Electronic Communications of the EASST*, 42, 2012.

- [KEK<sup>+</sup>15] Angelika Kusel, Jürgen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Wieland Schwinger, and Johannes Schönböck. Consistent Co-Evolution of Models and Transformations. In *Proceedings of MOD-ELS*, 2015.
- [KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.
- [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 *Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017*, pages 34–50. Springer International Publishing, 2017.
- [KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.
- [KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.
- [KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.
- [KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.
- [KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on*

- 
- Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.
- [KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.
- [KV10] Lennart C.L. Kats and Eelco Visser. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 444–463, New York, NY, USA, 2010. ACM.
- [Lee06] Edward A Lee. Cyber-Physical Systems - Are Computing Foundations Adequate? In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, 2006.
- [Lee08] Edward A Lee. Cyber Physical Systems : Design Challenges. *Electrical Engineering*, 2008.
- [Lil16] Carola Lilienthal. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. dpunkt, 2016.
- [LM12] David Lo and Shahar Maoz. Scenario-Based and Value-Based Specification Mining: Better Together. *Automated Software Engineering*, 19(4):423–458, 2012.
- [Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE*. Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, 2017.
- [LvdA15] Maikel Leemans and Wil M. P. van der Aalst. Process Mining in Software Systems. In *Proceedings of MODELS*, 2015.
- [LYBB15] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Oracle America, Java SE 8 edition, 2015.
- [Mav17a] Maven - Introduction to the Build Lifecycle <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>, visited: 12.06.2017.
- [Mav17b] Maven - Introduction <https://maven.apache.org/what-is-maven.html>, visited: 01.05.2017.

- [MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, 2017.
- [MSNRR15] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.
- [MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.
- [NAR17] NAR Plugin Website <http://maven-nar.github.io/index.html>, visited: 01.05.2017.
- [NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.
- [NW15] nWire Software Ltd nWire <http://www.nwiresoftware.com>, visited: 16.11.2015.
- [OMC<sup>+</sup>08] Tim O'Brien, Manfred Moser, John Casey, Brian Fox, Jason Van Zyl, Eric Redmond, and Larry Shatzer. *Maven: The Complete Reference*, 2008.
- [OMG14] Object Management Group. Object Constraint Language, February 2014. <http://www.omg.org/spec/OCL/2.4>.
- [OMG16] Object Management Group. OMG Meta Object Facility (MOF) Core Specification, November 2016. <http://www.omg.org/spec/MOF/2.5.1/PDF/>.
- [Or16] The Java Tutorials - What is a Package? <http://docs.oracle.com/javase/tutorial/java/concepts/package.html>, visited: 19.05.2016.
- [OvZF<sup>+</sup>10] T O'Brien, J van Zyl, B Fox, J Casey, J Xu, and T Locher. *Maven: By example. An introduction to Apache Maven*, 2010.
- [PCSF11] C Michael Pilato, Ben Collins-Sussman, and Brian Fitzpatrick. *Version Control with Subversion*. "O'Reilly Media, Inc.", 2011.

- [PKB13] Leo Pruijt, Christian Koppe, and Sjaak Brinkkemper. Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support. In *Proceedings of 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 220–229. IEEE, 2013.
- [PW15] Leo Pruijt and Jan Martijn EM van der Werf. Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, page 56. ACM, 2015.
- [Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.
- [Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data-Centric Applications with MontiDEX*. Shaker Verlag, 2017.
- [RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.
- [RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, 2017.
- [RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Kozirolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.
- [RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.
- [RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

## BIBLIOGRAPHY

---

- [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.
- [SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.
- [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.
- [SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using Dependency Models to Manage Complex Software Architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, pages 167–176, New York, NY, USA, 2005. ACM.
- [SMS15] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make Reference Manual*. Samurai Media Limited, 2015.
- [Son15] hello2morrow Sonargraph <https://www.hello2morrow.com/products/sonargraph>, visited: 16.11.2015.
- [SPBS15] Jocelyn Simmonds, Daniel Perovich, Maria Cecilia Bastarrica, and Luis Silvestre. A Megamodel for Software Process Line Modeling and Evolution. In *Proceedings of MODELS*, 2015.
- [Sta15] Odysseus Software GmbH stan4j <http://stan4j.com/>, visited: 16.11.2015.
- [Str15] Structure101 Software Architecture Environment (ADE) <http://www.structure101.com>, visited: 16.11.2015.
- [THR<sup>+</sup>13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.
- [VJBB13] Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière. Typing artifacts in megamodeling. *Software & Systems Modeling*, 12(1):105–119, 2013.

- [VKB13] Paola Vallejo, Mickaël Kerboeuf, and Jean-Philippe Babau. Specification of a Legacy Tool by Means of a Dependency Graph to Improve its Reusability. In *Proceedings of the Workshop on Models and Evolution (ME)*, pages 80–87. Citeseer, 2013.
- [Voe13] Markus Voelter. *Language and IDE Modularization and Composition with MPS*", pages 383–430. Springer, 2013.
- [Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.
- [Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.
- [Wen14] Sven Wenzel. Unique identification of elements in evolving software models. *Software & Systems Modeling*, 13(2):679–711, 2014.
- [WL08] Richard Wettel and Michele Lanza. CodeCity: 3D Visualization of Large-Scale Software. In *Companion of the 30th international conference on Software engineering*, pages 921–922. ACM, 2008.
- [ZPK<sup>+</sup>11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.





# Appendix A.

## Merged Artifact Model

```
1 classdiagram ArtifactModel {
2   // _____
3   // Artifact
4   // _____
5
6   abstract class Artifact {
7     String simpleName;
8     String nameExtension;
9     Date modified;
10    /String name;
11    /String fullName;
12    /boolean isRoot;
13  }
14
15  context Artifact inv:
16    name == ((nameExtension == "") ? simpleName :
17      simpleName + "." + nameExtension);
18
19  association /refersTo [*] Artifact -> Artifact [*];
20
21  association /produces [*] Artifact -> Artifact [*];
22
23  context Artifact a inv:
24    !a.produces**.contains(a);
25
26  // _____
27  // ArtifactContainer
28  // _____
29
30  abstract class ArtifactContainer extends Artifact {}
31
32  composition contains
33    [0..1] ArtifactContainer (parent) -> Artifact [*];
```

```

34
35 context Artifact inv:
36     isRoot <=> parent.isAbsent &&
37     !isRoot implies {p in parent** | p.isRoot}.size == 1;
38
39 context Artifact inv:
40     fullName == (isRoot ? "/" :
41                 (parent.isRoot ? "/" + name :
42                 parent.fullName + "/" + name));
43
44 context Artifact a, Artifact b inv:
45     a.name == b.name && a.parent == b.parent
46     implies a == b;
47
48 // _____
49 // Directory
50 // _____
51
52 class Directory extends ArtifactContainer {}
53
54 context Directory inv:
55     nameExtension == "";
56
57 context Artifact inv:
58     isRoot implies (this in Directory);
59
60 context Artifact inv:
61     isRoot <=> simpleName == "/";
62
63 context Artifact inv:
64     name.contains("/") implies isRoot;
65
66 // _____
67 // Archive
68 // _____
69
70 class Archive extends ArtifactContainer {}
71
72 // _____
73 // JavaArtifacts
74 // _____
75
76 abstract class JavaArtifact extends Artifact {}
77

```

```

78  association reliesOnJavaArtifact
79      [*] JavaArtifact -> JavaArtifact [*];
80
81  context JavaArtifact inv:
82      refersTo.containsAll(reliesOnJavaArtifact);
83
84  // _____
85  // JavaSourceFile
86  // _____
87
88  class JavaSourceFile extends JavaArtifact {}
89
90  association imports [*] JavaSourceFile -> JavaArtifact [*];
91
92  association compiledTo
93      [1] JavaSourceFile -> JavaClassFile [*];
94
95  context JavaSourceFile inv:
96      reliesOnJavaArtifact.containsAll(imports);
97
98  context JavaSourceFile inv:
99      produces.containsAll(compiledTo);
100
101  context JavaSourceFile inv:
102      nameExtension == "java";
103
104  // _____
105  // JavaClassFile
106  // _____
107
108  class JavaClassFile extends JavaArtifact {}
109
110  context JavaClassFile inv:
111      nameExtension == "class";
112
113  // _____
114  // JavaArchive
115  // _____
116
117  class JavaArchive extends Archive {}
118
119  context JavaArchive inv:
120      nameExtension == "jar";
121

```

```

122 // _____
123 // Package
124 // _____
125
126 class Package {
127     String name;
128     String fullName;
129     /boolean isRoot;
130 }
131
132 association belongsTo [*] JavaArtifact -> Package [1];
133
134 composition subPackage
135     [0..1] Package (parent) -> (child) Package [*];
136
137 context Package inv:
138     !name.contains(".");
139
140 context Package inv:
141     isRoot <=> parent.isAbsent &&
142     !isRoot implies {p in parent** | p.isRoot}.size == 1;
143
144 context Package inv:
145     !isRoot implies
146     fullName == parent.fullName + "." + name;
147
148 context Package a, Package b inv:
149     a.parent == b.parent && a.name == b.name implies a == b;
150
151 // _____
152 // Type
153 // _____
154
155 class Type {
156     String simpleName;
157     /String name;
158     /String fullName;
159     /boolean isInnerType;
160 }
161
162 association defines [*] JavaArtifact -> Type [*];
163
164 composition contains [1] Package -> Type [*];
165

```

```

166 composition
167   [0..1] Type (containingType) -> (innerType) Type [*];
168
169 association reliesOn [*] Type -> Type [*];
170
171 context Type inv:
172   isInnerType <=> !containingType.isAbsent &&
173   isInnerType implies
174     {t in containingType** | t.isInnerType}.size == 1;
175
176 context Type inv:
177   name == (isInnerType ?
178     containingType.name + "." + simpleName : simpleName);
179
180 context Type inv:
181   fullName == (package.isRoot ? name :
182     package.fullName + "." + simpleName);
183
184 // _____
185 // Relation between Java Artifacts and Java Types
186 // _____
187
188 context JavaSourceFile inv:
189   defines == compiledTo.defines;
190
191 context JavaSourceFile inv:
192   {t in defines | !t.isInnerType}.size == 1;
193
194 context JavaArtifact a1, a2,
195   Type t1 in a1.defines, Type t2 in a2.defines inv:
196   t1.reliesOn.contains(t2) implies
197     a1 == a2 || a1.reliesOn.contains(a2);
198
199 context JavaArtifact a, Type t in a.defines inv:
200   !t.isInnerType implies t.simpleName == a.simpleName;
201
202 context JavaArtifact inv:
203   parent.fullName.replaceAll("/", ".")
204     .endsWith(belongsTo.fullName);
205
206 context JavaSourceFile inv:
207   forall n in { t.fullName.replace(".", "/") |
208     t in defines && !t.isInnerType }:
209     fullName.endsWith(n + ".java");

```

```
210
211 context JavaClassFile inv:
212     parent.fullName.replaceAll("/", ".")
213         .endsWith(defines.package.fullName);
214
215 context JavaClassFile inv:
216     simpleName == defines.name.replaceAll(".", "$");
217
218 //_____
219 // Language
220 //_____
221
222 class Language {
223     String name;
224 }
225
226 class Grammar {}
227
228 class CoCo {
229     String name;
230 }
231
232 association grammar [*] Language -> Grammar [1];
233
234 association coco [*] Language -> CoCo [*];
235
236 context Language inv:
237     name != "";
238
239 //_____
240 // GrammarFile
241 //_____
242
243 class GrammarFile extends Artifact {}
244
245 association defines [1] GrammarFile -> Grammar [1];
246
247 association includes [*] GrammarFile -> GrammarFile [*];
248
249 context GrammarFile inv:
250     nameExtension == "mc4";
251
252 context GrammarFile inv:
253     refersTo.containsAll(includes);
```

```

254
255 // _____
256 // CoCo
257 // _____
258
259 association implementedBy
260     [*] CoCo -> JavaSourceFile [1..*];
261
262 // _____
263 // ModelFile
264 // _____
265
266 abstract class ModelFile extends Artifact {}
267
268 association conformsTo [*] ModelFile -> Language [*];
269
270 // _____
271 // CDModelFile
272 // _____
273
274 class CDModelFile extends ModelFile {}
275
276 association imports [*] CDModelFile -> Artifact [*];
277
278 association reliesOn [*] CDModelFile -> Artifact [*];
279
280 association defines [0..1] CDModelFile -> Type [*];
281
282 context CDModelFile inv:
283     nameExtension == "cd";
284
285 context CDModelFile inv:
286     reliesOn.containsAll(imports);
287
288 context CDModelFile inv:
289     refersTo.containsAll(reliesOn);
290
291 context CDModelFile m1, m2,
292     Type t1 in m1.defines, Type t2 in m2.defines inv:
293     t2 in t1.reliesOn implies m1 == m2 || m2 in m1.reliesOn;
294
295 context CDModelFile m, JavaArtifact a,
296     Type t1 in m.defines, Type t2 in a.defines inv:
297     t2 in t1.reliesOn implies a in m.reliesOn;

```



```
298
299 // _____
300 // System
301 // _____
302
303 abstract class System {
304     String name;
305     String version;
306 }
307
308 association consistsOf [*] System -> Artifact [*];
309
310 // _____
311 // Product
312 // _____
313
314 class Product extends System {}
315
316 // _____
317 // Tool
318 // _____
319
320 class Tool extends System {}
321
322 // _____
323 // Module
324 // _____
325
326 class Module {
327     String name;
328 }
329
330 association module [*] System -> Module [*];
331
332 association subModule [*] Module -> Module [*];
333
334 association artifact [*] Module -> Artifact [*];
335
336 // _____
337 // FreeMarkerTemplate
338 // _____
339
340 class FreeMarkerTemplate extends Artifact {}
341
```

```

342 association reliesOnTemplate
343     [*] FreeMarkerTemplate -> FreeMarkerTemplate [*];
344
345 association reliesOnJavaArtifact
346     [*] FreeMarkerTemplate -> JavaArtifact [*];
347
348 association reliesOnTemplate
349     [*] JavaArtifact -> FreeMarkerTemplate [*];
350
351 context FreeMarkerTemplate inv:
352     nameExtension == "ftl";
353
354 context FreeMarkerTemplate inv:
355     refersTo.containsAll(reliesOnTemplate);
356
357 context FreeMarkerTemplate inv:
358     refersTo.containsAll(reliesOnJavaArtifact);
359
360 context JavaArtifact inv:
361     refersTo.containsAll(reliesOnTemplate);
362
363 // _____
364 // Generator
365 // _____
366
367 class Generator extends Tool {}
368
369 association uses [*] Generator -> FreeMarkerTemplate [*];
370
371 association /template
372     [*] Generator -> FreeMarkerTemplate [*];
373
374 association inputLanguage [*] Generator -> Language [*];
375
376 context Generator g, JavaArtifact a in g.consistsOf inv:
377     g.template == g.uses.addAll(g.template.reliesOnTemplate**)
378         .addAll(a.reliesOnTemplate);
379
380 context Generator inv:
381     consistsOf.contains(template);
382
383 context Generator inv:
384     consistsOf.contains(template.reliesOnJavaArtifact);
385

```

```

386 // _____
387 // Action
388 // _____
389
390 class ExecutionProtocol {}
391
392 class Action {
393     Date start;
394     Date end;
395 }
396
397 class Event extends Action {}
398
399 association ExecutionProtocol -> Action [*] <<ordered>>;
400
401 association subaction
402     [0..1] Action -> (sub) Action [*] <<ordered>>;
403
404 context ExecutionProtocol p, int i,k inv:
405     (0 <= k && k <= p.action.size && 0 <= i && i <= k) implies
406     p.action[i].start <= p.action[k].start;
407
408 context ExecutionProtocol p inv:
409     p.action.nonEmpty implies (p.action.asSet ==
410     p.action[0].subaction**.asSet.add(action[0]));
411
412 context Action inv:
413     start <= end;
414
415 context Event inv:
416     start == end;
417
418 context Action a, Action s in a.subaction inv:
419     a.start <= s.start && s.end <= a.end;
420
421 context ExecutionProtocol p, int i,k inv:
422     (0 <= k && k <= p.action.size && 0 <= i && i <= k) implies
423     p.action[i].end <= p.action[k].start;
424
425 // _____
426 // ToolActions
427 // _____
428
429 class ToolExecution extends Action {}

```

```

430 class TemplateExecution extends Action {}
431 class JavaMethodCall extends Action {}
432 class ArtifactRead extends Action {}
433 class ArtifactExistenceCheck extends Action {
434     String fullName;
435 }
436 class ArtifactCreate extends Action {}
437 class WriteString extends Action {
438     String content;
439 }
440
441 association tool [*] ToolExecution -> Tool [1];
442
443 association writesTo
444     [*] TemplateExecution -> Artifact [0..1];
445
446 association template
447     [*] TemplateExecution -> FreeMarkerTemplate [1];
448
449 association file [*] JavaMethodCall -> JavaClassFile [1];
450
451 association reads [*] ArtifactRead -> Artifact [1];
452
453 association checks
454     [*] ArtifactExistenceCheck -> Artifact [0..1];
455
456 association creates [0..1] ArtifactCreate -> Artifact [1];
457
458 association writesTo [*] WriteString -> Artifact [1];
459
460 // _____
461 // ToolExecution
462 // _____
463
464 association /input [*] ToolExecution -> Artifact [*];
465
466 context ToolExecution inv:
467     input == {a in subaction** |
468         a in ArtifactRead || a in ArtifactExistenceCheck}.reads;
469
470 association /output [0..1] ToolExecution -> Artifact [*];
471
472 context ToolExecution inv:
473     output ==

```

```
474     {a in subaction** | a in ArtifactCreate}.creates;
475
476 association /creationDependency [*] Artifact -> Artifact [*];
477
478 context Artifact inv:
479     creationDependency == output.input;
480
481 context Artifact inv:
482     produces.containsAll(creationDependency);
483
484 // _____
485 // Action and Artifact Contribution
486 // _____
487
488 association /contributesTo [*] Action -> Artifact [*];
489
490 context ToolExecution inv:
491     contributesTo == sub.contributesTo;
492
493 context ToolExecution inv:
494     contributesTo == output;
495
496 context TemplateExecution inv:
497     contributesTo == sub.contributesTo.add(writesTo);
498
499 context JavaMethodCall inv:
500     contributesTo == sub.contributesTo;
501
502 context ArtifactRead inv:
503     contributesTo == {};
504
505 context ArtifactExistenceCheck inv:
506     contributesTo == {};
507
508 context ArtifactCreate inv:
509     contributesTo == {creates};
510
511 context WriteString inv:
512     contributesTo == {writesTo};
513
514 association /contributionFT
515     [*] FreeMarkerTemplate -> Artifact [*];
516
517 context FreeMarkerTemplate inv:
```

```

518     contributionFT == templateExecution.contributesTo;
519
520 context FreeMarkerTemplate inv:
521     produces.containsAll(contributionFT);
522
523 association /contributionJava
524     [*] JavaClassFile -> Artifact [*];
525
526 context JavaClassFile inv:
527     contributionJava == javaMethodCall.contributesTo;
528
529 context JavaClassFile inv:
530     produces.containsAll(contributionJava);
531
532 // _____
533 // Maven
534 // _____
535
536 class Maven extends Tool {}
537
538 // _____
539 // Maven Module
540 // _____
541
542 class MavenModule {
543     boolean jarModule;
544     boolean project;
545 }
546
547 association rootDir [0..1] MavenModule -> Directory [1];
548
549 association javaSrcDir
550     [0..1] MavenModule -> Directory [0..1];
551
552 association /javaSrcFile
553     [0..1] MavenModule -> JavaSourceFile [*];
554
555 context MavenModule inv:
556     javaSrcDir.fullName ==
557         rootDir.fullName + "/src/main/java";
558
559 context MavenModule inv:
560     javaSrcFile ==
561         {JavaSourceFile f | f in javaSrcDir.contains**};

```

```
562
563 context MavenModule m, JavaSourceFile f in m.javaSrcFile inv:
564     f.parent != m.javaSrcDir implies
565         f.parent.fullName == m.javaSrcDir.fullName + "/" +
566         f.belongsTo.fullName.replaceAll(".", "/");
567
568 context MavenModule inv:
569     jarModule implies javaSrcDir.isPresent;
570
571 association subModule
572     [0..1] MavenModule (parent) -> MavenModule [*];
573
574 association dependsOn [*] MavenModule -> MavenModule [*];
575
576 association dependsOnExternal
577     [*] MavenModule -> MavenModule [*];
578
579 association library [0..1] MavenModule -> JavaArchive [*];
580
581 context MavenModule inv:
582     (project <=> parent.isAbsent) &&
583     (!project implies {p in parent** | p.project}.size == 1);
584
585 context MavenModule inv:
586     jarModule implies subModule.isAbsent;
587
588 context MavenModule inv:
589     rootDir.contains(subModule.rootDir);
590
591 context MavenModule m1, MavenModule m2 in m1.dependsOn inv:
592     {m in m1.parent** | m.project} ==
593     {m in m2.parent** | m.project};
594
595 context MavenModule inv:
596     !project implies dependsOn.containsAll(parent.dependsOn);
597
598 context MavenModule inv:
599     !project implies dependsOnExternal
600         .containsAll(parent.dependsOnExternal);
601
602 context MavenModule inv:
603     !project implies library.containsAll(parent.library);
604
605 context MavenModule inv:
```

```

606     !(dependsOn**).contains(this);
607
608     association targetDir [0..1] MavenModule -> Directory[0..1];
609
610     association targetArtifact
611         [0..1] MavenModule -> JavaArchive [0..1];
612
613     context MavenModule inv:
614         jarModule implies targetDir.isPresent &&
615             targetArtifact.isPresent &&
616             targetDir.parent == rootDir &&
617             targetDir.simpleName == "target" &&
618             targetArtifact.parent == targetDir;
619
620     context MavenModule m, JavaSourceFile f in javaSrcFile inv:
621         m.jarModule implies (f.compiledTo.parent.fullName ==
622             ((f.parent == m.javaSourceDir) ?
623                 targetArtifact.fullName :
624                 m.targetArtifact.fullName + "/" +
625                 f.belongsTo.fullName.replaceAll(".", "/")));
626
627     context MavenModule inv:
628         library.containsAll(dependsOnExternal.targetArtifact);
629
630     // _____
631     // VCSRootDir
632     // _____
633
634     class VCSRootDir extends Directory {}
635
636     context MavenModule inv:
637         {VCSRootDir d | d in (rootDir.parent**)}.size == 1;
638
639     // _____
640     // POM
641     // _____
642
643     class POM extends Artifact {
644         String groupId;
645         String artifactId;
646         String version;
647     }
648
649     association pom [1] MavenModule -> POM [1];

```



```

650
651 context POM inv:
652     simpleName == "pom";
653
654 context POM inv:
655     nameExtension == "xml";
656
657 context MavenModule inv:
658     rootDir == pom.parent;
659
660 // _____
661 // MavenPhase
662 // _____
663
664 class MavenPhase {
665     String name;
666     int index;
667 }
668
669 association executes
670     [*] MavenModule -> MavenPhase [*] <<ordered>>;
671
672 association binds [*] MavenPhase -> Tool [*] <<ordered>>;
673
674 context MavenModule inv :
675     executes.size == 7;
676
677 context MavenModule inv:
678     executes.binds.containsAll(parent.executes.binds);
679
680 // _____
681 // MavenExecution
682 // _____
683
684 class MavenExecution extends Action {}
685
686 association buildRoot [*] MavenExecution -> MavenModule [1];
687
688 association providedAsParameter
689     [0..1] MavenExecution -> MavenPhase [1];
690
691 class MavenModuleExecution extends Action {}
692
693 association buildsModule

```

```
694     [1] MavenModuleExecution -> MavenModule [1];
695
696     class MavenPhaseExecution extends Action {}
697
698     association phase
699         [0..1] MavenPhaseExecution -> MavenPhase [1];
700
701     context MavenExecution inv:
702         sub in MavenModuleExecution;
703
704     context MavenModuleExecution inv:
705         sub in MavenPhaseExecution;
706
707     context MavenPhaseExecution inv:
708         sub in ToolExecution;
709
710     context MavenExecution inv:
711         contributesTo == sub.contributesTo;
712
713     context MavenModuleExecution inv:
714         contributesTo == sub.contributesTo;
715
716     context MavenPhaseExecution inv:
717         contributesTo == sub.contributesTo;
718
719     context MavenExecution inv:
720         buildRoot.mavenModuleExecution == sub[0];
721
722     context MavenExecution inv:
723         sub.containsAll(buildRoot.subModule**);
724
725     context MavenExecution me,
726         MavenModuleExecution mme in me.sub,
727         MavenPhaseExecution mpe in mme.sub inv:
728         me.providedAsParameter.index >= mpe.phase.index;
729
730     context MavenModuleExecution inv:
731         sub.containsAll(phase.binds.toolExecution);
732
733     context MavenModuleExecution me,
734         MavenPhaseExecution pe in me.sub inv:
735         pe.phase.name == "package" implies
736         me.buildsModule.targetArtifact.isPresent;
737 }
```



## Appendix B.

# Entire Application Model

```
1 classdiagram AMApplications {
2   // _____
3   // Tool Monitoring Analyses
4   // _____
5
6   class Analysis {}
7
8   association execs Analysis -> ExecutionProtocol [*];
9
10  association /unusedTools Analysis -> Tool [*];
11
12  context Analysis inv:
13    unusedTools == Tool.removeAll(
14      {te.tool | ToolExecution te in execs.action});
15
16  association /unknownTP Analysis -> FreeMarkerTemplate [*];
17
18  context Analysis inv:
19    unknownTP ==
20      {FreeMarkerTemplate t | !t in Generator.template};
21
22  association /usedTP Analysis -> FreeMarkerTemplate [*];
23
24  association /unusedTP Analysis -> FreeMarkerTemplate [*];
25
26  context Analysis inv:
27    usedTP ==
28      {te.template | TemplateExecution te in execs.action};
29
30  context Analysis inv:
31    unusedTP == FreeMarkerTemplate.removeAll(usedTP);
32
33  association /noContrib Analysis -> FreeMarkerTemplate [*];
```

```

34
35 context Analysis inv:
36     noContrib ==
37         {Template t in usedTP | contributionFT.isEmpty};
38
39 // _____
40 // Actual Architecture
41 // _____
42
43 association /containedModule [*] Module -> Module [*];
44
45 context Module inv:
46     containedModule == {this}.addAll(subModule**);
47
48 association /containedArtifact [*] Module -> Artifact [*];
49
50 context Module inv:
51     containedArtifact == containedModule.artifact;
52
53 association /externalArtifact [*] Module -> Artifact [*];
54
55 context Module inv:
56     externalArtifact ==
57         containedArtifact.reliesOn.removeAll(containedArtifact);
58
59 association /reliesOnModule [*] Module -> Module [*];
60
61 context Module inv:
62     reliesOnModule == externalArtifact.module;
63
64 association /moduleDirectory [*] Module <-> Directory [*];
65
66 context Module m, Directory d in m.containedArtifact inv:
67     m.moduleDirectory ==
68         {Directory sub | sub in d.contains**}.add(d);
69
70 association /modulePackage [*] Module <-> Package [*];
71
72 context Module m, JavaArtifact j in m.containedArtifact inv:
73     m.modulePackage == j.package;
74
75 association /packageDirectory [1] Package <-> Directory [*];
76
77 context Package inv:

```

```

78     javaSourceFile.parent in Directory implies
79         packageDirectory == javaSourceFile.parent;
80
81     // _____
82     // Generated Systems
83     // _____
84
85     association /generatedSystem [*] Generator -> System [*];
86
87     context Generator inv:
88         generatedSystem ==
89             {System s | s.consistsOf(toolExecution.contributesTo)};
90
91     // _____
92     // Indirect Template Relations
93     // _____
94
95     association /contributionsRelyOn
96         [*] FreeMarkerTemplate -> FreeMarkerTemplate [*];
97
98     context FreeMarkerTemplate inv:
99         contributionsRelyOn == contributionFT.contributionFT
100             .addAll(contributionFT.refersTo.contributionFT);
101
102     // _____
103     // Incremental Toolchain Execution
104     // _____
105
106     context Artifact inv:
107         toBeRecreated == !isHandcoded && (
108             removed ||
109             lastChange < max(createDependency.lastChange) ||
110             creationDependency.toBeRecreated ||
111             output.output.toBeRecreated);
112
113     context ToolExecution e, Artifact a in e.output inv:
114         a.toBeRecreated implies e.toRepeat;
115
116     // _____
117     // Unused Imports
118     // _____
119
120     association /typeUsage [*] JavaArtifact -> JavaArtifact [*];
121

```

```
122 association /unusedImports
123     [*] JavaSourceFile -> JavaArtifact [*];
124
125 context JavaSourceFile inv:
126     typeUsage == defines.reliesOn.javaArtifact.remove(this);
127
128 context JavaSourceFile inv:
129     unusedImports == imports.removeAll(typeUsage);
130
131 association /typeUsage [*] CDModelFile -> Artifact [*];
132
133 association /unusedImports [*] CDModelFile -> Artifact [*];
134
135 context CDModelFile inv:
136     typeUsage == defines.reliesOn.javaArtifact
137         .addAll(defines.reliesOn.cDModelFile).remove(this);
138
139 context CDModelFile inv:
140     unusedImports == imports.removeAll(typeUsage);
141 }
```

# Index

- Action, 45
- Architecture, 14, 66
- Archive, 12
- Artifact, 8
- artifact, 15
- Artifact Model, 7
- ArtifactContainer, 10
- ArtifactCreate, 47
- ArtifactExistenceCheck, 47
- ArtifactRead, 47
- AST, 36
  
- belongsTo, 21
- binds, 59
- buildRoot, 60
  
- CDModelFile, 37, 73
- Class Diagram, 4, 76
- CoCo, 34
- coco, 34
- compiledTo, 19
- conformsTo, 35
- consistsOf, 13
- containingType, 22
- contains, 22
- contributesTo, 50
- contributionFT, 52
- contributionJava, 52
- creationDependency, 49
  
- defines, 22, 35, 37
- dependsOn, 55
- dependsOnExternal, 55
- Directory, 12
- Event, 45
  
- executes, 59
- ExecutionProtocol, 45
  
- FreeMarkerTemplate, 42, 66, 70
  
- Generator, 69
- generator, 44
- Grammar, 34
- grammar, 34
- GrammarFile, 34
  
- implementedBy, 35
- imports, 18, 37, 73
- includes, 35
- innerType, 22
- input, 49
- inputLanguage, 44
  
- JavaArchive, 20
- JavaArtifact, 18, 73
- JavaClassFile, 19
- JavaMethodCall, 47
- JavaSourceFile, 18
- javaSrcDir, 55
- javaSrcFile, 55
  
- Language, 34
- library, 55
  
- Maven, 53
- MavenExecution, 60
- MavenModule, 54
- MavenModuleExecution, 60
- MavenPhase, 59
- MavenPhaseExecution, 60
- ModelFile, 35



Module, 14, 66  
module, 15  
MontiCore, 33, 35, 37, 41, 47, 57  
MontiDEx, 44  
  
Object Constraint Language, 4, 15, 76  
output, 49  
  
Package, 20  
POM, 57  
produces, 10  
Product, 13  
providedAsParameter, 60  
  
refersTo, 9  
reliesOn, 23, 38  
reliesOnJavaArtifact, 18, 42  
reliesOnTemplate, 42  
rootDir, 54  
  
subaction, 45  
subModule, 15, 55  
subPackage, 22  
Symbol, 18  
System, 13, 69  
  
targetArtifact, 57  
targetDirectory, 57  
template, 44  
TemplateExecution, 47  
Tool, 13  
ToolExecution, 49, 71  
Type, 22, 38  
  
uses, 44  
  
VCSRootDirectory, 57

# Related Interesting Work from the SE Group, RWTH Aachen

## Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: “Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.” Modeling will be used in development projects much more, if the benefits become evident early, e.g. with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR<sup>+</sup>06, GKR<sup>+</sup>08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR<sup>+</sup>09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG<sup>+</sup>14] we discuss how to improve reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

## Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivative of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR<sup>+</sup>06, GKR<sup>+</sup>08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

## Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP<sup>+</sup>98] and describe UML semantics using the “System Model” [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity

diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH<sup>+</sup>98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

## **Domain Specific Languages (DSLs)**

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR<sup>+</sup>06, KRV10, Kra10, GKR<sup>+</sup>08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR<sup>+</sup>07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK<sup>+</sup>11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK<sup>+</sup>07], guidelines to define DSLs [KKP<sup>+</sup>09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

## **Software Language Engineering**

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF<sup>+</sup>15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK<sup>+</sup>15a, HHK<sup>+</sup>13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

## **Modeling Software Architecture & the MontiArc Tool**

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition

and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR<sup>+</sup>11] using deltas [HRRS11, HKR<sup>+</sup>11] and evolution on deltas [HRRS12]. [GHK<sup>+</sup>07] and [GHK<sup>+</sup>08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

## **Compositionality & Modularity of Models**

[HKR<sup>+</sup>09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR<sup>+</sup>07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP<sup>+</sup>09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF<sup>+</sup>15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the “globalized” use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

## **Semantics of Modeling Languages**

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory in [RKB95, BHP<sup>+</sup>98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH<sup>+</sup>97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH<sup>+</sup>98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

## **Evolution & Transformation of Models**

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

## **Variability & Software Product Lines (SPL)**

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK<sup>+</sup>08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR<sup>+</sup>11, HRR<sup>+</sup>11] and to Delta-Simulink [HKM<sup>+</sup>13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK<sup>+</sup>13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

## **Cyber-Physical Systems (CPS)**

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK<sup>+</sup>11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

## **State Based Modeling (Automata)**

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP<sup>+</sup>11].

## **Robotics**

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR<sup>+</sup>13] framework allows robotics experts and laymen to model robotic assembly tasks.

## **Automotive, Autonomic Driving & Intelligent Driver Assistance**

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK<sup>+</sup>07, GHK<sup>+</sup>08]. [HKM<sup>+</sup>13] describes a tool for delta modeling for Simulink [HKM<sup>+</sup>13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW<sup>+</sup>15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural

descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

## **Energy Management**

In the past years, it became more and more evident that saving energy and reducing CO<sub>2</sub> emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP<sup>+</sup>11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

## **Cloud Computing & Enterprise Information Systems**

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK<sup>+</sup>14, HHK<sup>+</sup>15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSElab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

- [BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.
- [BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.
- [BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.
- [BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.
- [BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.
- [BGH<sup>+</sup>97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.
- [BGH<sup>+</sup>98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.
- [BHP<sup>+</sup>98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.
- [BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.
- [BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.



- [CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.
- [CCF<sup>+</sup>15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.
- [CEG<sup>+</sup>14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.
- [CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.
- [CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.
- [EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluwer Academic Publisher, 1999.
- [FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.
- [FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.
- [FLP<sup>+</sup>11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plessner, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.
- [FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plessner, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.
- [GHK<sup>+</sup>07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.
- [GHK<sup>+</sup>08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features,

- Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.
- [GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.
- [GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.
- [GKR<sup>+</sup>06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.
- [GKR<sup>+</sup>07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.
- [GKR<sup>+</sup>08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.
- [GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.
- [GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.
- [GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.
- [GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.
- [GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.
- [GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

- [HHK<sup>+</sup>13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.
- [HHK<sup>+</sup>14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.
- [HHK<sup>+</sup>15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.
- [HHK<sup>+</sup>15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.
- [HKM<sup>+</sup>13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.
- [HKR<sup>+</sup>07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.
- [HKR<sup>+</sup>09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineering in Research and Practice (SERP'09)*, pages 172–176, July 2009.
- [HKR<sup>+</sup>11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.
- [HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.
- [HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.
- [HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

- [HRR<sup>+</sup>11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.
- [HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.
- [HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 1 – 10. fortiss GmbH, 2011.
- [HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.
- [HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.
- [HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.
- [KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.
- [KKP<sup>+</sup>09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.
- [KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.
- [KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

- [KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.
- [Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.
- [KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapman & Hall, 1996.
- [KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.
- [KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.
- [KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.
- [KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.
- [KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.
- [KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.
- [KRV10] Holger Krahn, Bernhard Rumpe, and Stefan 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.
- [LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

- [MRR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.
- [MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.
- [MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.
- [MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.
- [MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.
- [MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.
- [MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.
- [MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.
- [MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.
- [NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.
- [PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.
- [PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

- [PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.
- [PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.
- [PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.
- [Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.
- [RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.
- [RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.
- [RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.
- [RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.
- [RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.
- [RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.
- [RSW<sup>+</sup>15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.
- [Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

- [Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations*, Seattle, pages 697–701. Idea Group Publishing, London, 2002.
- [Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.
- [Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.
- [Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.
- [Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.
- [Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.
- [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.
- [SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Meta-modelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.
- [THR<sup>+</sup>13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P State-charts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.
- [Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.
- [Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.
- [ZPK<sup>+</sup>11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.