Assessing and improving the quality of model transformations

Assessing and Improving the Quality of Model Transformations

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duin, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 9 januari 2012 om 16.00 uur

door

Marinus Franciscus van Amstel

geboren te Tilburg

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.G.J. van den Brand

Copromotor:

dr. A. Serebrenik

Assessing and Improving the Quality of Model Transformations

M.F. van Amstel

Promotor:	prof.dr. M.G.J. van den Brand (Technische Universiteit Eindhoven)
Copromotor:	dr. A. Serebrenik (Technische Universiteit Eindhoven)

Overige leden kerncommissie:

prof.dr. M. Akşit	(Universiteit Twente)
prof.dr. J. Bézivin	(Université de Nantes)
dr.ir. J.J.M. Trienekens	(Technische Universiteit Eindhoven)

The work in this thesis has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute with Vanderlande Industries as the industrial partner. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2011-19.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-2939-1

Printing: Printservice Technische Universiteit Eindhoven
Cover design: Ramon Kool

Preface

In your hands, you hold the manuscript that is the result of my first steps in the world of academic research. The last five years, I worked with great pleasure on the research presented in this thesis. This thesis could, however, not have been completed without the support of a number of people. Therefore, I want to dedicate a few pages to express my gratitude to the people who contributed to this thesis in one way or another.

First and foremost, I want to thank my promotor Mark van den Brand. Mark, I am very grateful for providing me with the opportunity to indeed take these first steps in the academic world. I very much enjoyed our collaboration over the past five years. From the very first meeting we had back in 2006, you stressed that the most important thing to succeed as a PhD. student is to find your 'personal itch'. The search for the right combination between my personal itch and the wishes of the FALCON project proved to be a challenge at first and was therefore sometimes a bit frustrating. Luckily, there were your encouraging motivational speeches. Once we started looking in the area of model-driven engineering and model transformations in particular, I soon knew that this was the topic I wanted to work on. We have had a lot of discussions on this topic over the time. These discussions I have always valued very highly, since most of the time they led to fresh insights and new research ideas to work on. Of course, there was always some time left to talk about non-research related topics. I am very glad to have you as my promotor and I want to thank you for that.

This thesis would not have become what it currently is, if it were not for the influence of a number of people. I want to thank Christian Lange, Alexander Serebrenik, and Tom Verhoeff for the inspiration and feedback they gave me on my research. Christian, or I should say 'Herr Doktor', during the time I was working on my master's thesis, you were my daily tutor. We talked a lot about what it is to be a PhD. student, which made me very enthusiastic about it. You had a great influence on my decision to pursue a PhD. degree myself. Also, it was

your talk during the BENEVOL workshop of 2007 in Namur that inspired me to start my research in the area of model transformation quality. The first steps in this research area we took together. This led to a very nice paper that was published at the ICMT conference in 2009, which Chapter 4 is based on. Tom, the comments you gave during a trial conference presentation made me rethink the contents of Chapter 3. We had some discussions about the meaning of quality in computer science and you thoroughly reviewed the entire chapter related to this topic. I want to thank you for that, since it greatly increased the quality of the chapter. Alexander, when my thesis began to take shape, Mark and I asked you to become my copromotor. I want to thank you for accepting this role and for your many exhaustive reviews of all the chapters in this thesis. Your suggestions, I think, really led to an overall improvement of the thesis. In hindsight, I regret that you were not involved in the process earlier on.

I also want to thank the reading committee, consisting of Mehmet Akşit from Twente University, Jean Bézivin from the University of Nantes, and Jos Trienekens from the Eindhoven University of Technology, for their time to review my thesis.

I have always enjoyed the working atmosphere at the university. This could not have been the same without my colleagues. Therefore, I want to thank my roommates and fellow PhD. students, Jeroen Arnoldus, Yanja Dajsuren, Luc Engelen, Arjan van der Meer, Zvezdan Protić, and Ulyana Tikhonova. The discussions we had in our office and during the infamous coffee breaks, both about research and other stuff that matters to us, were always a welcome interruption of the daily course of academic research. I also want to thank Bogdan Vasilescu, who joined us in the last months of my time as a PhD. student, for his seemingly endless supply of cookies, in which there was always one for me. Two of my colleagues I want to mention in particular, since I had the privilege of collaborating with them quite intensively. Zvezdan, I vividly remember the many long discussions we had at the beginning of our time as PhD. students in the ESI building. These discussions led to the paper that Chapter 2 of this thesis is based on. This paper was selected as one of the best papers of the first ICMT conference. Over the last five years we traveled together to quite a number of conferences. I always enjoyed your company during these trips and the many conversations we had over the years. For sure, I will never forget your passionate stories about 'svinjokolj'. Luc, I want to thank you for the pleasant collaboration on our 'Lego project', which resulted in a number of papers that together form Chapter 8 of this thesis. What started out as a simple test to see if we could generate code from simple models, soon became a case study that was also used for teaching and for starting master's projects around. I think this clearly indicates that our collaboration was a very successful one.

Over the last five years, I was involved in two projects: the FALCON project and the MSQ project. The FALCON project was a multi-disciplinary research project that had as goal increasing automation in warehousing systems. Because

this was a multi-disciplinary project, the project team consisted of a very diverse group of people. It was enlightening to hear from all of you how academic research is carried out in different research disciplines. I want to thank the members of the FALCON team for all the fun times we had, especially during the off-site events, such as the demonstrator days and the Argos trip. The MSQ project was one of the projects carried out as part of the 'kenniswerkersregeling'. The goal of this arrangement was to allow companies to maintain highly knowledgeable employees during the economic crisis by setting them to work at universities. Therefore, the research team did not consist of academic researchers, but of, in this case, business consultants. This difference in background meant that there were different views on how to perform research and how to streamline a collaboration process. I want to thank the members of the MSQ team for this pleasant collaboration and for showing me academic research from a different perspective.

During my time as a PhD. student, I cosupervised a number of students. In order of appearance they are Robert-Jan Bijl, Phu Hong Nguyen, Roy Bouten, Ivo van der Linden, and Tim Huyzen. It was a pleasure to work with every one of you. I particularly enjoyed the many in-depth discussions we had on the various projects you worked on related to my own research.

I also want to thank my close friends, Ad van Amelsfoort, Mark Brouwers, Ramon Kool, Mark Ligtvoet, Roy Maas, Bas Postema, Jeroen Remie, Maarten Rossou, Martijn van Steensel, Jan Stoop, Frank van den Tillaart, and Ferry Vermeer for providing me with the necessary distractions during the 'Boys Nights Out', the basketball practices, the dinners, the vacations, the 'rondjes door de stad', the trips on the racing bike, ... But also for your patience when I tried to explain to you that research in computer science is not about finding out what the best way is to fix your printers or wireless internet connections and that model transformation is not the same thing as generating swimsuit models. Some of you I want to mention in particular. First, Ramon for designing the nifty cover of my thesis and for listening to my nagging on the details of it, or 'miere*n*neuken', with an *n* as you called it. Second, Jan for helping me out with the statistics involved in the empirical studies presented in Chapters 4, and 5. Third, Mark (Ligtvoet) for significantly improving the performance of some of the SQL queries that are part of the tool presented in Section 4.4, or, as you put it, with reducing the time to complete my thesis by a factor sixty. Finally, Martijn and Jan for accepting the role of paranimf to second me during the defense.

Last, but certainly not least, I want to thank my parents, my sister Irma, and my grandmother for their endless and unconditional support. I especially want to thank my mom and dad for all their care to enable me to get this far, this would not have been possible without you.

Marcel van Amstel
Goirle, November 2011

Table of Contents

Chapter 1

Introduction

Throughout the history of computer programming, many different programming languages have been developed. The very first computers had to be programmed directly using binary machine code. Programming using such first generation programming languages was difficult, error-prone, and the programs were specific for the hardware they are designed for. The second generation of programming languages started appearing in the 1940s. In these so-called assembly languages, shorthand notations for machine codes were introduced in the form of symbolic names. These symbolic names are easier to understand for humans than machine code. Therefore, programming using second generation programming languages was easier and less error-prone than using the first generation ones. However, the programs were still specific for the hardware they are designed for. Second generation programming languages are still in use today, albeit in very rare cases. FORTRAN, developed in 1954, was the first third generation, or high-level, programming language [1]. High-level programming languages provide natural language-like statements and data structures. This prevents software engineers from having to program using machine concepts like they had to do with first and second generation programming languages. Therefore, programs written in high-level programming languages are much easier to write and understand. Moreover, they are mostly hardware platform independent.

The developments throughout the history of programming languages have one thing in common, i.e., with every generation, the level of abstraction raises. Although the abstractions provided by, especially high-level, programming lan-

guages greatly improve the productivity of software engineers, they are all oriented towards the computing domain. It is therefore the job of a software engineer to translate a customers' problem into a solution in the computing domain, i.e., a program. Software is becoming ubiquitous and more complex, while at the same time software quality demands increase as well. Model-driven engineering (MDE) is a software engineering paradigm that aims at dealing with this increasing software complexity and improving productivity and quality by raising the level of abstraction at which software is developed to a level where concepts of the domain in which the software has to be applied, i.e., the target domain, can be expressed effectively [2].

1.1 Model-Driven Engineering

Instead of writing code, the focus in MDE is on developing models, i.e., models are first-class citizens. A model is *an abstraction of an entity with a specific purpose* [3]. This is a broad definition that encompasses many things we regularly encounter in our everyday life. For example, the recipe for our favorite dish is a model aimed at describing how to prepare it. Also, an article in a newspaper is a model of an event aimed at describing that particular event. The concept of model is nicely illustrated by René Magritte's painting *La Trahison des Images*, in which not a pipe, but an image (model) of a pipe is shown (see Figure 1.1). In a similar way as the previous examples, a program written in a high-level programming language is a model as well.

Figure 1.1 La Trahison des Images (René Magritte 1929) [4]

The abstractions provided by high-level programming languages are oriented towards the computing domain. Models in MDE are intended to provide abstractions tailored to the domain in which the software should be applied, i.e., the target domain. For that purpose, domain-specific languages (DSLs) are employed. A DSL is a language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular domain [5, 6]. Since the domain concepts provided by DSLs are typically not computing oriented, domain-specific models are not directly usable for, among others, automatic execution. Therefore, these domain-specific models need to be translated to different, computing-oriented, models, preferably automatically. For that purpose, model transformations are employed [7]. A model transformation automatically transforms one or more input models to one or more output models. Using DSLs and model transformations, separation of concerns is achieved, viz., DSLs embody domain knowledge and model transformations embody software engineering knowledge. In the remainder of this section, domain-specific languages and model transformations will be explained in more detail.

1.1.1 Domain-specific Modeling Languages

DSLs are designed to capture the jargon of a specific domain. In this way, they enable modeling using domain concepts rather than concepts provided by existing formalisms, which typically do not provide the required or correct abstractions. In Table 1.1, a number of domains are listed along with examples of DSLs that are used in these domains.

Domain	**DSL**
Music	Staff notation
Traffic	Traffic signs
Arithmetic	Numbers and arithmetic operators
Process descriptions	Flowcharts
Typesetting	LaTeX
Web page mark-up	HTML
Database querying	SQL

Table 1.1 Examples of DSLs

The application of DSLs has both advantages and disadvantages. In Section 1.1.1.1, we discuss a number of advantages as well as a number of disadvantages of using DSLs. Similarly to traditional software engineering artifacts, a DSL also goes through a development cycle. In Section 1.1.1.2, we explain a typical DSL development cycle.

1.1.1.1 Pros and Cons

In a traditional software development process, a group of domain experts provides the software development team with the required information to craft a software solution for their problem. Because these groups of people have different backgrounds, this way of working can lead to various misinterpretations that may jeopardize the adequacy of the developed solution. Since the terminology and expressivity of a DSL is focused on a particular domain, an expert in that domain can understand the models created using a DSL. This reduces the risk of misinterpretations during knowledge transfer. Moreover, domain experts themselves can be more involved in the software development process [8].

Since the level of abstraction of a DSL suits the domain in which a model has to be developed, domain-specific models are typically concise. Moreover, the elements of a DSL embody knowledge of the domain. Therefore, domain-specific models are largely self-documenting. Partly because of these properties, the application of DSLs leads to more expeditious development cycles. Kieburtz et al. report on a study that shows that applying DSLs and accompanying transformations in a software development process indeed increases productivity and also the reliability of the software [9].

Unfortunately, it is not all sunshine and roses. The narrow focus of a DSL limits its applicability, i.e., only to the domain it is designed for. Although the application of DSLs leads to increased productivity in a software development process, the development of the DSL itself entails certain costs as well. Moreover, the code generated from domain-specific models may be less efficient than hand-written code.

1.1.1.2 Development

The development process of DSLs consists of three phases [5, 10], viz., domain analysis, language design, and language implementation.

Domain Analysis

In the domain analysis phase, the concepts that are perceived to be important by domain experts are defined, as well as the relations among them [11]. Aside from the input of domain experts, there are other sources of domain knowledge that can be used for the domain analysis. These are, among others, source code and documentation of existing systems in the domain. In this way, the domain for which a DSL has to be developed is defined by consensus, and its essence is the shared understanding of a community [12].

Language Design

In the language design phase, the concepts and relations defined in the domain analysis phase are grouped into semantic notions [13]. These semantic notions need to be mapped to syntactic carriers to enable the development of models. In the context of MDE, typically metamodels are used to define the syntax of a DSL. A metamodel describes the model elements that are available for developing a class of models as well as their attributes and interrelations. Analogously, a model describes the elements of a real-world object as well as their attributes and the way they interrelate. Therefore, a metamodel can be considered as a model of a class of models [14], or a model of a modeling language [15]. Since a metamodel is itself a model, the concepts and relations that can be used to define them need to be described as well. The metamodel used for this purpose is called a meta-metamodel. A meta-metamodel is typically a reflexive metamodel. This means that it is expressed using the concepts and relations it defines itself. This four-layer metamodeling architecture [16] is schematically depicted in Figure 1.2.

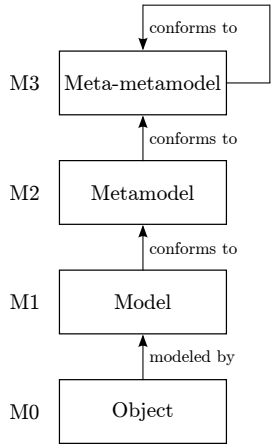


Figure 1.2 Four-layer metamodeling architecture

Language Implementation

In the language implementation phase, tools are developed that enable using the DSL. Typically this means that the models created using the DSL have to be executed. Three strategies for implementing DSLs can be distinguished for achieving this, viz., compiler or interpreter development, embedding, and transformation [5].

First, a compiler or interpreter can be developed to enable execution of domain-specific models. In this stand-alone approach there is no dependence on

other languages. This implies that no concessions have to be made with respect to domain-specificness. However, implementing and maintaining a compiler or interpreter is typically a costly task.

Second, a DSL can be embedded in an existing language, i.e., a host language. This is achieved by adding libraries implementing the domain-specific abstractions to this host language. The advantage of this approach is that tools available for the host language can be used for the DSL as well. However, since the DSL is an extension of the host language, it also inherits its look and feel. This may imply that concessions have to be made with respect to domain-specificness, i.e., certain domain-specific constructs may not be expressible as well as desired.

Last, domain-specific models can be transformed into different models that are suitable for the required purpose, e.g., execution or simulation. In MDE, typically this transformational approach is adopted. The main advantage of this approach is the flexibility it provides by reusing other formalisms. Enabling the use of the DSL for a different purpose solely requires the implementation of another model transformation. In this way, model transformations facilitate the transfer of models to and from specialized tools during the development life-cycle [17]. Implementing model transformations is facilitated by a growing number of model transformation languages (see Section 1.1.2.3) and is therefore a less effortful task than implementing a compiler or interpreter. The disadvantage of a transformational approach is that analyses are not performed on the domain level, but on the level of the target language [18]. This raises the issue of traceability, i.e., results acquired from analyzing the target models of a model transformations have to be related to their corresponding domain-specific source models. Moreover, it may occur that there is a semantic gap between the DSL and the target language. This means that for some constructs of the DSL to be transformed to the target language a laborious and inefficient transformation may be required, if transformation is possible at all.

1.1.2 Model Transformation

Model transformation has been defined in literature in a number of different but similar ways. The Object Management Group defines model transformation as the process of converting one model to another model of the same system [19]. According to the definition of Kleppe et al., a model transformation is the automatic generation of a target model from a source model according to a transformation definition [20]. They define a transformation definition as a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language, where a transformation rule is a description of how one or more elements in the source language can be transformed into one or more elements in the target language. Kurtev provides

the following definition. A model transformation is the process of automatic generation of a target model from a source model, according to a transformation definition, which is expressed in a model transformation language [15]. According to Tratt's definition, a model transformation is a program that mutates one model into another [17]. Mens and van Gorp suggest a generalization of the definition of Kleppe et al. by taking into consideration multiple source and target models [21]. Sendall defines model transformation as an automated process that takes one or more source models as input and produces one or more target models as output, while following a set of transformation rules [7]. The definition we adopt is the following. A model transformation is *a mapping from a set of source models to a set of target models defined as a set of transformation rules.*

The origin of model transformation can be traced back to string rewriting systems, or semi-Thue systems, introduced by Axel Thue in the early twentieth century [22]. A semi-Thue system $\mathcal{T}$ is a binary relation on strings from an alphabet Σ, i.e., $\mathcal{T} \subseteq \Sigma^* \times \Sigma^*$, where each element $(u, v) \in \mathcal{T}$ is a rewrite rule. A model transformation can be defined in a similar way. Let $\mathcal{E}$ be the set of all model elements. Then a model m is a finite non-empty subset of $\mathcal{E}$, i.e., $m \subseteq \mathcal{E} \setminus \emptyset$. The set of all possible models, $\mathcal{M}$, is then a finite non-empty subset of all combinations of model elements, i.e., $\mathcal{M} = \mathcal{P}(\mathcal{E}) \setminus \emptyset$. A set of source models S is a subset of the set of all models, i.e., $S \subseteq \mathcal{M}$. Similarly, a set of target models T is also a subset of the set of all models, i.e., $T \subseteq \mathcal{M}$. Possibly the sets of source and target models overlap, i.e., $S \cap T \neq \emptyset$. A model transformation $\mathcal{MT}$ is then a binary relation on sets of elements from source models $\mathcal{P}(\bigcup S)$ and target models $\mathcal{P}(\bigcup T)$, i.e., $\mathcal{MT} = \mathcal{P}(\bigcup S) \times \mathcal{P}(\bigcup T)$, where each element $(s, t) \in \mathcal{MT}$ is a transformation rule. If $\mathcal{MT}$ is a symmetric relation, then the model transformation is a bidirectional model transformation [23].

1.1.2.1 Architecture

Figure 1.3 schematically depicts the architecture of a model transformation [24]. The source and target models of a model transformation both adhere to the four-layer metamodeling architecture depicted in Figure 1.2. A model transformation can be considered as a model as well. Therefore, it also adheres to the four-layer metamodeling architecture. In case of model transformation, the *M0* (object) layer represents a run-time instance of a transformation. Note that for the source and target models the *M0* layer is not shown in the figure.

Model transformations are generally developed with reuse in mind. If a transformation would be needed only once, it would suffice to craft the target models by hand instead of taking the trouble to develop a model transformation. Therefore, a model transformation definition is not based on model instances, but on metamodels describing the set of models conforming to them. A transformation definition consists of a number of transformation rules. A transformation rule is a

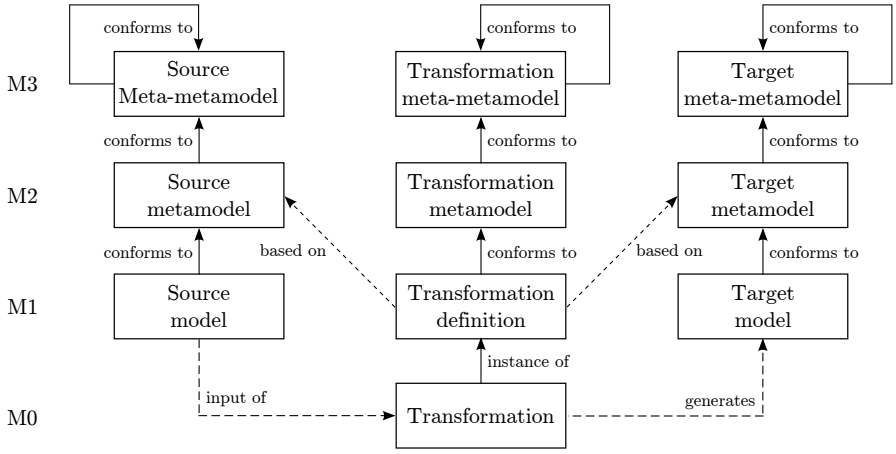


Figure 1.3 Transformation architecture

description of how one or more elements in the source language can be transformed into one or more elements in the target language [20].

1.1.2.2 Implementation Approaches

Three approaches for developing model transformations can be distinguished, viz., direct model manipulation, using an intermediate representation, and using a dedicated model transformation language [7].

Direct Model Manipulation

When implementing a model transformation by direct model manipulation, it is required that an API[1] is defined on both the source and the target metamodel of the transformation. These APIs enable direct access to the internal representation of the models involved in the transformation. In this way, they can be read and modified. An existing general-purpose programming language is used in combination with these APIs to implement the model transformation.

The advantage of this direct model manipulation approach is that transformation developers can use a general-purpose programming language they are familiar with. This prevents them from having to learn a new language, which may accelerate the development of the transformation. Moreover, existing libraries for the general-purpose language can be reused to facilitate implementation of complex algorithms that may be required for the transformation.

[1] Application Programming Interface

The disadvantage of this approach is that it does not scale well. A general-purpose programming language is not designed for model transformation, i.e., its level of abstraction does not fit the domain of model transformation. Moreover, the APIs themselves are abstractions from the metamodels that deviate from the abstractions intended by the language designers. Therefore, it is hard to write and maintain transformations using this approach. For small transformations the direct model manipulation approach is viable, but for larger transformations it is not suited.

Intermediate Representation

When implementing a model transformation using the intermediate representation approach, the source model of a transformation is exported to a format that is suitable for automatic transformation. Typically, an XML format such as the XML Metadata Interchange (XMI) format [25] is used for this. XMI is a standard for exchanging models between modeling tools. Transformation of XML documents is commonly performed using eXtensible Stylesheet Language Transformations (XSLT) [26] or using a library with functions for manipulating XML documents in combination with a high-level programming language. Such a library is in fact a DSL for manipulating XML documents embedded in a high-level programming language. Languages with comprehensive regular expression support such as Perl [27], Python [28], and Ruby [29] can also be used for transforming XML documents. However, these languages are suitable for simple transformations only, since implementations in these languages need to deal with the elaborate concrete syntax of XMI rather than with abstract syntax [30].

Most modeling tools provide functionality for exporting models to some form of XML. Therefore, the intermediate model approach is applicable to the majority of models that are being developed. The advantage of the approach is that existing tools for manipulating XML documents can be reused.

In contrast with a general-purpose programming language used in the direct model manipulation approach, a transformation language like XSLT or a library with functions for manipulating XML documents is designed for transformation of, in this case, XML documents. Therefore, the level of abstraction of the transformation language suits the level of abstraction of the intermediate representation of the models involved in a transformation. However, the XML representations of the models are, again, abstractions that deviate from the abstractions intended by the language designers. For instance, XML documents describe tree structures, whereas models typically have a graph structure. This, again, has repercussions for the scalability of transformations implemented using the intermediate representation approach. Furthermore, XSLT has no facilities for the implementation of complex algorithms that may be required for a transformation. However, a library for manipulating XML documents embedded in a high-level programming

language does not suffer from this problem, since the host language can be used for implementing algorithms.

Model Transformation Language

The emergence of MDE has lead to the development of numerous dedicated model transformation languages. These languages are in fact DSLs in the domain of model transformation. A model transformation can be considered as a model conforming to that language. This nicely fits the basic principle of MDE stating that everything is a model [31]. In Section 1.1.2.3, a number of model transformation languages and tools are discussed.

The advantage of using a dedicated model transformation language over the direct model model manipulation and intermediate representation approach is that the level of abstraction fits the domain it operates in. Therefore, frequently required operations such as pattern matching and model element creation are standard functionality in most model transformation languages.

While the level of abstraction provided by model transformation languages is appropriate for performing model transformations, it is often inappropriate for implementing complex algorithms which may be required for a transformation task. Therefore, some transformation languages provide escape mechanisms to enable inclusion of code from general-purpose programming languages in a model transformation.

1.1.2.3 Model Transformation Languages

This section shortly describes four model transformation languages, viz., ATL, QVT operational mappings, Xtend, and ATL. These four are selected, since these are the ones studied in this thesis.

ATL

The ATLAS Transformation Language (ATL) [32–34] has been developed in response to the QVT request for proposals issued by the Object Management Group (OMG) [35]. However, since then ATL has evolved separately from the QVT standard due to changing requirements resulting from experience with the language [36]. At the time of writing this thesis, ATL is one of the most widely used model transformation languages around. This can, among others, be concluded from the large number of publications in which ATL is used for developing model transformations and for experimenting with new transformational techniques (see for example [37–40]).

An ATL model transformation generates a new write-only target model from a read-only source model. ATL model transformations can be executed in one direction only, i.e., they are unidirectional. In-place model transformations, i.e., a

model transformation that adapts a model rather than generating a new one, can be simulated using the *refining mode* in which the untransformed part of the input model is automatically copied to the output model. This is not an actual in-place transformation, since the input model is not adapted but a new target model is generated. An ATL model transformation consists of transformation rules and helpers. Transformation rules are used to generate target model elements, generally from source model elements. Rules can be specified using both a declarative and an imperative style. Therefore, ATL is a hybrid language. The declarative style is used for specifying relations between source and target model elements. The developers of ATL encourage using this declarative style as much as possible [33]. The imperative style can be resorted to when it is hard to find a declarative solution for a transformation problem. Helpers are expressions defined in the Object Constraint Language (OCL) [41]. Their purpose is to define reusable chunks of code, mainly for navigating source models and storing values. ATL has support for modularity, although this is limited. Libraries containing helpers only can be defined. However, transformation rules cannot be distributed over different modules. It is possible to define parts of a transformation in Java, however this is very limited and currently not documented in the user guide [42].

The ATL language is accompanied by a set of tools for developing model transformations. This toolset is implemented as an Eclipse plug-in [43] and consists of the ATL engine, an editor, and a debugger. The ATL engine consists of two components, viz., the ATL compiler that compiles ATL code to bytecode and the ATL virtual machine that executes the bytecode. Although the ATL virtual machine does not explicitly require this, typically the models that are transformed with ATL are created using the Eclipse Modeling Framework (EMF) [44].

QVT

Query/View/Transformation (QVT) is a model transformation language standardized by the OMG [45]. QVT consists of three sub-languages, viz., relations, core, and operational mappings. Model transformations in the QVT relations language are expressed as relationships between models. The QVT core language is a declarative language that is equally powerful as the relations language. However, it is defined at a lower level of abstraction. The semantics of the relations language can be described by the core language. The QVT operational mappings language (QVTO) provides means to implement a model transformation using an imperative programming style. The operational mappings language can also be used to complement transformations specified in the relational language with imperative operations. This is referred to as a *hybrid* approach.

A QVTO model transformation can either generate a new target model from a source model or be an in-place model transformation. Like ATL, model transformations in QVTO are unidirectional. A QVTO model transformation

consists of operations. The mapping operation is the key operation in QVTO. It is used to implement mappings from source to target model elements. Another important operation type is the helper. Helpers are used to perform computations on model elements. In QVTO it is possible to implement parts of a transformation in Java, using *blackbox* operations. QVTO supports two forms of modularity. First, it is possible to define types and operations in library modules. Second, a chain of transformations can be created by composing multiple model transformations in one large transformation.

There are a number of implementations for the QVTO language, notably SmartQVT [46] and Eclipse M2M [47]. Both are implemented as Eclipse plug-ins and are restricted to transforming EMF-based models.

Xtend

OpenArchitectureWare is a framework aimed at facilitating MDE by providing tools for metamodel development, model validation, model transformation, and code generation [48]. Xtend is the model transformation language of this framework. The developers claim it to be a functional language. However, since functions can have side effects it is not a *pure* functional language.

Xtend supports both creation of new target models and in-place transformation. Similar to the aforementioned languages, model transformations in Xtend are unidirectional. Transformation functions in Xtend are called extensions. Besides creating model elements, navigation of source models and performing computations is also done using extensions. Xtend provides, similar to QVTO, facilities to implement parts of a transformation in Java by means of Java extensions. Modularity is supported as well, i.e., the extensions that comprise a model transformation can be divided over different modules.

All of the openArchitectureWare components are implemented as Eclipse plug-ins. For Xtend this means that it is restricted to transformation of EMF-based models. Xtend is an interpreted language. However, the successor of Xtend will have a compiler that compiles a model transformation directly to Java classes.

ASF+SDF

ASF+SDF is a term rewriting system that is mainly used for transformations between languages [49]. Although it was not designed for application in an MDE setting, it has been successfully applied in a number of MDE projects [50, 51]. Transformations are performed between languages specified in the syntax definition formalism SDF [52]. The main difference with the model transformation languages discussed before is that SDF is based on context-free grammars, rather than on metamodels. Consequently, the models that can be transformed with ASF+SDF have a tree structure instead of a graph structure. Transformations are implemented as conditional equations specified in the algebraic specification

formalism ASF [53]. ASF can best be described as a functional language. The two main advantages of ASF+SDF are modularity and syntax-safety. Syntax-safety in the context of model transformations means that every syntactically correct source model is transformed into a syntactically correct target model and that syntactically incorrect source models are not transformed at all.

ASF+SDF transformations are also unidirectional. When performing a transformation, a new target model is always generated. However, similar to ATL, in-place model transformations can be simulated using so-called transforming traversal functions. Transformations in ASF+SDF consist of transformation functions. Transformation functions are defined by signatures and equations. Signatures are defined in SDF, they describe the syntactic structure of a transformation function. Equations are defined in ASF, they form the implementation of a transformation function. Equation can have zero or more conditions. These conditions can be used, among others, to assign values to variables. In ASF+SDF it is possible to implement parts of a transformation in C.

ASF+SDF is the only language of the four described here that is not implemented as an Eclipse plug-in. Instead, it has its own development environment, viz., the Meta-Environment [54]. A transformation is executed in three steps. A source model, defined in SDF, is parsed using the SGLR parser [55] resulting in a parse tree. Thereafter, the transformation specification, which has been transformed to C code, is applied to that parse tree. The result of the transformation is again a parse tree, which can be unparsed to acquire the target model.

Summary

Table 1.2 summarizes the characteristics of the model transformation languages discussed above.

1.2 Problem Statement

MDE is gradually being adopted by industry [56]. Since MDE is becoming increasingly important, so are model transformations. Model transformations are in many ways similar to traditional software artifacts, i.e., they have to be used by multiple developers, have to be changed according to changing requirements, and should preferably be reused. Therefore, they need to adhere to similar quality standards as well. To attain these standards, a methodology for developing model transformations with high quality is required. However, before such a methodology can be developed, quality needs to be defined in the context of model transformation and factors that influence it should be identified. To identify these influences, there should be a methodology for assessing the quality of model transformations. Such methodologies have been developed for different kinds of software artifacts, but not yet for model transformations.

Characteristic	ATL	QVTO	Xtend	ASF+SDF
Directionality	Unidirectional			
In-place support	Refining mode	Yes		Transformer traversal functions
Transformation functions	Rules/helpers	Mappings/ helpers	Extensions	Equations
Programming style	Declarative/ imperative	Declarative	Functional	
Modularity	Helpers only	Yes, also composition of transformations	Yes	
Escape mechanism	Java, limited	Java		C
Platform	Eclipse			Meta-environment
Metamodeling formalism	Typically EMF	EMF		SDF

Table 1.2 Transformation language characteristics

An approach to increase the quality of model transformations is to support their development and maintenance process by means of analysis techniques. Numerous analysis techniques supporting development and maintenance exist for all kinds of software artifacts such as source code or models. However, few techniques currently exist for analyzing model transformations. A reason for this is that MDE, and thereby model transformation, is a relatively young research discipline. Most effort is currently invested in applications and in improving model transformation languages, techniques, and tools [37–40]. To prevent model transformations from becoming the next *maintenance nightmare*, analysis techniques should be developed for them to assist in the development and maintenance process. Moreover, appropriate tool support is required for further adoption of MDE by the industry [57, 58].

1.3 Research Questions

To address the problems stated in Section 1.2, we formulate the central research question to be discoursed in this thesis as follows.

> **RQ:** *How can the quality of model transformations be assessed and improved, in particular with respect to development and maintenance?*

In the remainder of this section we will decompose this central research question into more detailed research questions.

In the development cycle of a (software) system, simulations and analyses are often performed before the actual system is implemented, particularly when high-cost hardware is involved. To enable simulations and analyses, models are used. Traditionally, these models are no longer used afterwards and implementation of the actual system starts from scratch. By employing model transformations, these models can be (partially) reused for code generation. For these simulations and analyses a simplified view of the world is typically assumed. For instance, the notion of time may be abstracted from. For simulation and analysis this is fine. In real-world applications, however, this is unthinkable. Therefore, to successfully apply model transformations for generating code from simulation and analysis models, such semantic gaps need to be identified and bridged. This leads to the following research question.

> **RQ_1**: *How should model transformations that transform between models from different semantic domains be approached?*

Model transformations that need to bridge a semantic gap tend to be more complex than those that merely transform syntax. This increased complexity has consequences for the quality of these model transformations. Quality, in general, is a subjective concept. For instance, when a number of people are asked how they would rate the quality of a car, they will give different answers. For some its fuel consumption may be important, whereas for others this may be its durability or its ride comfort. Similarly, there are different faces to the quality of model transformations. This leads to the following research question.

> **RQ_2**: *How can quality be defined in the context of model transformations?*

Once quality has been defined in the context of model transformations, it can be assessed. There are many methods around that can be employed for assessing the quality of software artifacts, such as for example testing, auditing, and collecting metrics. Software metrics have been studied extensively over the last decades and have been proposed for measuring various kinds of software artifacts [59]. In this thesis, we focus on using metrics for assessing the quality of model transformations. This leads to the following research question.

> **RQ$_3$**: *How can metrics be used to assess the quality of model transformations?*

Metrics are used to measure certain characteristics of (software) artifacts, in this case model transformations. By empirically relating the identified metrics to different attributes of quality, insights are acquired as to what characteristics of model transformations affect their quality. These insights enable increasing the overall quality of model transformations and thereby also their maintainability. In traditional software engineering processes, often visualization techniques are employed to support maintenance and development. For model transformations, this approach can be adopted as well. This leads to the following research question.

> **RQ$_4$**: *What visualization techniques can be employed to support the development and maintenance process of model transformations?*

The development and validation of techniques and tools for answering the foregoing research questions have led to insights regarding the development of model transformations. Application of these insights, techniques, and tools should result in model transformations of higher quality. This leads to the following research question.

> **RQ$_5$**: *How can the acquired insights and developed techniques be applied to facilitate the development and maintenance of model transformations?*

1.4 Thesis Outline

In this section, we provide an outline of the structure of the remainder of this thesis. For every chapter we indicate the research question it addresses as well as the previous publications it is based on. Note that since this thesis is based on articles, there may be some overlap among the different chapters.

Chapter 2: Revealing and Bridging a Semantic Gap

In this chapter, we address research question RQ$_1$. We discuss a model transformation from models suitable for analysis, i.e., process algebra models, to models suitable for code generation, i.e., UML state machines. This transformation gives rise to a semantic gap. We identify this semantic gap and propose a solution for bridging it. This chapter is based on the following publication.

> M.F. van Amstel, M.G.J. van den Brand, Z. Protić, and T. Verhoeff: *Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap?* In: Theory and Practice of Model Transformations, Proceedings of the First International Conference on Model

Transformation (ICMT 2008), volume 5063 of Lecture Notes in Computer Science, pages 61–75, Zurich, Switzerland, July 2008 [50].

Chapter 3: Quality of Model Transformations

In this chapter, we address research question RQ_2. We distinguish two different views on model transformation quality as well as two different approaches for assessing it. For both assessment approaches we provide examples and discuss their advantages and disadvantages when used for assessing the different views on quality. In this chapter, we also explain why quality attributes that have been used for evaluating the quality of traditional software artifacts are relevant for model transformations as well. This chapter is based on the following publications.

M.F. van Amstel, C.F.J. Lange, and M.G.J. van den Brand: *Metrics for Analyzing the Quality of Model Transformations.* In: Proceedings of the Twelfth ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering (QAOOSE 2008) (co-located with ECOOP 2008), pages 41–51, Paphos, Cyprus, July 2008 [60].

M.F. van Amstel: *The Right Tool for the Right Job: Assessing Model Transformation Quality.* In: Proceedings of the Fourth IEEE International Workshop on Quality Oriented Reuse of Software (QUORS 2010) (co-located with COMPSAC 2010), pages 69-74, Seoul, South Korea, July 2010 [61].

Chapter 4: Quality Assessment of ASF+SDF Model Transformations

This is the first of three chapters in which we address research question RQ_3. We propose a set of metrics for evaluating the quality of model transformations developed using the term rewriting system ASF+SDF. To assess whether the metrics we propose are valid predictors for the quality attributes defined in Chapter 3, we have conducted an empirical study. This study shows that for most of the quality attributes there are metrics that correlate with them. This chapter is based on the following publications.

M.F. van Amstel, C.F.J. Lange, and M.G.J. van den Brand: *Metrics for Analyzing the Quality of Model Transformations.* In: Proceedings of the Twelfth ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering (QAOOSE 2008) (co-located with ECOOP 2008), pages 41–51, Paphos, Cyprus, July 2008 [60].

M.F. van Amstel, C.F.J. Lange, and M.G.J. van den Brand: *Using Metrics for Assessing the Quality of ASF+SDF Model Transformations.* In: Theory and Practice of Model Transformations, Proceedings of

the Second International Conference on Model Transformation (ICMT 2009), volume 5563 of Lecture Notes in Computer Science, pages 239–248, Zurich, Switzerland, June 2009 [62].

Chapter 5: Quality Assessment of ATL Model Transformations

This is the second of three chapters in which we address research question RQ_3. In this chapter, we propose a set of metrics for evaluating the quality of transformations developed using the model transformation language ATL. We also report on a similar empirical study as presented in Chapter 4. This chapter is based on the following publications.

M.F. van Amstel and M.G.J. van den Brand: *Quality Assessment of ATL Model Transformations using Metrics.* In: Proceedings of the Second International Workshop on Model Transformation with ATL (MtATL 2010), volume 711 of CEUR Workshop proceedings, pages 19–33, Málaga, Spain, June 2010 [63].

M.F. van Amstel and M.G.J. van den Brand: *Using Metrics for Assessing the Quality of ATL Model Transformations.* In: Proceedings of the Third International Workshop on Model Transformation with ATL (MtATL 2011), volume 742 of CEUR Workshop proceedings, pages 20–34, Zurich, Switzerland, July 2011 [64].

Chapter 6: Comparing Metric Sets for Model Transformations

This is the last of three chapters in which we address research question RQ_3. In this chapter, we propose two additional sets of metrics for two different model transformation languages, i.e., Xtend and QVT operational mappings. These two metric sets are compared with the metric sets for ASF+SDF and ATL. We discuss in this chapter the overlap and differences among them. This chapter is based on the following publication.

M.F. van Amstel, M.G.J. van den Brand, and P.H. Nguyen: *Metrics for Model Transformations.* In: Proceedings of the Ninth Belgian-Netherlands Software Evolution Workshop (BENEVOL 2010), Lille, France, December 2010 [65].

Chapter 7: Visualization Techniques for Model Transformations

In this chapter, we address research question RQ_4. We present in this chapter four different visualization techniques focused on facilitating model transformation comprehension. Two of these techniques are aimed at visualizing dependencies between the different components that comprise a model transformation. These techniques have been applied before to traditional software artifacts. Since model

transformations are in many ways similar, the techniques apply to them as well. The other two techniques are specific for model transformations. They visualize the relation between a model transformation and the metamodels it is defined on. This chapter is based on the following publication.

> M.F. van Amstel and M.G.J. van den Brand: *Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare.* In: Theory and Practice of Model Transformations, Proceedings of the Fourth International Conference on Model Transformation (ICMT 2011), volume 6707 of Lecture Notes in Computer Science, pages 108–122, Zurich, Switzerland, June 2011 [66].

Chapter 8: Fine-grained Model Transformations

In this chapter, we address research question RQ_5. We discuss in this chapter our experiences with developing a DSL and accompanying model transformations. The purpose of this DSL is to enable application of formal methods by means of model transformations. Consequently, no separate models have to be developed for that purpose. These model transformations, again, need to bridge semantic gaps. During their development, the quality attributes presented in Chapter 3 were taken in mind as well as the quality attribute verifiability. Moreover, the techniques and tools presented in Chapter 7 were used during the development process to assess their applicability. This chapter is based on the following publications.

> M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen: *An Exercise in Iterative Domain-Specific Language Design.* In: Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL 2010), pages 48–57, Antwerp, Belgium, September 2010 [67].

> M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen: *Using a DSL and Fine-grained Model Transformations to Explore the Boundaries of Model Verification – Extended Abstract.* In: Proceedings of the Seventh Workshop on Advances in Model Based Testing (A-MOST 2011), pages 63–66, Berlin, Germany, March 2011 [68].

> M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen: *Using a DSL and Fine-grained Model Transformations to Explore the Boundaries of Model Verification.* In: Proceedings of the Third Workshop on Model-Based Verification & Validation From Research to Practice (MVV 2011) (co-located with SSIRI 2011) (MVV 2011), pages 120–127, Jeju Island, South Korea, June 2011 [69].

Chapter 9: Conclusions

This chapter concludes the thesis. In this chapter, we recapitulate the main contribution of this thesis and reflect on the research questions. We will also discuss opportunities for further research.

Chapter 2

Revealing and Bridging a Semantic Gap

Many formalisms exist for modeling the behavior of (software) systems, each having a different goal. Process algebras are used for algebraic and axiomatic reasoning about the behavior of distributed systems. UML state machines are suitable for automatic software generation. We want to enable automatic software generation from process algebra models by transforming them to UML state machines. This transformation needs to preserve both behavioral and structural properties of the original model. The combination of these preservation requirements gives rise to a semantic gap*. This semantic gap implies that we cannot transform ACP models into UML state machines on a syntactic level only, but we have to take into account the semantics as well. We reveal the semantic gap between the formalisms and propose a way of bridging it.*

2.1 Introduction

There are many formalisms available for modeling the behavior of (software) systems, and more are being defined [70]. This variety of formalisms is the result of the need for different viewpoints on the behavior of systems. Therefore, some of these formalisms are more appropriate for visualization, some more for verification, and some more for automated software generation.

When transforming a model specified in one formalism into a model specified in another formalism, two issues play a role. First, there is a syntactic difference between the formalisms. Transforming syntax is a well-known and relatively simple problem. Second, since models in different formalisms serve different purposes, there can be differences in the semantic domains of the source and target formalism. In particular, these semantic domains may involve different paradigms. Also, a formal semantics may be lacking altogether. Typically, it is required that a model transformation preserves semantic properties as much as possible. Therefore, semantic gaps between the formalisms need to be revealed and bridged. In this chapter, we address research question RQ_1 by discussing a model transformation that transforms between two semantic domains that appear to be similar, but are in fact quite far apart.

> **$\mathbf{RQ_1}$**: *How should model transformations that transform between models from different semantic domains be approached?*

Process algebras are used for algebraic and axiomatic reasoning about the behavior of (concurrent) systems [71]. Typically, process algebra models are solely used for analyzing systems, in particular for verifying their correctness. We would like to use these verified models for generating correct executable code as well. However, little is known about automatically generating software from process algebra models. We propose to generate executable code from process algebra models using UML state machines [72] as an intermediate step, since multiple techniques are available for automated software generation from UML state machines. The state machine diagram is one of the thirteen diagram types available in the Unified Modeling Language (UML). It is a visual means to describe the behavior of an object by modeling the lifecycle of an object over time [73]. Although, there exist no official formal semantics of UML state machines, a number of attempts have been made to formalize their semantics [74–77]. We start with plain process algebra, i.e., without time, data, stochastic, etc., to acquire understanding of code generation from, and model transformations based on process algebras. Therefore, we use the well-known Algebra of Communicating Processes (ACP) [78, 79]. Already in the transformation of this limited process algebra we encounter a semantic gap. An advantageous side-effect of this transformation is that textual process algebra models can be visualized by means of UML state machine diagrams. Since UML is widely used in industry, this transformation makes formal methods, like process algebra, more accessible to the industry.

Processes in distributed systems are typically allocated on different machines that operate in parallel. Therefore, we have to ensure that the automatically generated software can also be deployed on a set of machines. This requires structural equivalence of the ACP models and the obtained state machines with respect to parallel decomposition. Structure preservation also facilitates traceability, which

in turn aids validation of algebraic specifications. An ACP model can be analyzed by studying the behavior of the state machine acquired from the transformation using a UML tool. When unintended or erroneous behavior is detected, this needs to be traced back to the original ACP specification such that it can be adapted appropriately. If structure is not preserved, then it can be hard to determine where the ACP model needs to be corrected. The ACP models and the state machines obtained from the transformation need to exhibit the same observable behavior. We do not require full semantic equivalence, but trace equivalence, i.e., it suffices if the state machine can generate the same execution traces as the corresponding ACP model. It is this combination of requirements, viz., preserving structural and behavioral properties, that gives rise to a semantic gap. This means that the transformation from ACP models to UML state machines encompasses more than just translating syntax. Special care is needed to ensure that the semantic gap is bridged to preserve both behavioral and structural properties.

The remainder of this chapter is structured as follows. In Section 2.2, some background information on ACP and UML state machines is provided. In Section 2.3, the transformation from ACP to UML state machines is described. In this section the semantic gap between the formalisms is revealed and our solution for bridging it is discussed. We implemented the transformation in a tool such that it can be applied automatically. This implementation is described in Section 2.4. An example of the transformation of an ACP model using our implementation is described in Section 2.5. In Section 2.6, we position our approach with respect to related work. Section 2.7 concludes this chapter.

2.2 Preliminaries

2.2.1 Process Algebra

Process algebras are used for studying the behavior of distributed systems [71]. The word *process* refers to behavior of a system and the word *algebra* indicates that the reasoning about behavior is done in an algebraic and axiomatic manner. Process algebras have been developed for describing parallel behavior, since this was difficult in the methods of denotational, operational, and axiomatic semantics [71]. Three well-known process algebra formalisms are Milner's Calculus of Communicating Systems (CCS) [80], Hoare's Communicating Sequential Processes (CSP) [81], and the Algebra of Communicating Processes (ACP) from Bergstra and Klop [78, 79]. Most other process algebra formalisms are extensions of one of the three aforementioned ones. For instance, Timed CSP [82] is an extension of CSP that includes time, mCRL2 [83] extends ACP with data and time, and Timed CCS [84] is an extension of CCS that includes time. Here we focus on the Algebra of Communicating Processes.

2.2.1.1 Algebra of Communicating Processes

A concurrent system in ACP is described by a process term and a communication function. Process terms are composed of atomic actions and operators. The communication function describes how the atomic actions may interact.

Atomic actions are abstractions of real-world events. Typically, meaningful names such as *send* or *receive* are used to denote the atomic actions. Since these actions are atomic, they cannot be divided into smaller actions. In addition to atomic actions, there are two constants for expressing termination. First there is the deadlock constant 0, denoting unsuccessful termination. Second there is the empty process constant 1, denoting successful termination.

Operators are used to compose atomic actions and process terms to represent behavior. We consider seven of the ACP operators.

- Sequential composition ($\cdot$) and action prefix (.)
  The sequential composition of n process terms, $P_1 \cdot P_2 \cdot \ldots \cdot P_n$, denotes that the execution of process term P_1 precedes the execution of process term P_2 and so on. The process term $a.P$, containing the action prefix operator, denotes that first action a is executed, whereafter process term P is executed. From the operational semantics of ACP can be derived that the action prefix operator is similar to the sequential composition, therefore we consider it as such.

- Alternative composition ($+$)
  The alternative composition of n process terms, $P_1 + P_2 + \ldots + P_n$, denotes that only one of these process terms is executed. The choice for which process term to execute is made non-deterministically.

- Parallel composition ($\|$)
  The parallel composition of n process terms, $P_1 \| P_2 \| \ldots \| P_n$ represents the arbitrary interleaving of the process terms $P_1, P_2, \ldots, P_n$, but also allows communication between the process terms.

- Left merge ($\lfloor\!\lfloor$)
  The left merge operator is closely related to the parallel composition. It denotes that the first action on the left-hand side of the operator is executed first, whereafter the remaining process term continues as a parallel composition. Consider the process term $(a.x) \lfloor\!\lfloor y$. This means that first action a is executed, whereafter the process term behaves as $x \| y$. This operator occurs for technical reasons in the reduction of ACP specifications and is seldomly used in modeling directly [85].

- Communication merge (|)
  This operator is used together with the communication function γ to express communication, or interaction, between two actions. The communication function expresses which actions can communicate and what the result of this communication is. For example $\gamma(give, take) = pass$ expresses that in the process term $give|take$ actions *give* and *take* communicate, resulting in action *pass*. If this communication function does not exist, actions *give* and *take* cannot communicate. This means that the process term $give|take$ results in a deadlock (0). The communication merge operator also occurs for technical reasons in the reduction of ACP specifications and is seldomly used in modeling directly [85].

- Encapsulation ∂_H
  This operator prevents actions in the encapsulation set H from being executed. Its main purpose is to enforce communication between actions in a parallel composition by preventing interleaved execution of these actions individually. The effect of the encapsulation operator is demonstrated later in this section.

In ACP, reasoning about the behavior of a system is done by means of *rewriting*. This is done by applying axioms to the process term that models a system. The entire set of axioms of ACP used for rewriting can be found in Appendix A. Rewriting changes the syntactic structure of the term, while preserving its (trace) semantics. By appropriate application of the axioms, a process term can be rewritten to its *normal form*. A term that is in its normal form no longer contains any parallelism, i.e., it is *linearized*. A linear process term consists of actions, sequential composition operators, and alternative composition operators only. In this normal form, the order in which actions are executed, i.e., the traces of the system, can easily be derived. This means that the effects of concurrency are still visible, but there is no parallel composition operator anymore, i.e., the structure has been lost. An example of the rewriting process is shown in Figure 2.1. In this example, the ACP model consisting of the process term $\partial_{\{give,take\}}(give.1 \| take.1)$ and the communication function $\gamma(give, take) = pass$ is rewritten to its normal form. This example demonstrates the effect of the encapsulation operator in combination with a parallel composition. The actions in the encapsulated part of the ACP specification is allowed, but execution of the individual actions is not.

2.2.2 Unified Modeling Language

The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system [73]. The UML version 2 provides thirteen diagram types to model the static structure and the behavior of a software system [72]. For our

$$
\begin{aligned}
& \partial_{\{give,take\}}(give.1 \| take.1) \\
= \quad & \{\text{Axiom M} : x\|y = x\lfloor\!\lfloor y + y\lfloor\!\lfloor x + x|y\} \\
& \partial_{\{give,take\}}(give.1\lfloor\!\lfloor take.1 + take.1\lfloor\!\lfloor give.1 + give.1|take.1) \\
= \quad & \{\text{Axiom LM3} : a.x\lfloor\!\lfloor y = a.(x\|y)\ (2\times)\} \\
& \partial_{\{give,take\}}(give.(1\lfloor\!\lfloor take.1) + take.(1\lfloor\!\lfloor give.1) + give.1|take.1) \\
= \quad & \{\text{Axiom D5} : \partial_H(x+y) = \partial_H(x) + \partial_H(y)\ (2\times)\} \\
& \partial_{\{give,take\}}(give.(1\lfloor\!\lfloor take.1)) + \partial_{\{give,take\}}(take.(1\lfloor\!\lfloor give.1)) \\
& +\partial_{\{give,take\}}(give.1|take.1) \\
= \quad & \{\text{Axiom D3} : \partial_H(a.x) = 0,\ \text{if } a \in H\ (2\times)\} \\
& 0 + 0 + \partial_{\{give,take\}}(give.1|take.1) \\
= \quad & \{\text{Axiom A1} : x + y = y + x\} \\
& \partial_{\{give,take\}}(give.1|take.1) + 0 + 0 \\
= \quad & \{\text{Axiom A6} : x + 0 = x\ (2\times)\} \\
& \partial_{\{give,take\}}(give.1|take.1) \\
= \quad & \{\text{Axiom CM5} : a.x|b.y = c.(x\|y),\ \text{if } \gamma(a,b) = c\} \\
& \partial_{\{give,take\}}(pass.(1\|1)) \\
= \quad & \{\text{Axiom SC2} : x\|1 = x\} \\
& \partial_{\{give,take\}}(pass.1) \\
= \quad & \{\text{Axiom D4} : \partial_H(a.x) = a.\partial_H(x),\ \text{if } a \notin H\} \\
& pass.\partial_{\{give,take\}}(1) \\
= \quad & \{\text{Axiom D1} : \partial_H(1) = 1\} \\
& pass.1
\end{aligned}
$$

Figure 2.1 Rewriting the ACP model $\partial_{\{give,take\}}(give.1\|take.1)$, $\gamma(give, take) = pass$ using the ACP axioms

purposes, we require class diagrams and state machine diagrams only. Therefore, only these two diagram types will be explained in more detail.

2.2.2.1 Class Diagrams

Class diagrams are used for modeling the static parts of a system. These are, among others, classes, types, and their interrelations. Every class has a name and may have attributes and operations. The dynamics of a class is expressed in other diagrams, such as a state machine diagram. A state machine diagram describes

the life cycle of the class from the moment it is instantiated, to the moment its execution has terminated.

2.2.2.2 State Machines

State machine diagrams contain, as their name suggest, *state machines*. State machines are hierarchical structures used for modeling the behavior of a class. Here, we consider a subset of UML state machines only, viz. without the history mechanism. The basic building blocks of state machines are states and transitions. A graphical representation of the constructs available in UML state machine diagrams is depicted in Figure 2.2. There are three types of states, viz. simple states, composite states, and pseudo states. A simple state can have an entry, internal, and exit activity. A composite state is a state that can contain other states. It is used to express hierarchy. It can have the same activities as a simple state. An entry activity is executed in its entirety upon entry of the state. An exit activity is executed in its entirety when the state is left. Internal activities are executed when the state is active and may be interrupted. Activities usually represent the operations modeled in a class.

There are six types of pseudo-states. *Start* states represent the initiation of behavior. *End* states represent the termination of a (sub) state machine. *Choice* states represent a non-deterministic choice between multiple execution paths. *Fork* states are used to split behavior into two parallel execution paths that are active simultaneously. *Junction* and *join* states are used to merge the execution paths that were split by a choice or a fork state respectively.

States are connected by transitions. A transitions can have a guard condition and an activity. A transition can be taken when its guard condition holds. When the transition is taken, the activities are executed in their entirety.

2.3 Transformation

The transformation f from ACP models to UML state machines takes an ACP model, which consists of a process term and a communication function, as an argument and returns a UML state machine.

$$f : \text{ACP model} \rightarrow \text{UML state machine}$$

Every non-atomic process term is built from smaller process terms, resulting in an implicit tree structure. The transformation traverses this tree and transforms every subtree into a partial state machine that is structurally equivalent to the process term in the node.

The axiomatic rewrite rules of ACP can be used to rewrite an ACP model into a different, but equivalent model. To ensure the required structural equivalence

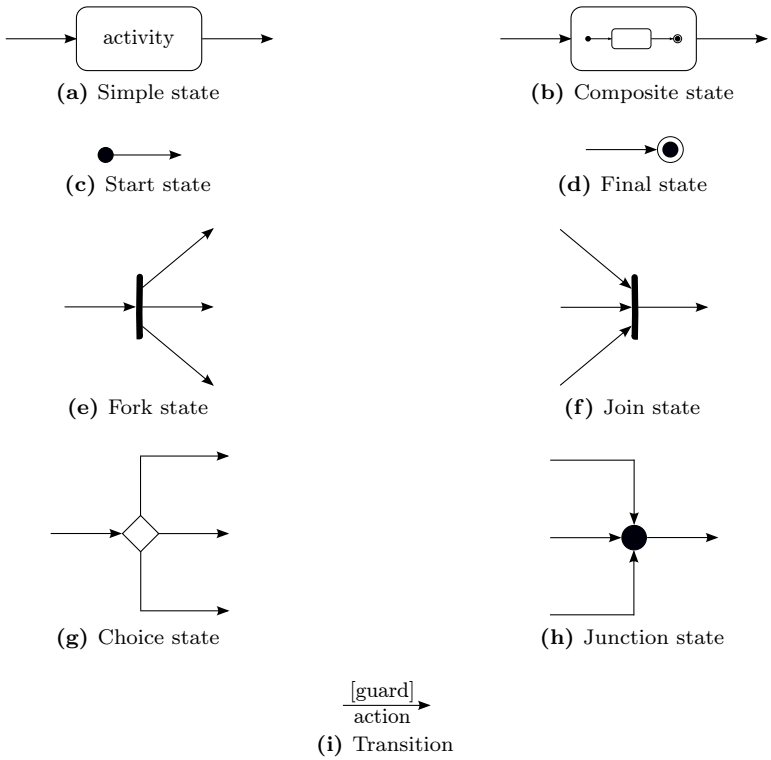


Figure 2.2 State machine constructs

with respect to parallel behavior, the original ACP model should not be rewritten to remove parallel composition operators. It is not required to maintain the structure of the ACP model with respect to the other ACP constructs. However, we will maximize structure preservation for all ACP constructs, since we want to preserve designer's choices as closely as possible. Therefore, we will avoid rewriting as much as possible.

We use the formal semantics of ACP described in [86] and the semantics description of UML presented in [72] to explain informally the behavioral equivalence of ACP constructs and the resulting UML state machines. With behavioral equivalence we mean that the state machines need to define exactly the same traces as the original ACP models.

In the remainder of this section, the transformation is explained in five steps. In each subsection, a new step which introduces one or more new ACP constructs

is discussed. Every step, together with the steps preceding it, presents a complete solution for the transformation of the ACP constructs that have been discussed until then.

2.3.1 Atoms and Sequential Composition

The transformation generates a partial state machine for every ACP construct used in the model. These partial state machines do not have a start and an end state. Instead, partial state machines are connected to each other in such a way that it corresponds to the syntax tree of the ACP model it represents. In this way, compositionality is achieved. Only the complete state machine, representing the full ACP model, has a start and an end state. Figure 2.3 depicts the state machine that is acquired when a full ACP model is transformed. The model consists of a process term P and a communication function γ. The state labeled $f(P)$ denotes the partial state machine acquired after applying the transformation to process term P.

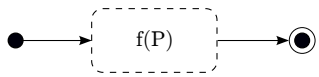


Figure 2.3 Full specification

Note that the result of the transformation of process term P will have an incoming and an outgoing transition. The outgoing transition of the start state and the incoming transition of the transformation result will merge. Also the incoming transition of the end state will merge with the outgoing transition of the transformation result. When transitions merge, the guard condition on the merged transition will be the conjunction of the guard conditions on the transitions that merge. In this case, the guard conditions on the incoming and outgoing transitions of the transformation result apply, since the outgoing transition of the start state and the incoming transition of the end state have no guard conditions.

2.3.1.1 Atoms

The atomic actions in an ACP specification represent real-world events. The transformation maps the atomic action a to the state machine depicted in Figure 2.4. The entry activity on the simple state executes the atomic action.

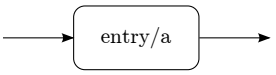


Figure 2.4 Atomic action

The empty process constant 1 represents successful termination of a process term. The transformation maps the empty process constant to a partial state machine similar to the one depicted in Figure 2.4. The only difference is that there is no entry action on the simple state. An alternative solution would have been to map the empty process constant to a final state. However, a process term can be followed by another process term, viz., using the sequential composition operator. Therefore, the state machine should continue instead of terminating by progressing into a final state. Termination of a state machine in this way can be circumvented by mapping every process term to a composite state, since a state machine will progress into the next state when a composite state has successfully terminated [87]. Although this is a valid solution, we did not opt for it because we want to have as little hierarchy as possible in order not to clutter the state machine diagrams.

The deadlock constant 0 denotes unsuccessful termination of a process term, or inaction. The transformation maps the deadlock constant to the partial state machine depicted in Figure 2.5. The guard on the outgoing transition of the simple state ensures that that state can never be exited. In this way the state machine cannot proceed from this state. Since in this section (2.3.1) we deal with atoms and sequential composition only, this results in a total deadlock.

Figure 2.5 Deadlock

2.3.1.2 Sequential Composition

Process terms composed by the sequential composition operator $\cdot$ are executed in sequence. The transformation maps the sequential composition $P_0 \cdot P_1 \cdot \ldots \cdot P_n$ to the partial state machine depicted in Figure 2.6. This construct enforces that the execution of P_i precedes the execution of P_{i+1}, for all $i < n$. States labeled $f(P_i)$ represent the partial state machines acquired after applying the transformation to process term P_i.

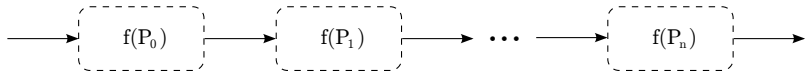


Figure 2.6 Sequential composition

2.3.2 Alternative Composition

Of the process terms composed by a alternative composition operator +, only one is executed. The choice for which process term to execute is made non-deterministically. The transformation maps the sequential composition $P_0 + P_1 + \ldots + P_n$ to the partial state machine depicted in Figure 2.7. The choice state ensures that only one of the paths will be selected for execution. States labeled $f(P_i)$ represent the partial state machines acquired after applying the transformation to process term P_i. The junction state is used to connect all partial state machines.

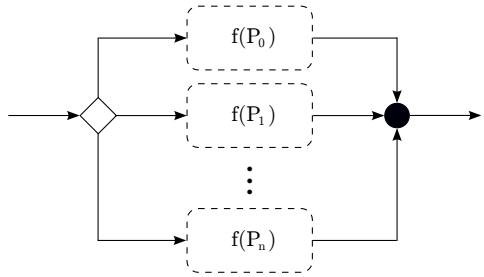


Figure 2.7 Alternative composition

2.3.3 Parallel Composition

Process terms composed by a parallel composition operator $\|$ are executed quasi-parallel and may communicate. Quasi-parallel execution means that the actions in the parallel process terms are arbitrarily interleaved whilst maintaining their internal ordering. Consider the parallel composition $(a \cdot b)\|(c \cdot d)$. The arbitrary interleaving will *never* result in a situation where the execution of action b precedes the execution of action a, or where the execution of action d precedes the execution of action c. Communication can occur when actions in the parallel process terms are executed simultaneously. An ACP specification consists of a process term and a communication function γ. This communication function specifies which actions can communicate and what the result of this communication is. The parallel composition $P_0\|P_1$ terminates when both process P_0 and process P_1 have terminated.

The transformation maps the parallel composition $P_0\|P_1\|\ldots\|P_n$ to the partial state machine depicted in Figure 2.8. The fork and join states are used to split and merge the control flow. They ensure that all state machines running in parallel will start and end simultaneously. A junction state cannot be used here instead of a join state, because it does not synchronize the branches of execution.

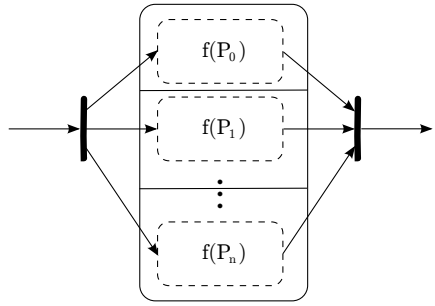


Figure 2.8 Parallel composition

2.3.3.1 Semantic Gap

The state machine construct depicted in Figure 2.8 represents arbitrary interleaving and simultaneous execution of actions. However, communication as a result of the simultaneous execution of actions in parallel branches is not represented by this construct. In UML state machines the simultaneous execution of two actions cannot result in another action being executed like in ACP. This is part of the gap between the semantics of ACP and UML state machines.

One possibility to bridge this gap is to use the ACP axioms to rewrite an ACP model such that all parallel composition operators are removed. In this way all communication is made explicit. Consider the process term $a \| b$ together with the communication function $\gamma(a, b) = c$. The state machine acquired after rewriting and transforming is sketched in Figure 2.9. This is not a valid solution since one of the requirements is that the UML state machines need to preserve the structure of the ACP models, at least with respect to the parallel composition. In fact, the combination of the requirements of preserving both behavioral and structural properties gives rise to the semantic gap. Our solution to bridge the semantic gap is presented in Section 2.3.3.2.

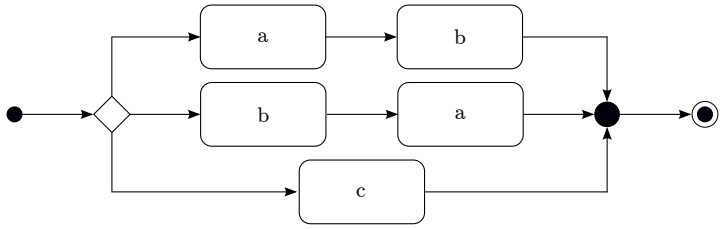


Figure 2.9 State machine representations of $a \| b$ and $\gamma(a, b) = c$ after rewriting

2.3.3.2 Action Dispatcher

To bridge the semantic gap, we chose to exploit the semantic openness of the UML. Therefore, we propose an action dispatcher that takes care of executing all actions. This solution requires a change in the way atoms are handled. This change will be explained later in this section.

Figure 2.10 depicts the class diagram representing the (single) action dispatcher. This action dispatcher object has an *action pool* of zero or more action objects. The action pool consists of all action objects ready for execution. The γ attribute of the class is the communication function γ. It is, like in ACP, used to determine whether a pair of actions can communicate. The methods of the class handle adding actions to, executing actions in, and removing actions from the action pool. The functionality of the methods is explained below. The action dispatcher is generic. This means that the same action dispatcher is generated for all ACP models. Only the γ attribute is generated from the model.

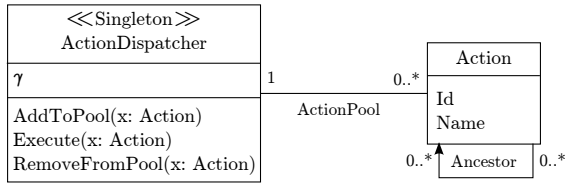


Figure 2.10 Action dispatcher class diagram

An action object has two attributes, viz., an identifier to uniquely identify the syntactic occurrence of the action and the name of the action. This name is the same name as the one occurring in the ACP process term. An action can be related to other actions, its *ancestors*. If an action x is the result of communication, e.g. $\gamma(a, b) = x$, actions a and b are considered to be *parents of* action x. The set of ancestors of x can be found by taking the transitive closure of the '*is parent of*' x relation. Note that an action that is the result of communication can communicate with other actions, e.g., $\gamma(a, b) = c$ and $\gamma(c, d) = e$. Because an action cannot communicate with its ancestors, the ancestors of an action need to be known to correctly handle communication. If an action is not the result of a communication it does not have any ancestors.

The life cycle of an action object is such that it will first be added to the action pool. After some time, it may be executed whereafter it is removed from the action pool. In case of communication, action objects can also be removed from the action pool without having been executed themselves. Action objects can even stay in the action pool forever in case of a deadlock.

Transformation of Atoms

Atoms are no longer executed in the state machine. Instead they are announced at the action pool, which ensures their execution. All atoms are treated in the same way.

The atom a maps to the state machine depicted in Figure 2.11. The entry activity on the simple state creates an action object from atom a and invokes the *AddToPool* method of the action dispatcher. For the empty process and deadlock constants also action objects will be created with names 1 and 0 respectively. The entry activity then puts the newly created action object in the action pool. To ensure that the state machine does not continue until the action has been executed, a guard is present on the outgoing transition. This guard is *true* when the action object is not in the action pool. This is the case when the action has been executed or has communicated.

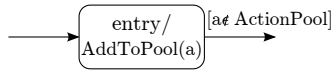


Figure 2.11 Transformation of atoms for use with the action dispatcher

Addition

The method *AddToPool(x)* is invoked by the entry activity on the simple state an atom is mapped to (see Figure 2.11). Its purpose is to extend the action pool with x and to maintain closure of the action pool under γ. If an action object x is added to the action pool, and it can communicate with another action object a already in the pool that is not one of its ancestors, then a new action object for the communication result given by γ is recursively added to the action pool (lines 3–6 in Figure 2.12). On line 5, a new action object is created by calling its constructor. This action object is assigned a new identifier by the function *NewId()* and a name provided by the communication function γ. Also, the set containing all its ancestors is assigned to the action object.

Execution

An action object x in the action pool that does not represent a deadlock constant will non-deterministically be selected at an arbitrary moment for execution. The purpose of method *Execute(x)* is to find all actions that execute along with x, i.e., all actions that, directly or indirectly, gave rise to x through communication, and to clean up the action pool. If the state machine cannot proceed and there are only action objects in the action pool that represent a deadlock constant, it is in a deadlock state.

```
1.  AddToPool(x: Action):
2.      ActionPool := ActionPool ∪ {x};
3.      ∀a : γ(a.Name, x.Name) = y
4.          → if a ∈ ActionPool ∧ a ∉ x.Ancestor
5.              → NewAction := Action(NewId(), y, a.Ancestor ∪ x.Ancestor ∪ {a, x});
6.                  AddToPool(NewAction)

7.  Execute(x: Action):
8.      ∀a : a ∈ x.Ancestor
9.          → Execute(a)
10.     RemoveFromPool(x)

11. RemoveFromPool(x: Action):
12.     ActionPool := ActionPool − {x};
13.     ∀a : a ∈ ActionPool
14.         → if x ∈ a.Ancestor
15.             → RemoveFromPool(a)
```

Figure 2.12 Pseudo code for the action dispatcher methods

Removal

The purpose of method *RemoveFromPool(x)* is to remove action object x from the action pool. To maintain closure of the action pool under γ, also all action objects that are the result, directly or indirectly, of communication involving x are removed. Note that these resulting action objects do not occur in the conditions of outgoing transitions (see Figure 2.11), because they are the result of communication.

2.3.3.3 Correctness Considerations

Interference between methods of the action dispatcher can be avoided by executing them under mutual exclusion. The action dispatcher controls the state machine through conditions of the form $a \notin \mathit{ActionPool}$ only. Note that a is added to the action pool, falsifying the condition, upon entry into the immediately preceding simple state, and is removed upon its execution, making the condition true. During execution of each method, the action pool changes monotonically to avoid undesired condition changes. This means that no conditions in the state machine are evaluated until the action pool is in a stable state, i.e., until the invoked method, either *AddToPool()* or *Execute()* has successfully terminated.

2.3.4 Left Merge and Communication Merge

2.3.4.1 Left Merge

The left merge operator $\between$ denotes that the first action on the left-hand side of the operator is executed first, whereafter the remaining process term continues as

a parallel composition. The left merge operator occurs for technical reasons in the reduction of ACP specifications and is seldomly used in modeling directly [85]. Furthermore, this operator cannot be expressed in a natural way in a state machine. It is impossible to express that a specific action in one branch of a parallel composition should be performed first. Therefore, the left merge operator is eliminated by rewriting according to the axioms of ACP.

2.3.4.2 Communication Merge

The communication merge operator | together with the communication function γ is used to express communication between two parallel processes. Like the left merge, the communication merge operator occurs for technical reasons in the reduction of ACP specifications and is seldomly used in modeling directly [85]. Therefore, the communication merge operator is eliminated by rewriting according to the axioms of ACP.

2.3.5 Encapsulation

The main purpose of the encapsulation operator is, as described in Section 2.2.1.1, to enforce communication between actions in a parallel composition by preventing the interleaved execution of these individual actions. For a set of actions H, the encapsulation operator ∂_H prevents actions in the encapsulation set H from being executed. Instead, it maps these actions to the deadlock constant. Therefore, besides explicit occurrences of deadlocks, i.e., deadlock constants in a process term, they can also occur implicitly, i.e., when an action gets encapsulated. In itself this is not an issue, because actions can be annotated such as to derive whether it is encapsulated or not. However, the combination of the encapsulation operator and the alternative composition leads to additional difficulties. In ACP, the non-deterministic choice operator is actually not completely non-deterministic. Choices that immediately lead to a deadlock will never be made. For example in the process term $a + 0$, the choice will always be to execute action a (see axiom A6 in Appendix A). Since deadlocks can occur implicitly, for example the result of a communication may be encapsulated, it is not always obvious which process terms in an alternative composition lead to a deadlock immediately. This means that in such cases, without additional bookkeeping, it is not clear which process terms in an alternative composition may be executed. Therefore, the combination of encapsulation and alternative composition leads to another semantic gap.

We have a solution to bridge this semantic gap as well. However, it is not presented here.

2.4 Implementation

Since we want to use the resulting state machines in UML tools, we defined another transformation from UML state machines to the XML Metadata Interchange (XMI) format [25]. XMI is a standard used for, amongst others, exchanging UML models between various tools. The tool Telelogic Rhapsody [88] can be used to generate simulation software from UML state machines. We use this feature to simulate the execution of process algebra specifications.

We use the term rewriting system ASF+SDF [49, 54] for the development of our metamodels[1] and for the implementation of our transformation. The transformation from ACP models to UML state machines expressed in the XMI format is too complex to implement in a single step. Therefore, we split the transformation into four independent steps. This has as additional advantage that every step is (re)usable in isolation.

In the first step of the transformation, the ACP model is rewritten using the ACP axioms to remove all instances of the left merge and communication merge operator. Consider for example the ACP process term $a \between (b|c)$ and suppose $\gamma(b, c) = d$. This rewrites to $a.d$. This step has as input and output an ACP model conforming to the ACP metamodel we defined. After this step the ACP model will only consist of constructs that have a state machine equivalent, viz., atoms, action prefixes, sequential compositions, alternative compositions, and parallel compositions. With a few extensions the transformation used in this step can also be used for rewriting ACP models to their normal form.

In the second step, the implicit tree structure of an ACP model is made explicit. For the representation of this tree structure we use an intermediate language for which we also defined a metamodel. This language uses a prefix format. Consider for example the alternative composition $P_0 + P_1 + \ldots + P_n$. The transformation function finds all the alternatives and represents them as $alt(P_0, P_1, \ldots, P_n)$.

These first two steps are mere preparation for the actual transformation. In the third step, the tree representation of an ACP model is transformed into a state machine. This state machine is defined in a state machine language for which we have also defined a metamodel. This language closely resembles UML state machines. The only difference is that it does not support the history mechanism. We chose to use this intermediate format to avoid having to transform into complex XMI constructs directly. Moreover, it enables the transformation of any UML state machine defined in our state machine language into XMI. This third transformation step is similar to Thompson's algorithm for transforming regular expressions into non-deterministic finite automata [89]. The transformation function has as arguments an ACP process term represented as a tree and a start and an end state. For the alternative and parallel composition these start state

[1] Our metamodels are in fact context-free grammars.

and end state are respectively the choice and junction state, and the fork and join state. In Figure 2.13 an example is depicted in which the partial state machines for n alternatives are generated and connected to the choice and junction states. The sequential composition needs to be handled differently since the end state of the first partial state machine in the sequence is the start state for the next. These states are not known in advance. To overcome this, dummy states are inserted such that the start and end states are known in advance. These dummy states are removed afterwards.

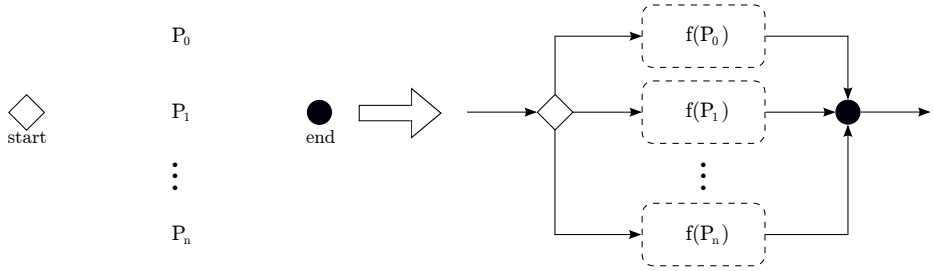


Figure 2.13 Transformation of the alternative composition

In the last step a state machine is transformed into its XMI [25] representation. This back-end part is isolated, because the XMI standard is actually not so standard. Most UML tools use a different dialect of XMI, requiring different back-ends. Currently our implementation is able to generate XMI files for the UML tools ArgoUML [90] and Telelogic Rhapsody [88]. Since there is a one-to-one mapping from UML state machine constructs to XMI, this final transformation step is straightforward. State machines can be simulated using the Telelogic Rhapsody tool. To ensure correct handling of communication, an implementation of the action dispatcher presented in Section 2.3.3.2 is added to the XMI file.

2.5 Illustration

We transformed a number of ACP models to verify the correctness of our transformation. In this section, we describe the transformation of an ACP model of a conveyor system into a UML state machine.

The conveyor system is schematically depicted in Figure 2.14. Machines M_1 and M_2 put products on a conveyor belt. The products from machine M_1 can go to machines M_3 or M_4 for further processing and the products from machine M_2 can go to machines M_4 or M_5. When products are sent to machine M_4 by both machines M_1 and M_2 at the same time a collision will occur and an operator should ensure that both products can still enter the machine for processing.

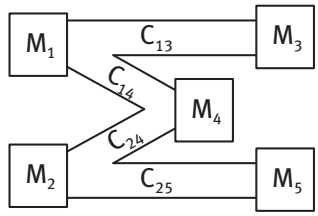


Figure 2.14 Conveyor system

The ACP model representing this system is depicted in Figure 2.15. The process term expresses that a product is produced by machine M_1 which is then sent to machine M_3 or M_4 for further processing and that another product is produced by machine M_2 which is then sent to machine M_5 or M_4 for further processing, possibly at the same time. The communication function γ expresses that an operator rearranges products that collide if two products go from machines M_1 and M_2 to machine M_4 at the same time. Note that in this example only one iteration is modeled.

$$\gamma(C_{14}, C_{24}) \quad = \quad operator$$

$$(M_1.(C_{13}.M_3 + C_{14}.M_4) \quad \| \quad M_2.(C_{25}.M_5 + C_{24}.M_4)).1$$

Figure 2.15 ACP model of the conveyor system

The state machine resulting from the transformation should exhibit the same behavior. A screen shot of the state machine acquired from the transformation imported in ArgoUML can be found in Figure 2.16.

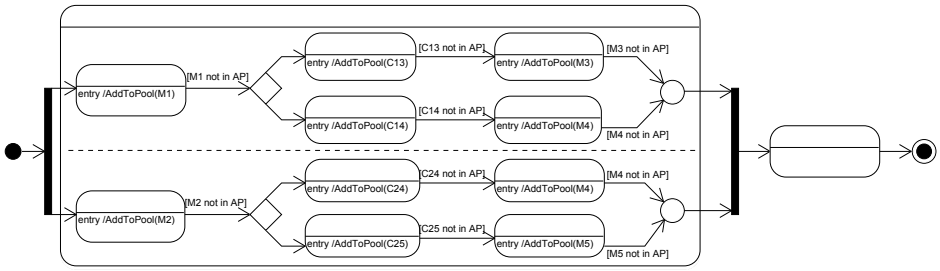
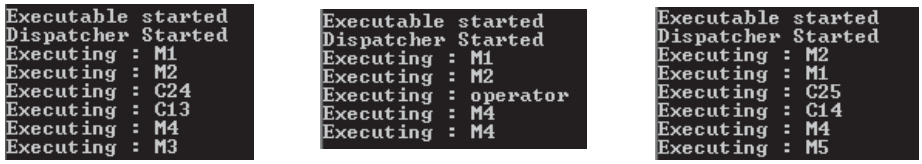


Figure 2.16 ArgoUML screen shot depicting the acquired state machine diagram

We also transformed this ACP model into an XMI file for Telelogic Rhapsody. This enables simulation of the state machine. Three screen shots showing the results of three different executions of the simulation are depicted in Figure 2.17. In trace 1 and 3 both products go to different machines. In trace 2 the operator is needed to rearrange collided products.



(a) Trace 1 **(b)** Trace 2 **(c)** Trace 3

Figure 2.17 Three Telelogic Rhapsody execution results

2.6 Related Work

In this section, we position our approach with respect to related work. First, we discuss work related to bridging a semantic gap. Several papers have been published on this topic. However, the solutions to deal with a semantic gap differ among them. Second, we discuss a number of approaches that deal with transforming process algebra models to models expressed in different other formalisms. We present here just a selection of such approaches, there are many more around.

2.6.1 Semantic Gap

Corbett et al. describe an approach for automatically extracting finite state models that can be used for verification from Java code [91]. The authors observe a semantic gap between artifacts produced by software developers and artifacts accepted by verification tools, which they refer to as the model construction problem. Their solution to this problem is twofold. First, clever abstractions should be defined to avoid semantic mismatches. Second, if these abstractions cannot be defined, the input should be bounded such that it can be transformed into the target formalism. They also address the issue of traceability, i.e., how to trace the results of a model checking process back to the original Java program. They solve this by using intermediate representations that can be used to relate nodes in the abstract syntax trees of the model used for model checking and the original Java program. We avoid the problem of traceability by preserving the structure of the process algebra model as much as possible.

An approach similar to the one presented by Corbett et al. is described by Sabetta et al. [92]. The authors aim at generating analysis models from UML models. Their solution is to aggregate model elements to raise the level of abstraction. These abstractions should semantically be closer to the target model.

Sawada defines a transformation that enables analysis of models defined in the hardware description language VHDL by the theorem prover ACL2 [93]. His first defines a transformation in the opposite direction. The semantic gap that arises by the different properties of the languages is minimized by considering only a limited form of the source language. This approach is similar to the one we adopted. We first considered a limited form of the ACP process algebra, viz., without the encapsulation operator, which led to a smaller semantic gap.

A transformation from business process models to another formalism on the same level of abstraction is described by Grangel et al. [94]. The authors do not bridge the semantic gap that arises from their transformation, instead they avoid it by adapting the target formalism. They create a UML profile that has all the features of the source language. In this way the transformation is straightforward.

Rountev et al. note that in the field of reverse engineering little work has been done on constructing reverse engineered sequence diagrams [95]. Their goal is to build a tool that can reverse engineer UML sequence diagrams from Java code through static analysis. One of the semantic gaps between the two formalisms they encounter is the way *breaks* are handled. The break fragment for UML sequence diagrams is shallow, i.e., it breaks out of the immediately surrounding fragment, whereas the break statement in Java is deep, i.e., it breaks out of several levels of nesting. Their solution to deal with this semantic gap is to augment the UML notation with a generalized break fragment that allows breaking out of multiple enclosing fragments. In this case, also, the semantic gap is not bridged, but avoided by adapting the target formalism.

Gerber et al. question how to determine the correctness of model transformations [30], both on a syntactic and on a semantic level. They argue that it is not always needed to have a complete and consistent model. This does not hold in our case. The output of our transformation should be structurally and behaviorally equivalent to the input.

2.6.2 Transforming Process Algebra

An approach for transforming the Algebra of Timed Processes (ATP) into timed graphs is described by Nicollin et al. [96]. The goal of the authors is to *unify* behavioral description formalisms for timed systems. An advantage of their transformation is that model checking can be applied to ATP specifications via timed graphs. The main difference with our approach is that they apply their transformation on a canonical form of the initial specification, whereas we preserve

the structure of the process algebra specification as much as possible.

Pardo et al. describe a transformation from a self-defined timed process algebra based on LOTOS operators to dynamic state graphs [97]. The authors have defined an operational semantics for their process algebra and proved that their transformation preserves semantics. Their goal is also to simulate execution of process algebra models. Similar to the approach described by Nicollin et al., the structure of the process algebra models is not preserved. The authors note that their approach does not scale well, but that it is suitable for smaller models.

An approach to employ process algebra for the design and verification of web services is proposed by Ferrara [98]. Thereto two model transformations are implemented, viz., from BPEL to LOTOS and the other way around. In addition, the author provides guidelines for transforming from arbitrary process algebra formalisms to BPEL.

Doxsee and Gardner note that although formal methods contribute to reliable software, they are still not widely used in industry [99]. Therefore, they advocate a technique called *selective formalism* for concurrent system design. This is a hybrid technique that incorporates both formal methods and traditional software engineering practices. One of the key components of this approach is a tool that can transform formal models into executable code. For this purpose they use a framework called CSP++. One of the components in this framework is used to transform CSPm models to C++ code.

2.7 Conclusions

In this chapter, we presented a transformation from the process algebra ACP to UML state machines. We have addressed the semantic gap that arises in this transformation. Transforming a model specified in one formalism into a model in another formalism involves more than transforming syntax. Differences in the characteristics of semantics need to be handled meticulously to ensure correctness of the transformation. In our case, a semantic gap emerged as a result of the requirements on the transformation, viz., the transformation should preserve both structural and behavioral properties. Our transformation preserves structure for all operators except for the seldomly used left merge and communication merge operators. It also preserves behavior by exploiting the semantic openness of UML state machines. We have extended UML state machines with an action dispatcher to ensure that they can generate the same execution traces as the ACP model.

Note that trace equivalence is in general only one aspect of semantic equivalence. Without providing a formal semantics for UML state machines we cannot guarantee that we have bridged the semantic gap completely. Since there are many formalisms with different (or without) formal semantics, there are probably many model transformations that are not proven to be semantics preserving. Proving that a

model transformation preserves semantics requires different expertise.

Using the tool Telelogic Rhapsody we can generate software to simulate the acquired UML state machine and action dispatcher. In this way, the execution of an ACP model can be simulated. Since our transformation preserves most structure of ACP models, UML tools can be used for visualizing this structure. We performed several case studies using our implementation to illustrate our mapping of ACP constructs to UML state machines.

Chapter 3

Quality of Model Transformations

Model transformations play a pivotal role in MDE. Consequently, they have to be treated in a similar way as traditional software artifacts. It is, therefore, necessary to define and assess their quality. We present two definitions for two different views on the quality of model transformations. We also provide some examples of quality assessment techniques for model transformations. This chapter concludes with a discussion about which type of quality assessment technique is most suitable for either of the views on model transformation quality.

3.1 Introduction

MDE is gradually being adopted by industry [56]. Since model transformations play a key role in MDE, they are becoming increasingly important as well. Model transformations are in many ways similar to traditional software artifacts, i.e., they also require maintenance and should preferably be reused.

Three categories of software maintenance activities can be distinguished, viz., adaptive, corrective, and perfective maintenance [100]. Adaptive maintenance is performed to make a model transformation usable in a changed environment. This type of maintenance occurs for example when the metamodels of the source or target language of a model transformation change and the transformation needs to co-evolve, or to add functionality to a model transformation, e.g., due to changing requirements. Corrective maintenance is performed to correct faults in

a model transformation. Perfective maintenance is performed to improve a model transformation, e.g., performance or modularity.

Models are frequently used in traditional software development. However, their applicability is typically limited to documentation and analysis. Reusing models is one of the challenges that should be solved by MDE [101]. However, reuse in MDE is not limited to models. Model transformations should be considered as reusable assets as well. A model transformation can be reused for every model adhering to the source metamodel on which it is defined. This *as-is* type of reuse is referred to in [102] for application generators, such as model transformations, in general, i.e., they are typically implemented once and reused often. Yet, model transformations can be reused in different ways as well. A model transformation that is part of a chain of model transformations, may be reused in another chain of model transformations that is defined on the same metamodels. Besides as-is reuse, *reuse-with-modify* also plays an important role in the context of model transformations. As the need for new target platforms arises, e.g., Java code is required instead of C++ code, new model transformations will have to be developed. In such cases, it is desirable to reuse as much as possible from existing model transformations.

Since model transformations are in many ways similar to traditional software artifacts, they need to adhere to similar quality standards as well. To attain these standards, a methodology for developing model transformations with high quality is required. However, before such a methodology can be developed, quality needs to be defined in the context of model transformation. Therefore, we address in this chapter research question $\mathrm{RQ}_{\mathbf{2}}$.

> **$\mathbf{RQ}_2$**: *How can quality be defined in the context of model transformations?*

The remainder of this chapter is structured as follows. In Section 3.2, we will distinguish two views on the definition of model transformation quality. Section 3.3 describes the quality attributes that we consider relevant for model transformations, in particular with respect to maintenance and reuse. In Section 3.4, we will show for both of the views on quality different ways to assess it. Section 3.5 concludes this chapter.

3.2 Internal vs. External Quality

It is often said that quality is in the eye of the beholder, i.e., it depends on the perspective and objectives of the evaluator [103]. Therefore, quality can be defined in different ways. Five approaches to the definition of quality are given by Garvin [104]. Trienekens and van Veenendaal and also Kitchenham and Pfleeger

describe these five definitions from the perspective of software development as follows [105, 106].

1. Transcendent definition:
   Quality is an ideal toward which we strive. It is easy to recognize, but hard to measure. It depends on the perceptions and affective feelings of a person or a group towards a software product.

2. Product definition:
   This view on quality concerns the inherent characteristics of a software product. Quality is based on a well-defined set of internal software quality attributes. These quality attributes can be measured in an objective way.

3. User-based definition:
   Quality is fitness for use [107]. It should be determined by the users of a software product in a specific situation. It is therefore subjective and cannot be measured in an objective way.

4. Manufacturing-based definition:
   Quality is conformance to requirements [108]. It is aimed at the development process of a software product by stating that a software product is of high quality when the developers have ensured that it does what it should do. Higher quality in this case implies less rework during development and after delivery.

5. Value-based definition:
   Quality should be determined by means of a decision process on trade-off's between performance of the product, time, effort, and cost.

For model transformations, the definition that is applicable depends on the chosen perspective. A model transformation can be considered in two different ways, viz., as a (static) transformation definition or as a (dynamic) runtime instance of that definition, i.e., the process of transforming a source model to a target model (see Figure 3.1). Accordingly, the quality of a model transformation can be considered in two different ways as well.

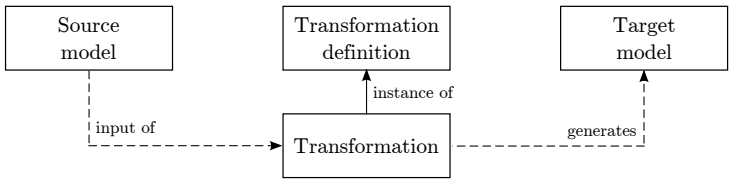


Figure 3.1 Model transformation context

When a model transformation is considered as the definition according to which a source model is transformed into a target model, the model transformation is considered as a product. Therefore, a product definition is most applicable to this view on model transformation quality. The quality of the definition is of importance during its development and maintenance and therefore most relevant to developers. Using this view on quality, the focus is on the internals of a model transformation. Therefore, this is referred to as *internal quality*. This definition is consistent with the accepted definition of internal quality that is adopted for traditional software [109].

When a model transformation is considered as the process of transforming a source model to a target model, in fact the effect of the model transformation is considered. This concerns quality aspects of a transformation that are observable from the outside, i.e., the transformation is considered as a black-box. These quality aspects include among others conformance to requirements and performance. This view on model transformation quality is primarily of importance for the users of a transformation. Therefore, a user-based definition is most applicable in this case. Using this view on quality, the focus is on the externals of a model transformation. Therefore, this is referred to as *external quality*. Again, this definition is consistent with the accepted definition of external software quality.

Besides internal and external quality, two other views on quality can be distinguished, viz., quality of the development process and quality in use [109]. However, we will not consider these views here. The research presented in this thesis mainly focuses on the internal quality of model transformations. However, in Section 3.4, we will also address issues regarding the assessment of their external quality. Note that a number of studies have been performed that confirm the intuitive observation that the internal quality of a software artifact affects its external quality. Plösch et al. report on these studies, but also on a study that criticizes the findings of these studies [110].

3.3 Quality Attributes

In the previous section, we observed that there are many faces to quality. The different characteristics that quality can be decomposed into are referred to as quality attributes. In the remainder of this section, we describe the attributes of internal quality that we consider as relevant for model transformations, in particular with respect to development and maintenance. Most of these quality attributes have already been defined earlier for software artifacts in general [109, 111–113]. We explain here why they are relevant for model transformations in particular.

Understandability

Understandability refers to the amount of effort that is required for understanding the purpose of a model transformation. High understandability of a model transformation is of crucial importance when it has to be maintained or reused. For traditional software artifacts, a significant proportion of the time required for maintenance, debugging, and reusing tasks is spent on understanding the software [114,115]. Since model transformations are similar to traditional software artifacts, we expect that this observation holds for model transformations as well.

A model transformation is a set of functions that transform metamodel elements. Therefore, the understandability of a model transformation is affected by the understandability of the involved metamodels. Their syntax and semantics need to be understood to fully comprehend the model transformation.

Modifiability

Modifiability refers to the amount of effort required for adapting a model transformation to provide different or additional functionality. A model transformation may have to be changed for various reasons, viz., to remove faults, i.e., corrective maintenance, to accommodate metamodel changes , i.e., adaptive maintenance, or to adhere to changed requirements, i.e., perfective maintenance. Modifiability is closely related to understandability, since typically a modification of a model transformation requires the model transformation to be understood.

Reusability

Reusability refers to the extent in which parts of a model transformation can be reused by other (related) model transformations. Reuse is one of the fundamental points of MDE. Therefore, we argued in Section 3.1 that model transformations should be considered as reusable assets as well.

Reuse is closely related to maintenance. Therefore, reusability is closely related to understandability as well. A model transformation needs to be understood in order to evaluate whether (parts of) it can be reused for another model transformation. In Section 3.1, we noted that a model transformation may need modifications in order for it to be reusable. Therefore, reusability is related to modifiability as well.

Modularity

Modularity refers to the extent in which a model transformation is systematically separated and structured. Most model transformation languages have support for structuring model transformations by packaging transformation rules into modules [116]. Proper modularization has a number of advantages [117]. First, every

module in a model transformation can be developed separately. Second, a module can be modified without affecting the other modules in a model transformation. Last, modules should be sufficiently small to be understood in isolation which benefits the overall understanding of a model transformation. Since modularity benefits understandability, it also benefits modifiability and reusability.

Completeness

Completeness refers to the extent in which a model transformation is fully developed. A model transformation is fully developed when it fulfills its requirements, i.e., when it correctly transforms models conforming to its source metamodel to models conforming to its target metamodel.

This is the only quality attribute we list in this section that addresses both internal and external quality. Completeness is defined as conformance to requirements. Conformance to user requirements is an attribute of external quality since this is of interest to the users of a model transformation. Conformance to software requirements is an attribute of internal quality since this is of interest to the developers of a model transformation.

Consistency

Consistency refers to the extent in which a model transformation is implemented in a uniform manner. A model transformation is consistent if a uniform programming style is used throughout the transformation. Programming style has many facets [118]. Here, we focus mainly on notation, i.e., the terminology and symbology used in a transformation. This definition of consistency is similar to the definition of internal consistency of Boehm et al. [112]. They also define external consistency, which is similar to our definition of completeness.

There is a wide range of causes for inconsistencies [119]. Our definition of consistency refers to programming style. The most prominent cause for inconsistency of this type is the multiplicity of developers involved in the development process. Consistency is related to understandability, since typically the use of uniform notation increases understandability [112].

Conciseness

Conciseness refers to the extent in which a model transformation is free of superfluous elements. Superfluous elements include variables that are never used and transformation functions that are never invoked. Conciseness may conflict with understandability, since a transformation function that is written using less elements may be more difficult to comprehend.

3.4 Quality Assessment of Model Transformations

In Section 3.2, we distinguished two views on model transformation quality, viz., internal and external quality. Similarly, two methods for assessing quality (both internal and external) can be distinguished. On one hand, a model transformation can be treated as a white-box. In this case, measurements can be performed on the transformation definition. Since in this approach the transformation definition is assessed directly, we refer to this type of quality assessment as *direct quality assessment*. The quality Q of a model transformation $\mathcal{MT}$ can thus be expressed as a function of the transformation definition, i.e., $Q = f(\mathcal{MT})$ for some function f. On the other hand, a model transformation can be treated as a black-box. In this case, it is not possible to perform measurements on the definition, but only on its environment. Since in this approach the effect of a model transformation is assessed and not the transformation definition itself, we refer to this type of quality assessment as *indirect quality assessment*. One method to assess the quality of a model transformation indirectly is by measuring a source model and the corresponding target model and comparing the results of these measurements. In this case, quality Q can be expressed as $Q = \bigoplus(f(M), g(M'))$, where $\bigoplus$ is a comparison operator and functions f and g are used to assess source model M and target model M' respectively. Another method to assess the quality of a model transformation indirectly is by calculating the differences between the source and the corresponding target model and measuring these differences. Obviously, it should be possible to calculate these differences, for example using the techniques described by Protić [120]. Therefore, this method is only applicable when the source and target metamodel of a transformation is the same, i.e., in case of an endogenous model transformations [21]. In this case, quality Q can be expressed as $Q = f(\Delta(M, M'))$.

Intuitively, it is to be expected that direct quality assessment is most appropriate for assessing the internal quality of a model transformation, since the subject under evaluation is the one from which the quality should be assessed. Likewise, indirect quality assessment seems to be most appropriate for assessing the external quality of a model transformation. In the remainder of this section, we will provide examples of direct and indirect quality assessment. These examples will show that direct quality assessment can be used for assessing external quality as well.

3.4.1 Direct Quality Assessment

When applying direct quality assessment to model transformations, the model transformation itself is being measured. The source and target model of the transformation are not considered in the evaluation. In this thesis, we mainly focus on assessing the internal quality of model transformations directly using metrics. In Chapters 4, 5, and 6, five sets of metrics are presented for measuring model

transformations developed with five different model transformation languages. Since different languages have different characteristics, a metric set is specific for a model transformation language. In Chapter 6, we will show that conceptually similar metrics can be defined for different model transformation languages. For two of the metric sets, viz., for ASF+SDF and ATL, we report on an empirical study aimed at determining whether the metrics can be used for assessing the quality attributes presented in Section 3.3. For both languages, significant correlations were found between a number of metrics and the quality attributes.

Although metrics for model transformations do not measure the process of transforming a source model to a target model, they are applicable for determining certain characteristics of this process. This is not unexpected, since the implementation, i.e., the transformation definition, prescribes how the process should flow. An external quality attribute that can be assessed by means of direct quality assessment of model transformations is performance. We performed experiments with the model transformation languages ATL and QVTO to determine what factors influence the execution time of a model transformation [121]. One of the experiments was aimed at evaluating the influence of the chosen implementation strategy on performance. The implementation strategy that is chosen for a model transformation is reflected in the values of a number of metrics. We performed regression analyses to determine whether a relation exists between these metrics and the execution time of a model transformation. Such relations were found, however, because of the limited number of subjects in the study, they are insufficiently significant. Although more research into this relation is required, the preliminary results we acquired do confirm our expectation.

For other external quality attributes, such as completeness, direct assessment is less applicable. Only very general observations can be made. Consider a model transformation that adds accessor methods to a class diagram. The number of model elements in a target model should be larger than the number of model elements in the source model. In Section 5.3, we will show a metric for ATL that measures how much a transformation function increases or decreases the number of model elements in the target model. If there is no transformation function that increases the number of model elements, this may imply that there is a fault in the transformation. Note, however, that such an observation should be treated merely as an indication and no far-reaching conclusions should be drawn.

3.4.2 Indirect Quality Assessment

Software metrics have been studied extensively over the last decades [59]. Metrics have been proposed for measuring various kinds of software artifacts, e.g., object-oriented programs [122], UML models [123], and process models [124]. Such metric sets can be used for measuring and comparing the source and corresponding target

model of a model transformation. The result of such a comparison reflects the effect of a model transformation. This result can be used to judge how well the model transformation adheres to its requirements, i.e., how complete it is. Note, however, that the extracted metric values may be incomparable, in particular when the source and target model of a transformation are not defined in the same language, i.e., in case of an exogenous model transformation [21]. Therefore, care has to be taken to prevent comparing apples and oranges. Note also that, in general, indirect quality assessment is case specific, i.e., the outcome will be different for each pair of source model and corresponding target model. It may therefore be hard to draw general conclusions about the external quality of a model transformation based on a limited number of observations.

Saeki and Kaiya advocate indirect quality assessment for assessing the external quality of model transformations [125]. They propose to extend the source and target metamodels on which a model transformation is defined with metrics and methods for their calculation. These metrics can then be used to assess the quality change of a model induced by a model transformation.

Differences between the source and corresponding target model of a model transformation can be calculated by model comparison tools [120, 126]. These differences represent the changes induced on the source model by the model transformation. They can be used to assess whether the changes to the source model are as expected, and thereby whether the model transformation adheres to its requirements. Note, however, that a discrepancy between the expected changes and actual changes may also indicate a flaw in the source model. For this method of indirect quality assessment, a subset of the code churn metrics as defined by Nagappan and Ball can be used [127].

Indirect quality assessment using metrics seems to be inadequate for assessing the internal quality attributes for model transformations presented in Section 3.3. The reason for this is that metrics that can be derived from models have no relation at all with any of the internal quality attributes for model transformations.

3.4.2.1 Validation of Model Transformations

In the previous sections, we presented approaches based on metrics. There are also approaches that do not use metrics that can be used for assessing the quality of model transformations indirectly. Model checking can be used to validate whether model transformations preserve certain behavioral properties [128]. This type of validation can be used to acquire insight in the completeness of model transformations. A schematic overview of the approach is depicted in Figure 3.2. A source model M and the resulting target model M' of a model transformation t are transformed by model transformations t_1 and t_2 into models that can be used for model checking, M_{mc} and M'_{mc} respectively. For the transformed source model M_{mc}, a property p is defined that it should satisfy. A model checker can be

used to verify the validity of property p. This property is then transformed using another transformation t_p into an equivalent property p' that the transformed target model M'_{mc} should satisfy. Again, a model checker can be used to verify whether model M'_{mc} in fact satisfies property p'. If this is not the case, it can be concluded that model transformation t does not maintain the validity of property p and may, therefore, be defective.

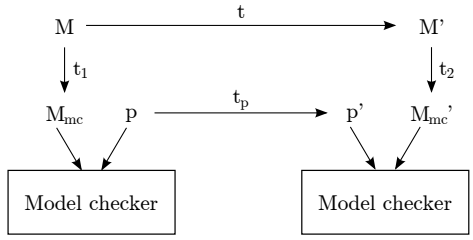


Figure 3.2 Model checking model transformations

There are, however, a few threats to the validity of this approach. The validity of the approach is affected by three model transformations, viz., model transformations t_1, t_2, and t_p. When model transformation t is an endogenous model transformation, the same model transformation can be used for generating models M_{mc} and M'_{mc}, i.e., $t_1 = t_2$. Since the technique is aimed at validating behavioral equivalence, the property that needs to be verified can remain the same in most cases, i.e., $p = p'$. Therefore, the validity of the approach, when applied to endogenous model transformations, is affected by one model transformation only, i.e., the one that transforms the source and target model of model transformation t into the models suitable for model checking.

The approach presented here is similar to the one presented by Varró and Pataricza [129]. They also present solutions to some of the threats to validity. To ensure the correctness of transformations t_1, and t_2, they define the operational semantics of the source and target language of the transformation using graph transformation rules. They use the algorithm defined by Varró to automatically generate transition systems from the source and the target model that are suitable for model checking [130]. To ensure the validity of transformation t_p, they state that a domain expert should be consulted to verify that the generated property $p\prime$ is correct.

Note that verifying whether properties are maintained by a model transformation does not guarantee the completeness of that model transformation. It can be used to validate whether the model transformation maintains the behavioral properties that have been verified, and for the models against which they have been verified only. Therefore, the strength of this method depends on the number of realistic models and properties that are verified. To guarantee completeness of

model transformations, other techniques should be used. One such approach is described by Giese et al. [131]. In their approach, a theorem prover is used to ensure semantic equivalence between a model and the code generated from it by a model transformation.

3.5 Conclusions

In this chapter, we have addressed the need for techniques to analyze the quality of model transformations. We gave two definitions for model transformation quality, viz., internal quality and external quality. We also defined and gave examples of two different types of quality assessment techniques for model transformations, viz., direct assessment and indirect assessment.

Direct quality assessment is specific for a model transformation language and not for a pair of metamodels. For assessing the internal quality attributes presented in Section 3.3 it is most appropriate. In Chapters 4 and 5, we will present two studies aimed at establishing a relation between metrics for model transformations and the internal quality attributes. Although the metamodels of the source and target models of a transformation are not taken into account, it can be used for assessing the external quality attribute performance. Indirect quality assessment may be used for assessing the external quality of a model transformation, but only if comparison of (metrics collected from) the source and the target models is possible. It should not be used for assessing the internal quality of a model transformation. At least not using metrics, since metrics that are used for evaluating a model are unrelated to any of the quality attributes for model transformations. Applying model checking to a source model and corresponding target model of a model transformation to validate whether it maintains a behavioral property seems a valid approach for indirectly assessing the external quality of a model transformation. Note that indirect quality assessment provides a partial conclusion only, i.e., it is as good as the amount of cases that have been assessed. Indirect quality assessment is related to testing, since it requires executing a model transformation. Table 3.1 summarizes the conclusions.

Quality type	Assessment technique	
	Direct	Indirect
Internal	Yes see Chapters 4 and 5	No
External	some quality attributes only, e.g., performance	if M and M' are comparable

Table 3.1 Types of quality and quality assessment techniques

Chapter 4

Quality Assessment of ASF+SDF Model Transformations

Since model transformations are becoming increasingly important, there is a need for a methodology to assess their quality. Software metrics have been successfully employed for assessing the quality of various kinds of software artifacts. Therefore, we present in this chapter a set of 28 metrics for assessing the quality of model transformations developed with ASF+SDF. Metrics alone, however, do not suffice for evaluating quality. Therefore, we conducted an empirical study aimed at determining whether the metrics we defined are valid predictors for the quality attributes presented in Chapter 3. Based on the results of this empirical study, we identified a set of predicting metrics for the quality attributes for model transformations developed with ASF+SDF.

4.1 Introduction

In Chapter 3, we indicated that model transformations need to adhere to quality standards similar to those of traditional software artifacts. Therefore, there should be a methodology for assessing their quality. An approach that is frequently applied for assessing the quality of various kinds of software artifacts is the application of software metrics [59]. Although it has been recognized that metrics should be proposed for measuring model transformations as well [101], little

research has been performed in this area. Therefore, we address in this chapter research question RQ_3.

> **RQ_3**: *How can metrics be used to assess the quality of model transformations?*

In this chapter, we focus on direct assessment of the quality of model transformations developed with the ASF+SDF term rewriting system [49]. We defined 28 metrics for measuring ASF+SDF model transformations. While, these metrics are specific for ASF+SDF, in Chapter 6 we will show that conceptually similar metrics can be defined for different model transformation languages as well. Having metrics alone does not suffice for assessing quality. A relation should be established between the metrics and quality attributes the metrics intend to assess. To establish this relation, we conducted an empirical study. Metric values were collected from six ASF+SDF transformations by a tool we created. The quality, in terms of the quality attributes presented in Section 3.3, of the same transformations has been assessed by ASF+SDF experts manually. We analyzed the correlations between the metrics data and the expert data to explore the relations between the metrics and the quality attributes. In this way we could assess whether the automatically collected metrics are appropriate for assessing the quality attributes.

The remainder of this chapter is structured as follows. A short introduction to ASF+SDF is given in Section 4.2. In Section 4.3, the metrics we defined for analyzing ASF+SDF transformations are described. The tool we created for automatically extracting these metrics from ASF+SDF transformations is described in Section 4.4. In Section 4.5, we report on the empirical study we conducted. Related work is described in Section 4.6. Section 4.7 concludes this chapter and is used to reflect on research question RQ_3.

4.2 ASF+SDF

In this chapter, we consider quality attributes of model transformations defined using the term rewriting system ASF+SDF. One of the main applications of ASF+SDF is transformations between languages. These transformations are performed between languages specified in the syntax definition formalism (SDF) [52]. Transformations in ASF+SDF are implemented as conditional rewrite rules specified in the algebraic specification formalism (ASF) [53].

SDF is used to define context-free grammars that represent the concrete syntax of a (modeling) language. In the context of MDE, a grammar represents a metamodel. Such a grammar is used to generate a parser for the language described by that grammar. This parser can be used to generate concrete syntax parse trees from input models that adhere to the grammar. A rewrite rule,

specified in ASF, is applied to a node in such a parse tree, resulting in a new parse tree node. The new parse tree node is part of the parse tree of the output model. Rewrite rules are based on the concrete syntax of the source and target language. The parse tree representing the output model can be unparsed to acquire the output model. Since rewrite rules generate parse tree nodes that adhere to the grammar of the target language, syntactical correctness of target models is guaranteed.

A model transformation in ASF+SDF consists of multiple transformation functions. A transformation function is defined by one or more signatures and implemented by one or more equations. The signatures of a transformation function consist of the name of the transformation function, followed by a list of arguments and a return value. Signatures are defined in SDF. Apart from function signatures, SDF is also used to define variables and their types that can be used in the rewrite rules. Variables are usually defined in the `hiddens` section of an SDF module, which means that they can only be used in the module they are defined in. It is possible to define an unlimited number of variables in ASF+SDF using the Kleene star operator (*) [132]. An example showing two signatures and two variable definitions in SDF is depicted in Listing 4.1. In this example there are two signatures for one transformation function, i.e., the transformation function is overloaded. The first signature defines that the function `transform` accepts one attribute as argument and returns a list of attributes. The second signature defines that the function `transform` accepts two attribute as arguments and returns a list of attributes. Consider the variable definition `"$Attribute"[0-9]*`. This variable is of type `Attribute`, which is specified after the `->`-sign. Since the Kleene star operator is applied to the `[0-9]` part, `$Attribute` can be postfixed with any number of digits. In this way, an unlimited number of instances of variables of type `Attribute` are defined.

```
1  exports
2   context-free syntax
3
4    transform(Attribute)            -> List[[Attribute]]
5    transform(Attribute, Attribute) -> List[[Attribute]]
6
7  hiddens
8   variables
9
10   "$Attribute"[0-9]*    -> Attribute
11   "$Return_value"[0-9]* -> List[[Attribute]]
```

Listing 4.1 Function signature and variable definition in SDF

Transformation functions are implemented by one or more equations. Equations are conditional rewrite rules defined using ASF. The equations implementing

a transformation function have to adhere to the signature of the transformation function that has been defined using SDF. An equation can have zero or more conditions. Two types of conditions can be distinguished, viz., (in)equality conditions and matching conditions. An (in)equality condition is used to restrict the applicability of the equation, they can be used to implement case distinctions when an equation is overloaded. A matching condition is typically used to assign values on the righthand side of a condition to variables on the lefthand side of that condition. Upon evaluation of a transformation function, all alternatives, i.e., all equations implementing the transformation function, are evaluated in arbitrary order until one of them is applied or none of them can be applied. An equation is applied if all of its conditions, and the equation itself can be successfully evaluated. The conditions are evaluated in order, i.e., if one fails, evaluation stops and the equation is not applied. Two examples of implementations of a transformation function with one condition in ASF are depicted in Listing 4.2. The equations adhere to the signatures depicted in Listing 4.1. The variables used in the equations also adhere to their definitions depicted in Listing 4.1. An equation has the following form. First there is a label (lines 3 and 8). Then there are one or more conditions (lines 4 and 9). Conditions typically consist of a lefthand side, a match symbol (`:=`) or an (in)equality symbol (== or `!=`), and a righthand side. After the condition there is a separator symbol (`====>`) to separate an equation from its conditions (lines 5 and 10). An equation consists of a lefthand side, an =-symbol, and a right-hand side. The equation labeled `[equation-1]` takes one attribute as argument and returns a list with that attribute as only element. The equation labeled `[equation-2]` takes two attributes as arguments and returns a list with those attributes as elements. When the transform function is applied to a single attribute, only the first equation can be applied, since it is the only one that accepts one attribute as argument. Upon evaluation, the attribute is stored in the `$Attribute` variable. Then the first condition is evaluated (line 4). This condition creates a list containing the attribute, by putting it between square brackets, and assigns this list to the variable `$Return_value`. Recall that the type of the variable `$Return_value` is a list of attributes (see Listing 4.1, line 11). Finally, the value of the variable `$Return_value` is returned as a result.

Recall that a transformation function in ASF+SDF is applied to a node in a parse tree. Navigation over a parse tree has to be defined explicitly in a transformation. This means that to apply a function at a certain node, a number of auxiliary functions are needed to descent to this node. Traversal functions provide a shortcut to this. A traversal function has a start point and an end point. The start point is the node at which it should be invoked. The end point is the node that should be rewritten. For the start point, only a signature needs to be defined. For the end point, a signature needs to be defined as well as an equation that specifies how the node should be rewritten. The parse tree traversal from

```
equations

 [equation-1]
   $Return_value := [$Attribute]
   ====>
   transform($Attribute) = $Return_value

 [equation-2]
   $Return_value := [$Attribute1, $Attribute2]
   ====>
   transform($Attribute1, $Attribute2) = $Return_value
```

Listing 4.2 Function implementation in ASF (equations)

start point to end point is performed automatically, so no auxiliary functions are needed to descent to the end-point. For more details on and examples of the traversal functions in ASF+SDF, the reader is referred to [133].

4.3 Metrics

This section describes the 28 metrics we defined for assessing the quality of ASF+SDF model transformations. The metrics are divided into three categories, viz., transformation function metrics, module metrics, and consistency metrics. In the remainder of this section, we will address each of these categories and elaborate on the metrics belonging to them. An overview of all the metrics can be found in Tables 4.1, 4.2, and 4.3.

4.3.1 Transformation Function Metrics

A measure for the size of a model transformation is the number of transformation functions it encompasses. In Section 4.2, we noted that a transformation function in ASF+SDF consists of one or more signatures and one or more equations. Therefore, we define the metric *number of transformation functions* as the number of signatures that are implemented by at least one equation. Since transformation functions may be defined by more than one signature and may be implemented by more than one equation, a transformation function may have multiple variants. The size of a model transformation is affected by the number of variants of transformation functions. To measure the number of variants of a transformation function, we propose the metrics *number of signatures per transformation function* and *number of equations per transformation function*. Equations may have conditions. We measure the size of an equation as the *number of conditions* it has. The total size of the implementation of a transformation function is the sum of the sizes of all variants. This is measured by the metric *number of equations and conditions per transformation function*.

The complexity of a transformation function can be measured by the number of arguments it takes and by the number of values it returns. In ASF+SDF, a transformation function can be overloaded by defining multiple signatures and equations for it. These signatures may have different arguments. We propose to measure the number of arguments per transformation function. This metric is called *val-in*. A transformation function in ASF+SDF can return only one value. Therefore, there is no need to measure the number of return values of a transformation function, i.e., val-out. However, different signatures of an overloaded transformation function may return values of different types. Therefore, we measure the *number of distinct return types per transformation function*.

Transformation functions generally depend on other transformation functions. To measure this dependency, we measure *fan-in* and *fan-out* of transformation functions in a similar way as is done for traditional software artifacts [134]. Fan-in of a transformation function f is the number of times f is invoked by an another transformation function f'. Fan-out of a transformation function f is the number of times f invokes another transformation function f'. Note that recursive function calls are not taken into consideration when calculating fan-in and fan-out. Related to these dependency metrics are Henry and Kafura's complexity metric [134] and Shepperd's information-flow metric [135]. Although we did not do so, these metrics can be defined for ASF+SDF as well.

In ASF+SDF, there are a few mechanisms to influence the flow of control of the transformation engine. These are conditions, default equations and traversal functions. Two types of conditions can be distinguished, viz., matching conditions and (in)equality conditions. We measure how often the different condition types are used, by measuring the *number of matching conditions per equation* and the *number of (in)equality conditions per equation*. The number of matching conditions is of particular interest. It is possible to write equations that express the same in different ways. One can either write relatively small equations with a relatively large number of matching conditions, or relatively large equations with relatively few matching conditions.

Equations can be marked as default by prefixing its label with the string `default-`. Upon evaluation, default equations are always evaluated last. This means that in case both a default and a non-default equation can be applied, the non-default one will be applied. Since it is not possible to influence the order of evaluation of equations in another way, it may be the case that transformation functions without a default equation may not rewrite properly. It is possible to define multiple default equations per transformation function. We measure the *number of default equations per transformation function* because it may indicate the completeness of a function. Note that ASF+SDF has built-in facilities for determining the completeness of a transformation function. When a transformation function is annotated with the `complete` annotation, an error message will be

generated if none of the equations that implement the function can be applied to the arguments of the function.

Traversal functions can also be used to change the way evaluation of a transformation function is performed [133]. A traversal function visits every node of a tree once, whereas a standard transformation function is applied to one node only. In this way, traversal functions allow a collapse of the number of transformation functions corresponding to a syntax directed translation scheme. Therefore, we distinguish *traversal functions* when measuring the number of transformation functions. In other transformation languages, different mechanisms are used to influence the transformation engine. For example, ATL distinguishes between matched rules and lazy matched rules (see Section 5.2). A matched rule is applied only once to a model element, whereas a lazy matched rule is applied as often as it is invoked.

Table 4.1 shows all the transformation function metrics we defined.

No.	Metric
1.	Number of transformation functions
2.	Number of traversal functions
3.	Number of signatures per transformation function
4.	Number of equations per transformation function
5.	Number of matching conditions per equation
6.	Number of (in)equality conditions per equation
7.	Number of conditions per equation
8.	Number of equations and conditions per transformation function
9.	Number of default equations per transformation function
10.	Transformation function val-in
11.	Transformation function fan-in
12.	Transformation function fan-out
13.	Number of distinct return types per transformation function

Table 4.1 Transformation function metrics

4.3.2 Module Metrics

Most model transformation languages have support for structuring model transformations by packaging transformation rules into modules [116]. This is also the case for ASF+SDF. A module in ASF+SDF consists of an SDF file and optionally an ASF file with the same name.

The *number of modules* is another metric that can be used for measuring the size of a model transformation. The size of an individual module can be measured in different ways. We introduce three metrics to measure the size of a module, viz., the *number of transformation functions per module*, the *number of signatures per*

module, and the *number of equations per module*. These metrics can be compared with the mean values over all modules to assess the balance of a module with respect to the rest of the model transformation.

The dependency of a module on other modules and vice versa can be measured on a module level. A module m depends on another module m' if module m imports module m'. To measure this type of dependency between modules, we measure the *number of import declarations per module* and the *number of times a module is imported* by other modules.

Dependencies between modules can also be measured on the level of transformation functions. Transformation functions may invoke transformation functions defined in other modules. To measure this type of dependency between modules, we measure *fan-in* and *fan-out* for modules. Fan-in of a module m is the number of times a transformation function defined in module m is invoked by a transformation function defined in another module m'. Fan-out of a module m is the number of times a transformation function defined in module m invokes a transformation function defined in another module m'. The metrics *module fan-in* and *module fan-out* are similar to Martin's metrics for measuring afferent couplings and efferent couplings respectively [136]. These metrics are also related to Marchesi's package metrics he defined for the UML [137].

ASF+SDF allows creating parameterized modules. A parameterized module is similar to a generic class in C++. Examples of parameterized modules in ASF+SDF are the container modules, such as list and table. These are generic lists and tables that can be parameterized such that they can contain elements of any type. Parameterized modules increase reusability. Therefore, we measure the *number of parameterized modules* in a transformation.

Table 4.2 shows all the module metrics we defined.

No.	Metric
14.	Number of modules
15.	Number of transformation functions per module
16.	Number of signatures per module
17.	Number of equations per module
18.	Number of parameterized modules
19.	Number of import declarations per module
20.	Number of times a module is imported
21.	Module fan-in
22.	Module fan-out

Table 4.2 Module metrics

4.3.3 Consistency Metrics

Recall that a transformation function consists of signatures and equations. Each signature is related to one or more equations. A signature may have no related equations. This can, for instance, occur when a transformation is still under development. To detect this inconsistency, we measure the *number of signatures without equations*. An equation that is not related to a signature will result in an error message being generated by ASF+SDF. Therefore, we do not measure this.

Variables are usually defined in a `hiddens` section. This means that they can only be used in the module they are defined in. Therefore, a variable needs to be redefined if it is to be used in other modules. This may lead to inconsistencies in variable naming, i.e., a variable name in one module can be related to a different type in another module, or vice versa. It may also cause (re)definition of variables that are not used in a module. To detect these inconsistencies, we measure the *number of (distinct) variable names per type*, the *number of types per variable name* and the *number of unused variables*. A variable is unused in a module if there are no instances of it used in the module it is defined in.

A start-symbol defines a starting point of a transformation. If only a part of a transformation needs to be used, for instance during testing and debugging, additional start-symbol may have to be defined. Usually, a transformation has only one starting point and thus requires only one start-symbol. The presence of multiple start-symbols may indicate that some are obsolete. Therefore, we measure the *number of start-symbols*.

Table 4.3 shows all the consistency metrics we defined.

No.	Metric
23.	Number of signatures without equations
24.	Number of variable names per type
25.	Number of distinct variable names per type
26.	Number of types per variable name
27.	Number of unused variables
28.	Number of start-symbols

Table 4.3 Consistency metrics

4.4 Tool

We implemented a tool that enables automatic collection of the metrics presented in Section 4.3 from ASF+SDF specifications. Figure 4.1 shows the architecture of the tool. The tool is divided into two parts, viz., a front-end and a back-end. The front-end is a C program that reads an ASF+SDF specification and uses

the ASF+SDF API to extract facts from the specification that are required to perform the measurements. The API is generated from the ASF and SDF language definitions (metamodels) by the ApiGen tool [138]. The extracted facts are stored in a relational database. The database management system used is SQLite [139]. The back-end is a Java program that uses the facts stored in the database to calculate the values of the metrics. These metrics are presented in an HTML and a CSV file. The decoupling of front-end and back-end enables an easy extension of the metrics, i.e., it only requires changing the back-end.

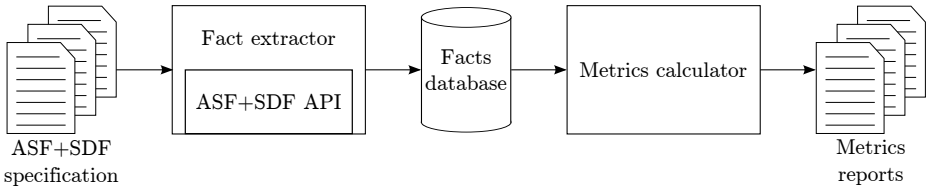


Figure 4.1 Tool architecture

Library modules can have great effects on the analysis results. For example, ASF+SDF comes with a library module for arithmetic operations on integers that contains over 350 equations. This number is far larger than the typical number of equations in a module. A metric such as *number of equations per module* is affected by this in such a way that the value for it does not provide an accurate reflection of the model transformation under analysis. Therefore, the tool produces two analysis reports, viz., one that includes library modules and one that does not. These two reports can be used to analyze the influence of the domain-independent (library) part of the transformation on the results.

4.5 Empirical Study

The presented metrics can be collected in a fast and repeatable way using our metrics collection tool. The quality attributes that are relevant for the evaluation of model transformations in practice are not directly measurable. Therefore, we are interested in the relation between metrics and quality attributes. The purpose of the case study described in this section is to explore the relation between the metrics and the quality attributes. In the case study, we used six model transformations specified in ASF+SDF. For each of the transformations we collected metrics data using the metrics collection tool described in Section 4.4. To evaluate the quality attributes for each of the transformations, we used a questionnaire that was completed by four experts in ASF+SDF. In this section we describe the design of the case study and the statistical analysis and interpretation of the collected data.

4.5.1 Objects

The experimental objects are six model transformations specified using ASF+SDF. To diversify the object set, we selected transformations created by different developers in different research projects. The transformations differ in size, style, structure and functionality.

The first transformation is used to transform ACP process algebra models into UML state machines. This transformation is discussed in Chapter 2 and in [50]. The second transformation generates a UML activity diagram from a UML activity diagram that is annotated with surface language elements. This surface language provides a shorthand textual notation for elaborate graphical structures required in UML activities, e.g., for assignments of values to variables. A number of static semantic checks of surface language statements are performed by the third transformation. The fourth transformation is used to generate a visual representation from UML activity diagrams. It takes an XMI file representing an activity and generates a file that serves as input for the visualization engine Dot [140]. All three transformations have been used in a research project [51]. The fifth transformation is part of the ASF+SDF compiler. It transforms ASF equations into C code [141]. The last transformation is a template engine, named Repleo. Repleo fills a template with input data while guaranteeing syntax correctness of the result. Repleo has been developed as part of a research project [142]. Table 4.4 summarizes the characteristics of the transformations.

Transformation	LOC	# Funcs.	Purpose	Ref.
ACP2UML	5694	173	Transform process algebra models into UML	[50]
SL2XMI	1851	70	Transform surface language into activities	[51]
SLCheck	1430	58	Surface language wellformedness checker	[51]
UML2Dot	1553	28	Transform UML activities to the input language of the visualization engine Dot	[51]
ASF2C	7096	396	Generate C code from ASF specifications	[141]
Repleo	4058	47	Syntax-safe template engine	[142]

Table 4.4 Characteristics of the analyzed model transformations

4.5.2 Participants

The participants in the study were four experienced users of ASF+SDF. All participants are researchers who have developed several ASF+SDF transformations and some of them are involved in teaching ASF+SDF. None of the authors of the article this chapter is based on participated as participant in this study. The participants were not informed about the purpose of the study on beforehand.

4.5.3 Task

The task of the participants was to quantitatively evaluate the quality of the ASF+SDF transformations in the object set. They were requested not to consider the standard library modules provided with ASF+SDF. The participants were asked to fill in a questionnaire consisting of 23 questions, each addressing one of the quality attributes. To enable checking the consistency of the answers provided by participants, the questionnaire contained at least three similar, but different questions for every quality attribute. For instance, in one of the questions the participants were asked to rate the understandability of the model transformation and in another one they were asked to indicate how much effort it would cost them to comprehend the model transformation. In each question, the participants had to indicate their evaluation on a five-point Likert scale (1 indicating a very low value and 5 indicating a very high value) [143]. For all six model transformations, the same questionnaire was used. The questionnaire can be found in Appendix B. During the evaluation, the participants had the transformation opened in the ASF+SDF Meta-Environment [54] on their own computer. There was no time-bound for the evaluation task.

Five of the transformations in the object set were evaluated by three participants, the transformation "ASF2C" was evaluated by four participants. Since the developers of the transformations we used as objects for the study were among the participants, they also evaluated the transformation they developed themselves. This may lead to biased results. However, in Section 4.5.4, we will show that there are no large inconsistencies between the evaluations of the developer and the other participants.

In addition to the quantitative evaluation of the transformations, a semi-structured interview was conducted after the questionnaire task to obtain qualitative statements. For each of the quality attributes, the participants were asked what characteristics of an ASF+SDF transformation influences the quality attribute.

4.5.4 Quality of the Analyzed Model Transformations

Each of the transformations has been manually evaluated by three participants (four for ASF2C) using the questionnaire. Recall that at least three similar, but different questions were asked for every quality attribute. For each participant, the evaluation of a particular quality attribute we used in our analysis is the mean of the answers he provided to all questions addressing that quality attribute. The results of the manual evaluation can be found in Table 4.5. The table shows per model transformation and per quality attribute three values. The first value is the mean evaluation of the participants. This value is used for establishing the relation between the metrics and the quality attributes. The second value is the standard

deviation of the evaluations. The standard deviation gives an indication of the consistency of the evaluations of the participants, i.e., it indicates disagreement. Since the standard deviations are in general low (< 1), we can conclude that the evaluations of the participants are relatively consistent. The last value is the difference between the evaluation of the developer of the transformation and the other participants. A positive value indicates that the developer's evaluation was higher than the average evaluation of the other participants and a negative value indicates that the developer's evaluation was lower than the average evaluation of the other participants. These differences indicate that there are in general no large inconsistencies between the evaluations of the developer and the other participants. However, there are some exceptions.

		Understandability	Modifiability	Reusability	Modularity	Completeness	Consistency	Conciseness
ACP2SM	Mean	2,50	3,25	2,89	3,89	3,89	3,61	3,11
	Std. Dev.	0,90	0,87	1,05	0,33	0,33	0,60	1,05
	Difference	0,00	-1,13	-1,33	-0,33	0,17	0,08	0,83
SL2XMI	Mean	3,17	3,17	3,11	2,00	4,11	3,89	3,33
	Std. Dev.	0,83	0,83	1,27	0,87	0,60	0,78	0,71
	Difference	0,13	0,13	0,33	0,00	-0,17	0,67	0,50
SLCheck	Mean	3,04	2,92	2,89	2,22	3,83	4,00	3,22
	Std. Dev.	0,92	0,70	1,17	0,83	0,61	0,00	0,97
	Difference	0,31	0,13	0,67	-0,33	-0,25	0,00	0,17
UML22Dot	Mean	3,38	3,25	3,56	2,00	4,06	4,11	3,56
	Std. Dev.	0,77	0,62	0,53	0,87	0,39	0,60	0,73
	Difference	0,19	0,00	-0,83	0,00	0,42	0,33	-0,33
ASF2C	Mean	1,69	1,44	2,25	4,00	4,00	3,50	2,67
	Std. Dev.	0,60	0,51	1,14	0,60	1,41	0,80	0,89
	Difference	0,42	-0,25	-1,67	-0,44	1,33	0,22	0,44
Repleo	Mean	2,38	2,54	3,11	3,50	3,33	3,06	3,11
	Std. Dev.	0,77	0,66	0,78	1,12	0,87	0,63	0,60
	Difference	0,00	-0,63	0,33	1,50	1,50	-0,83	-0,67

Table 4.5 Quality of the analyzed model transformations

Table 4.5, shows the differences in quality among the transformations as

evaluated by the participants. We explain a number of these differences using the feedback we obtained from the interviews. We also discuss other influences on the different quality attributes indicated by the participants.

Understandability is one of the quality attributes that shows differences between the transformations. An explanation given by the experts is that the understandability of a transformation is related to its size, viz., larger transformations are harder to understand. From Tables 4.4 and 4.5 we can conclude that the ascending ordering of transformation size, both in terms of the number of lines of code and in terms of the number of transformation functions they comprise, is almost the same as the descending ordering of their understandability. The participants also indicated that the use of descriptive names for variables and functions benefits the understandability of a model transformation. Besides characteristics of the transformation itself, the participants indicated that the understandability of a model transformation is affected by the understandability of its source and target metamodel.

According to the participants, the modifiability of a model transformation is strongly related to its understandability and its modularity. Understandability is related to modifiability since a transformation needs to be understood before it can be modified. Modularity is related to modifiability, since a modification can be performed more easily when the modification fits the module structure of the transformation. Therefore, also the type of modification that is required has influence on modifiability.

Modularity is affected by the number of modules in a transformation, but a low number of modules does not mean that the transformation is not modular per se. There are other means to mimic a module structure. For example, in the transformations "SL2XMI", "SLCheck", and "UML22Dot", comments are used to indicate where a new piece of functionality, or "module", starts.

The participants found it hard to assess the reusability of the transformations, since they had no idea what to reuse parts of the transformations for. Therefore, care should be taken when drawing conclusions regarding reusability based on the evaluations of the participants. In Chapter 3, we explained that there are two types of reuse, viz., as-is reuse and reuse-with-modify. As the name suggest, the latter type of reuse requires modifications. Therefore, reusability and modifiability are closely related. The participants indicated that reusability is, similar to modifiability, affected by understandability as well. A model transformation should be understood to assess whether it is eligible for reuse and how it should be modified if that is required. The participants also indicated that they expect that transformations with smaller modules are more reusable.

The participants also found it hard to assess the completeness of the transformations, since they did not know the exact specification of the transformations. In one of the questions in the questionnaire, the participants were asked to indicate

their familiarity with the transformations they had to evaluate. All participants knew of the existence of most of the transformations. However, except for the developer, they had no detailed knowledge about them. Since they knew that the transformations had been applied in practice already, they assumed that the transformations adhere more or less to their specifications. This explains why the values for completeness are about the same. Therefore, again, care should be taken when drawing conclusions regarding completeness based on the evaluations of the participants. The participants indicated that the source and target metamodel of a transformation should be well understood to assess whether all (required) elements of the source metamodel are covered by the transformation. In Section 7.2.2, we describe a visualization technique for analyzing the metamodel coverage of a model transformation.

For two of the analyzed transformations, viz., "ACP2SM" and "ASF2C", the participants indicated that it was clearly visible that multiple developers worked on these transformations. This was reflected by using a different style of writing comments and equations. However, the involvement of multiple developers is not the sole cause for inconsistencies, since the consistency of the "Repleo" transformation, which has been developed by one person only, was rated the lowest. Inspection of the source code of the "Repleo" transformation revealed that symbology and indentation are not used in a uniform manner throughout the transformation. The participants further indicated that the use of consistent naming is the main factor that influences consistency in general.

Conciseness is hard to evaluate according to the participants because the transformation needs to be well-understood to judge whether it can be written down in a more concise way. They did indicate that dead code in a transformation should be removed since it leads to inconciseness and decreases understandability.

4.5.5 Relating Metrics to Quality Attributes

To establish a relation between metrics and quality attributes, we analyze the correlation between them. The metrics were collected using the metrics collection tool we presented in Section 4.4. Since the participants were requested not to consider the standard libraries provided with ASF+SDF, metrics were also collected without considering library modules. For the metrics that require aggregation, we used the mean. An example of a metric requiring aggregation is *number of functions per module*. The value we used in the analysis represents thus the *mean number of functions per module*.

The data that has been acquired from the questionnaire is ordinal. Therefore, we use a non-parametric rank correlation test [59]. Since we have a small data set and we expect a number of tied ranks, we use Kendall's τ_b rank correlation test to acquire the correlations [144]. This test returns two values, viz., significance and

		Understandability		Modifiability		Reusability		Modularity		Completeness		Consistency		Conciseness	
No.	Metric	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.
1.	# Transformation functions	-,550	,002	-,303	,092	-,242	,183	,439	,017	,053	,772	-,190	,307	-,438	,020
2.	# Traversal functions	-,356	,048	-,432	,016	-,216	,235	,159	,385	,053	,772	-,136	,465	-,368	,051
3.	# Signatures per function	,032	,858	,019	,914	-,033	,857	,040	,828	-,172	,347	-,203	,274	-,021	,912
4.	# Equations per function	-,265	,141	-,213	,237	,033	,857	,279	,128	-,146	,426	-,285	,125	-,063	,741
5.	# Matching conditions per equation	,693	,000	,587	,001	,268	,140	-,571	,002	,172	,347	,407	,029	,507	,007
6.	# (In)equality conditions per equation	-,278	,123	-,097	,591	-,072	,692	,372	,043	,026	,885	-,068	,715	-,202	,286
7.	# Conditions per equation	,550	,002	,432	,016	,190	,297	-,518	,005	,238	,193	,488	,009	,424	,025
8.	# Equations and conditions per function	,175	,332	,252	,162	,229	,208	-,093	,612	-,040	,828	-,068	,715	,257	,173
9.	# Default equations per function	,084	,641	-,161	,370	,098	,590	-,080	,664	-,053	,772	-,041	,827	,076	,686
10.	Function val-in	-,123	,495	-,136	,452	-,098	,590	,080	,664	-,172	,347	-,231	,215	-,104	,581
11.	Function fan-in	-,136	,451	-,058	,747	,111	,541	,106	,562	-,040	,828	-,176	,343	,035	,854
12.	Function fan-out	,019	,914	,097	,591	,177	,331	,066	,717	-,040	,828	-,149	,422	,118	,532
13.	# Distinct return types per function	-,537	,003	-,355	,049	-,150	,408	,465	,011	-,093	,613	-,515	,006	-,382	,043
14.	# Modules	-,801	,000	-,553	,003	-,256	,166	,713	,000	-,164	,379	-,483	,011	-,532	,007
15.	# Functions per module	-,278	,123	-,032	,858	-,111	,541	,279	,128	,106	,563	-,054	,770	-,285	,131
16.	# Signatures per module	-,123	,495	,123	,496	-,059	,746	,120	,515	,106	,563	,027	,884	-,146	,440
17.	# Equations per module	,667	,000	,381	,035	,177	,331	-,638	,001	,212	,247	,570	,002	,452	,017
18.	# Parameterized modules	-,195	,338	-,137	,500	,012	,955	,235	,256	-,234	,257	-,432	,039	-,049	,818
19.	# Import declarations per module	-,162	,370	,148	,410	-,059	,746	,319	,082	-,172	,347	-,258	,165	-,090	,632
20.	# Times a module is imported	-,655	,000	-,393	,032	-,175	,343	,658	,000	-,232	,213	-,566	,003	-,437	,023
21.	Module fan-in	-,601	,001	-,307	,104	-,195	,306	,690	,000	-,073	,703	-,374	,054	-,399	,044
22.	Module fan-out	,188	,298	,432	,016	,059	,746	-,199	,277	,106	,563	,271	,144	,160	,397
23.	# Signatures without equations	-,481	,009	-,146	,424	-,135	,466	,631	,001	-,150	,420	-,476	,012	-,366	,057
24.	# Variable names per type	-,758	,000	-,432	,016	-,268	,140	,678	,000	-,053	,772	-,407	,029	-,507	,007
25.	# Distinct variables names per type	-,758	,000	-,432	,016	-,268	,140	,678	,000	-,053	,772	-,407	,029	-,507	,007
26.	# Types per variable name	-,658	,000	-,378	,045	-,152	,426	,631	,001	-,190	,321	-,584	,003	-,399	,044
27.	# Unused variables per module	-,291	,106	-,071	,694	-,124	,494	,332	,070	,053	,772	-,054	,770	-,313	,098
28.	# Start-symbols	-.680	,000	-,497	,006	-,190	,297	,611	,001	-,119	,515	-,407	,029	-,382	,043

Table 4.6 Kendall τ_b correlations

correlation coefficient. The significance of the correlation indicates the probability that there is no correlation between two variables even though correlation is reported, i.e., the probability for a coincidence. The correlation coefficient indicates the strength and direction of the correlation. A positive correlation coefficient means that there is a positive relation between metric and quality attribute and a negative correlation coefficient implies a negative relation. For more information on the interpretation on these values, the reader is referred to [145]. Since we are performing an exploratory study and not an in-depth study, we accept a significance level of 0,10. Note that correlation does not indicate a causal relation between metric and quality attribute[1]. Table 4.6 contains the correlations we acquired. The columns labeled C.C. list correlation coefficients and the columns labeled Sig. list (two-tailed) significance values.

Metrics that indicate the size of a transformation, viz., *number of (traversal) functions* and *number of modules*, correlate negatively with the quality attributes understandability and modifiability. This means that larger model transformations are harder to understand and to modify, which was acknowledged by the participants. Since understanding and modifying are the core activities in a maintenance process, this correlation implies that larger transformations are harder to maintain. A similar argument would hold for reusability as well. The results of our empirical study show a negative correlation between the aforementioned size metrics and reusability. However, these correlations are not significant. Modularity correlates positively with the size metrics. More functions implies a larger transformation. Especially in larger transformations the need for splitting functionality over modules becomes higher. Therefore, a larger transformation often implies a more modular transformation. The data indeed show a significant, positive correlation between the metrics *number of functions* and *number of modules* (see Table 4.7). A higher number of modules implies higher modularity. However, a high number of modules alone is not enough for a model transformation to be modular. Functionality should be well-spread over the modules. This usually leads to smaller modules, i.e., modules with fewer equations. This explains the negative correlation between *number of equations per module* and modularity. The metric *number of modules* correlates negatively with consistency as well. A high number of modules implies a high number of interfaces between modules. This may lead to inconsistencies. Also, when multiple developers work on a transformation, it is likely that they work on separate modules. Since every developer has his own style, this can lead to inconsistencies. Inspection of the "ASF2C" transformation, which has been developed by multiple developers, indeed reveals different programming styles. The size metrics also correlate negatively with conciseness. A model transformation is concise if it is free of superfluous elements. For larger

[1]The reader is referred to [146, p. 122] for some anecdotal examples of situations where correlation is interpreted as a causal relation.

transformations it is to be expected that their size can somehow be reduced.

The metric *number of (matching) conditions per equation* is positively correlated with understandability and modifiability. When writing equations, a tradeoff has to be made between writing a complex equation with little matching conditions or writing a simple equation with more matching conditions. The correlation indicates that simple equations with more matching conditions are preferred. The metric *number of (matching) conditions per equation* also correlates positively with conciseness. This may seem surprising, but the more (matching) conditions are used, the smaller, and hence more concise they become. Here conciseness plays a role on a different level, i.e., on the level of condition rather than on the level of complete transformation.

In transformations consisting of multiple modules, modules depend on each other. These dependencies are measured by the metrics *module fan-in*, *module fan-out*, *number of times a module is imported*, and *number of import declarations*. The metrics *module fan-in*, *number of times a module is imported* significantly correlate with modularity in a positive way. The metric *module fan-out* also correlates with modularity positively, however, this correlation is not significant.

The metrics *module fan-in* and the *number of times a module is imported* correlate negatively with modifiability. When a module on which other modules depend needs modifications, attention should be paid that these dependencies remain correct.

The metric *number of distinct return values per function* is negatively correlated with understandability and modifiability. When a function has multiple return values, it usually also has multiple equations. This has two disadvantages with respect to modifying a transformation. First, if only the equations belonging to one, or a few signatures need modifications, attention should be paid that the correct equation is modified. Second, more equations imply more modifications. From the correlation between the *number of distinct return values per function* and modifiability, it is to be expected that also a negative correlation is found between *number of signatures per function* and modifiability. However, this is not the case. The metric *number of distinct return values per function* also correlates negatively with consistency. This is to be expected because the return values are not consistent with each other. Another form of inconsistency is measured by the metric *number of types per variable*, since a variable that is of a different type in different modules is defined inconsistently. The results presented in Table 4.6 show that this metric indeed correlates with consistency negatively. Related to this is the negative correlation between the metric *number of (distinct) variables per type* and consistency. Redefinition of variables may lead to inconsistent naming. In fact, the metric *number of (distinct) variables per type*, which correlates negatively with consistency, measures this directly.

None of the metrics correlate significantly with reusability and completeness.

The participants in the empirical study indicated that they cannot evaluate reusability properly because they do not see what they can reuse parts of the transformations for. The reason that completeness does not correlate significantly with any of the metrics is that the experts could not evaluate completeness properly because they did not have the specification of the transformations they analyzed. Moreover, the time needed to get acquainted with the source and target language of the transformations is large.

4.5.5.1 Relations between Metrics

The metrics we defined in Section 4.3 are aimed at measuring certain concepts of ASF+SDF model transformations. Most of these concepts are related to each other in some way. Therefore, correlations can be found between the metics as well. Tables 4.7 and 4.8 present a few of them. Again, we use Kendall's τ_b rank correlation test to acquire the correlations.

We already observed when transformations grow larger, the need for splitting functionality over modules becomes higher. This is supported by the data, that show a significant, positive correlation between the metrics *number of functions* and *number of modules*. The metrics *number of variables per type* and *number of distinct variables per type* correlate significantly with the metric *number of modules* as well. Since variables are usually declared in a hiddens section of an SDF module, they have to be redefined in every module they are required. The metric *number of start symbols* also correlates significantly with the metric *number of modules*. An explanation for this is that the start-symbols are leftovers from unit testing the module.

	# Modules	
Metric	C.C.	Sig.
# Transformation functions	,474	,012
# Variable names per type	,722	,000
# Distinct variable names per type	,722	,000
# Start-symbols	,846	,000

Table 4.7 Kendall τ_b correlations between metrics (1)

Another pair of metrics that goes hand-in-hand is module fan-in and the number of times a module is imported (see Table 4.8). Functions in a module that is imported by another module are expected to be used by that module, which is expressed by fan-in.

	Module fan-in	
Metric	C.C.	Sig.
# Times a module is imported	,433	,028

Table 4.8 Kendall τ_b correlations between metrics (2)

4.5.6 Threats to Validity

Conducting empirical studies inherently involves threats to validity [147]. Here we discuss how we addressed potential threats to the validity of the study presented in this chapter.

An important issue that should be taken into account for empirical studies in general is the representativeness of the experimental design with respect to practice. We selected experienced ASF+SDF users as participants in our study. Our experience shows that model transformations are developed and maintained by experts in practice. Therefore, we exclude participant experience as a threat to the validity of our study.

The model transformations used as objects in our study are designed in and applied for practical purposes. Additionally, our sample of transformations is heterogeneous with respect to several characteristics. Hence, we do not consider the object selection as a threat to the validity of this study.

In our study, the participants conducted an evaluation task that is not representative for practical model engineering tasks, and therefore this is a potential threat to the validity. We addressed this threat in the design of our study by using at least three self-controlled questions for each quality attribute and we used the evaluations of four experts. The results were relatively consistent between the experts and between the self-controlled questions, respectively (see Table 4.5). Therefore, we minimized this threat to validity.

Our choice for the transformation language ASF+SDF could be discussed. Future replications of our study must prove whether the findings for ASF+SDF presented in this study will also hold for conceptually similar metrics for other transformation languages. Such a study is presented in Chapter 5 for the model transformation language ATL [34].

The number of observations in this study is rather small. This is a potential threat to the validity. It is difficult to find a larger number of experts in ASF+SDF for participation in such a study. We accepted this threat to the validity, since the study is a first exploration of transformation quality metrics. In future studies we plan to address this threat more accurately.

The participants were asked to evaluate the quality attributes of the transformations on a five-point Likert scale. The questionnaire contained at least three similar, but different questions for every quality attribute. In the statistical

analysis, we used the mean of the evaluations for each of the quality attributes. In general, the intervals between ordinal categories, such as Likert categories, cannot be assumed to be equally spaced since it cannot be assumed that participants regard them this way [148]. Therefore, an ordinal scale should not be treated as an interval scale. This means that the mean of a number of evaluations on an ordinal scale cannot be considered as the *average* evaluation. However, if the categories on the ordinal scale resemble an interval scale, i.e., if the categories appear to be equally spaced, it can be treated as an interval scale [149]. For each quality attribute, a similar scale is used for all questions addressing that quality attribute. Moreover, the evaluations were relatively consistent among those questions (see Table 4.5). Therefore, we treat the Likert scale as an interval scale and accept the small error that may be introduced.

There are a number of metrics that measure the size of a model transformation, in particular the metrics *number of transformation functions* and *number of modules*. From Table 4.6 can be observed that in case both these size metrics correlate with the quality attributes, other metrics do so as well. This may be an indication that the size of the model transformation has a confounding effect on the results of the empirical study [150]. To verify whether this is the case, we analyzed correlations between the size metrics and other metrics. The results of this analysis show that from the metrics that correlate significantly with all quality attributes when the size metrics do so, seven of them correlate significantly with both size metrics. We observed in Section 4.5.5.1, that for some of these metrics this is to be expected. However, additional research should be performed, preferably with more data, to investigate whether the size of a transformation has a confounding effect on the validity of the results of the empirical study.

4.6 Related Work

Harrison observes that research into metrics for assessing the quality of software systems mostly focusses on imperative and object-oriented software [151]. The study he presents in his article is aimed at defining metrics that can be used to assess quality attributes of functional programs. An experiments to relate a subset of these metrics to quality attributes is presented by Harrison et al. in [152]. Since ASF is a functional language, we adapted a number of the metrics he defined such that they can be used to measure the quality of model transformations.

In this chapter, we proposed a set of metrics for assessing the quality of model transformations created using ASF+SDF. Alves and Visser propose a set of metrics to monitor iterative grammar development in SDF [153]. They took metrics applicable to measure (E)BNF grammars and adapted them such that they can be used for measuring SDF grammars. Since SDF provides constructs for disambiguating grammars, they also introduced metrics for measuring the

number of disambiguation constructs used in grammars. The main difference with our work is that they focus on grammar development using SDF, whereas we focus on transformation development using both ASF and SDF. Therefore, we could not reuse the metrics they defined.

Kapová et al. have defined a set of metrics for evaluating maintainability of model transformations created with QVT Relations [154]. Most of the 24 metrics they defined are similar to the metrics we have defined. Their extraction process of 21 of their metrics has been automated by means of a tool in a similar way as we have done for ATL (see Section 5.4). They have applied their tool to three different transformations to demonstrate how to judge the maintainability of a model transformation using their metrics. This judgment is based on expectations rather than empirical evidence. Performing empirical validation is a point they indicate for future work.

Empirical studies have been performed for other types of software artifacts[2]. Basili et al. performed an empirical study to assess whether a set of metrics for measuring characteristics of object-oriented systems are suitable quality predictors [155]. The participants in their study, eight groups of three students each, had to develop a medium-sized system. Metrics were extracted from the source code of the various developed systems at the end of the implementation phase. Data was also collected on faults detected during the testing phase. Correlations between the metrics data and the fault data were analyzed to assess whether the metrics can be used to detect fault-prone classes. A similar study for model transformations can give valuable insights into the causes for faults in model transformations, and thereby on the influences on their quality. Based on the result of such a study, guidelines can be formulated aimed at decreasing the probability for faults.

Lange presents a number of empirical studies aimed at assessing and improving the quality of UML models [123]. In one of the studies, industrial UML models were analyzed for defects. Based on this study, two follow-up studies were conducted to assess whether respectively modeling conventions and the use of visualization techniques can reduce the number of defects in UML models.

4.7 Conclusions

In software engineering, metrics are frequently employed for evaluating the quality of software. Although it has been recognized that metrics should be proposed for measuring model transformations, little research has been performed in this area. Therefore, we presented in this chapter a set of 28 metrics for assessing the quality of model transformations developed with ASF+SDF. However, metrics alone do

[2]See for example the Empirical Software Engineering journal or the proceedings of the International Symposium on Empirical Software Engineering and Measurement

not suffice. They have to be related to quality attributes in order to establish whether they serve as valid predictors for these quality attributes. We presented the results of an empirical study in which we try to find relations between metrics that can be automatically derived from a set of ASF+SDF model transformations and a quantitative quality evaluation of the same set of transformations by a group of ASF+SDF experts. Our study is a first step into this direction and provides data that supports the selection of metrics for particular quality attributes. For most of the quality attributes we found metrics that correlate with them. In this study, we also requested the participants to state what in their opinion influences the different quality attributes of an ASF+SDF model transformation.

Metrics can be used, besides assessing quality, for different purposes as well. They can provide quick insights into the characteristics of a model transformation such as its size. Metrics can also be used to detect bad smells in a model transformation. Some of the traditional code smells [156] apply, maybe in a slightly adapted form, to model transformations as well. Examples of this are *large module*, *long parameter list*, and *dead code*.

Chapter 5

Quality Assessment of ATL Model Transformations

At the time of writing this thesis, ATL is one of the most popular model transformation languages around. We present in this chapter a set of 66 metrics for assessing the quality of model transformations developed with ATL. Again, we conducted an empirical study aimed at determining whether the metrics we defined are valid predictors for the quality attributes presented in Chapter 3. In this chapter, we present the results of this study.

5.1 Introduction

In Chapter 4, we presented a set of metrics for analyzing the quality of model transformations developed using the ASF+SDF term rewriting system. ASF+SDF has originally been developed for program transformation. However, it has proved its applicability to model transformation as well. Typically, modelware languages are used for developing model transformations instead of grammarware languages such as ASF+SDF [157]. Modelware languages are based on metamodels rather than on context-free grammars, which is the case for grammarware languages. The main difference between the two is that metamodels describe graphs, whereas grammar rules describe trees [158]. For model transformation languages based on metamodels also little research has been performed to establish metric sets to

assess the quality of model transformations developed with them. Therefore, we address in this chapter again research question RQ_3.

> **RQ_3**: *How can metrics be used to assess the quality of model transformations?*

At the time of writing this thesis, ATL is one of the most popular model transformation languages around. Therefore, we focus here on ATL. We have defined a set of 66 metrics for measuring ATL model transformations and developed a tool for automatically collecting them. Similarly to Chapter 4, we analyzed correlations between expert evaluations and metric values for a number of model transformations to establish a relation between the metrics and the quality attributes presented in Section 3.3.

The remainder of this chapter is structured as follows. Since this chapter mainly focuses on ATL, a short introduction to ATL is given in Section 5.2. In Section 5.3, the metrics we defined for analyzing ATL transformations are described. The tool we created for automatically extracting these metrics from ATL transformations is described in Section 5.4. In Section 5.5, we report on the empirical study we conducted. The research described in this chapter is strongly related to the research described in Chapter 4. Therefore, we will not describe related work here. Instead, the reader is referred to Section 4.6. Section 5.6 concludes this chapter.

5.2 ATL

In this chapter, we consider quality attributes of model transformations defined using the model transformation language ATL. A model transformation in ATL consists of rules and helpers. Rules are used to generate target model elements. Helpers are expressions defined in the Object Constraint Language (OCL) [41]. Their purpose is to define reusable chunks of code, typically intended for navigating source models and storing values. A model transformation in ATL is specified in one module, however, helpers can be specified in separate modules, called libraries.

Two types of rules can be distinguished, viz., declarative rules and imperative rules. A declarative rule consists of a source pattern and a target pattern. A source pattern consists of a set of model element types (metaclasses) that exist in any of the source metamodels of the transformation. During execution of a transformation, a rule *matches* a set of model elements from the source model that conform to the model element types defined in the source pattern of the rule. Optionally, a source pattern has a filter condition that restricts a match. After a rule has matched, target model elements are generated with their attributes and references as prescribed by the target pattern. Optionally, a rule has a `do`-section that provides imperative programming facilities. There are three types of

declarative rules, viz., matched rules, lazy matched rules, and unique lazy matched rules. Matched rules are automatically matched by the ATL engine, whereas (unique) lazy matched rules have to be invoked explicitly. A set of model elements may be matched by one matched rule only. Guards on source patterns can be used to enforce this. Lazy matched rule generate new target model elements upon each invocation, whereas unique lazy matched rules generate new target model elements only once. There is only one type of imperative rule, viz., the called rule. Called rules provide imperative programming facilities and, since they are rules, can also generate target model elements. Instead of having input patterns, called rules accept parameters, which are typically model elements. The target pattern is optional for called rules.

The definition of a helper comprises a context, a name, an optional list of parameters, a return value, and a body. The context of a helper defines the kind of (model) elements it applies to. The body of a helper is an OCL expression that should return a value of the specified return type. Local variables can be defined by means of `let`-clauses. Two types of helpers can be distinguished, viz., attribute helpers and operation helpers. Attribute helpers do not accept parameters. The main difference between the two is in their execution semantics. The return value of an operation helper is computed on each invocation of the helper, whereas the value of an attribute helper is computed only once.

An example of an ATL model transformation is depicted in Listing 5.1. The transformation is the book to publication transformation that can be found in the ATL zoo [159]. Its purpose is to transform a book model that adheres to the book metamodel into a publication model that adheres to the publication metamodel. This has to be done as follows. The title of a publication should be the title of a book. The authors attribute of a publication should be set to the concatenation of the authors of each of the chapters separated by the word 'and' and there should be no duplicates. The number of pages of the publication should be the sum of the number of all of the pages of the chapters in the book.

The name of the transformation specified in this ATL module is defined on line 1. On line 2, the input metamodel and output metamodel the transformation is based on are defined. In this case these are the `Book` and `Publication` metamodel respectively. The transformation consists of one matched rule and two operation helpers. The helper defined on line 4 is named `getAuthors`. It is defined in the context of the `Book` metaclass from the Book metamodel, meaning that it can be applied to model elements of that type. The parameter list of this helper is empty and its return type is `String`. The body of this helper (lines 5 to 14) collects all the authors (line 6) of the chapters of the book model element the helper is applied to (line 5), removes duplicates (line 7), and returns the concatenation of the authors of each of the chapters separated by the word 'and' (lines 8 to 14). The helper defined on line 16 is named `getSumPages`.

```
1  module Book2Publication;
2  create OUT: Publication from IN: Book;
3
4  helper context Book!Book def: getAuthors(): String =
5    self.chapters
6          ->collect(e | e.author)
7          ->asSet()
8          ->iterate(authorName;
9                    acc: String = '' |
10                   acc + if   acc = ''
11                         then authorName
12                         else ' and ' + authorName
13                         endif
14                 );
15
16 helper context Book!Book def: getSumPages() : Integer =
17   self.chapters
18         ->collect(f | f.nbPages).sum();
19
20 rule Book2Publication {
21   from  b   : Book!Book(b.getSumPages() > 2)
22   to    out : Publication!Publication(
23                 title   <- b.title,
24                 authors <- b.getAuthors(),
25                 nbPages <- b.getSumPages()
26               )
27 }
```

Listing 5.1 Book to publication transformation

Like the helper `getAuthors`, it is defined in the context of the `Book` metaclass from the Book metamodel. The parameter list of this helper is empty and its return type is `Integer`. The body of this helper (lines 17 and 18) collects the number of pages (line 18) of the chapters of the book model element the helper is applied to (line 17) and returns the sum of this (line 18). The rule defined on line 20 is named `Book2Publication`. It matches on model elements of type `Book` from the Book metamodel. The filter condition states that for the rule to match on a model element of type book, the result of the call to the operation helper `getSumPages`, which calculates the number of pages in a book, should be larger than two. The model element on which the rule matches is bound to variable `b`. The output pattern (lines 22 to 26) specifies that a model element of type `Publication` from the Publication metamodel should be generated. The `title` field of this model element should be set to the title of the book. The `authors` field should be set to the result of the helper `getAuthors` applied to the source model element. The `nbPages` field should be set to the result of the helper `getSumPages` applied to the source model element.

5.3 Metrics

This section describes the metrics we defined for assessing the quality of ATL model transformations. The metrics described here are specific for ATL. However, we will show in Chapter 6 that for most of them a conceptually equivalent metric can be defined for other model transformation languages as well.

Vignaga also presented a set of metrics for ATL [160]. We used the metrics he defined to complete our own set of metrics. Therefore, his metrics set overlaps with ours. In his report, Vignaga relates his metrics to quality attributes based on his expectations.

The metrics set we defined, can be divided into four categories, viz., rule metrics, helper metrics, dependency metrics, and miscellaneous metrics. In the remainder of this section, we will address each of these categories and elaborate on the metrics belonging to them. An overview of all the metrics can be found in Tables 5.1, 5.2, 5.3, and 5.4.

5.3.1 Rule Metrics

A measure for the size of a model transformation is the *number of transformation rules* it encompasses. In ATL, four different types of rules can be distinguished, viz., matched rules, lazy matched rules, unique lazy matched rules, and called rules. We have defined metrics for measuring the number of rules of every type. In case of a completely declarative model transformation, i.e., one with non-lazy matched rules only, it is to be expected that the amount of non-lazy matched rules is related to the size of the input metamodel, since typically a matched rule matches on one metamodel element. However, this is not necessarily the case, since matched rules can have input patterns that match on multiple metamodel elements at the same time or it may be the case that only part of the metamodel needs to be transformed.

Matched rules are scheduled by the ATL virtual machine, hence they do not have to be invoked explicitly. Lazy matched rules and called rules however need to be invoked explicitly in an ATL model transformation. Therefore, it may be the case that there are lazy matched rules or called rules in an ATL model transformation that are never invoked. This can have a number of reasons, e.g., the rule has been replaced by another rule. To detect this form of dead code, we propose to measure the *number of unused lazy matched rules* and the *number of unused called rules*.

ATL has support for rule inheritance. The use of inheritance may affect the quality of a model transformation in a similar way as it affects object-oriented software [155]. A rule deeper in the rule inheritance tree may be more fault-prone because it inherits a number of properties from its ancestors. Moreover, in deep hierarchies it is often unclear from which rule a new rule should inherit

from. To acquire insights into the rule inheritance relations in an ATL model transformation, we defined a number of metrics. We propose to measure the *number of rule inheritance trees* and per such tree the *maximum depth* and the *maximum width*. Note that this definition of the metric *maximum depth of inheritance tree* differs from the common definition of the metric depth of inheritance tree for object-oriented software [122]. Furthermore, we defined the metric *number of abstract transformation rules*. We also propose to measure for each abstract rule *the number of children* that inherit from it.

We have also defined a number of metrics on the input and output patterns of rules. The metrics *number of elements per input pattern* and *number of elements per output pattern* measure the size of the input and the output pattern of rules respectively. For an example, see Listing 5.2. In this example a rule is shown that has one input pattern element and two output pattern elements. These two metrics can be combined. The metric *rule complexity change* measures the amount of output model elements that are generated per input model element. For example, if the input pattern consists of one model element and two model elements are generated, the rule complexity change is $\frac{2}{1} = 2$. We do not consider model elements that are generated within `distinct foreach` blocks, since the amount of generated elements depends on the input model and can therefore not be determined statically. This metric may be used for measuring the external quality of a model transformation because it addresses the size increase (or decrease) of a model. Note that the metrics *number of elements per input pattern*, and hence the metric *rule complexity change*, are defined on matched rules only, since called rules do not have an input pattern. Instead, called rules have parameters similar to operation helpers. Therefore, for called rules we defined the metric *number of parameters per called rule*. It may be the case that some of these parameters are never used. To detect this form of dead code, we defined the metric *number of unused parameters per called rule*.

```
rule In2Out {
  from  in_1  : InMetamodel!MetaClassA
  to    out_1 : OutMetamodel!Metaclass1
                  binding_1   <- in_1.AttributeA,
                  binding_2   <- in_1.ReferenceA
              ),

        out_2 : OutMetamodel!Metaclass2
                  binding_1   <- in_1.AttributeB
              )
}
```

Listing 5.2 Example transformation rule

Bindings are used to initialize target model elements in an output pattern, which is also shown in Listing 5.2. The metric *number of bindings per output pattern* is another measure for the size of the output pattern of a transformation rule. Typically, the bindings of an output pattern are initialized with attributes and references derived from elements in the input pattern. We propose the metric *number of unused input pattern elements* to detect input pattern elements that are never referred to in any of the bindings and may therefore be obsolete. Matched rules require input pattern elements for the matching. In case that none of the input pattern elements of a lazy matched rule are used in that rule, this could be an indication that a called rule may be used instead. Note, however, that this is not always the case, since switching from a lazy matched rule to a called rule will no longer provide implicit tracing information that can be used elsewhere in the transformation. A related metric is the metric *number of direct copies*. This metric measures the number of rules that only copy (part of) an input model element to an output model element without changing any of the attributes. Note that this only occurs when the input metamodel and the output metamodel are the same, i.e., when the transformation rule is endogenous [21].

The input pattern of a matched rule can be constrained by means of a filter condition. The metric *number of rules with a filter condition* measures the amount of rules that have such an input pattern. Using such filter conditions, a rule matches only on a subset of the model elements defined by the input pattern. Therefore, it may be the case that there are multiple matched rules that match on the same input model elements. We defined the metric *number of matched rules per input pattern* to measure this. Note that it is required, except in case of rule inheritance, to have a filter condition on the input pattern since ATL does not allow multiple rules to match on the same input pattern.

Transformation rules can have local variables. These variables are often used to provide separation of concerns, i.e., to split the calculation of certain output bindings in orderly parts. We define two metrics to measure the use of local variables in rules, viz., *number of rules with local variables* and *number of local variables per rule*. We also measure the *number of unused local variables defined in rules* to detect obsolete variable definitions.

ATL allows the definition of imperative code in rules in a `do` section. This can be used to perform calculations that do not fit the preferred declarative style of programming. To measure the use of imperative code in a transformation, we defined two metrics, viz., *number of rules with a* `do`*-section* and *number of statements per* `do`*-section*.

Table 5.1 shows all the rule metrics we defined.

No.	Metric
1.	Number of transformation rules
2.	Number of matched rules
3.	Number of lazy matched rules
4.	Number of unique lazy matched rules
5.	Number of called rules
6.	Number of unused lazy matched rules
7.	Number of unused called rules
8.	Number of rule inheritance trees
9.	Maximum depth of inheritance tree
10.	Maximum width of inheritance tree
11.	Number of abstract transformation rules
12.	Number of children per abstract rule
13.	Number of elements per input pattern
14.	Number of elements per output pattern
15.	Rule complexity change
16.	Number of parameters per called rule
17.	Number of unused parameters per called rule
18.	Number of bindings per output pattern
19.	Number of unused input pattern elements
20.	Number of direct copies
21.	Number of rules with a filter condition
22.	Number of matched rules per input pattern
23.	Number of rules with local variables
24.	Number of local variables per rule
25.	Number of unused local variables defined in rules
26.	Number of rules with a `do`-section
27.	Number of statements per `do`-section

Table 5.1 Rule metrics

5.3.2 Helper Metrics

Besides transformation rules, an ATL transformation also consists of helpers. Helpers also affect the size of a model transformation. Therefore, we defined metrics to measure the *number of helpers* in a transformation. There are a number of different types of helpers, in particular two orthogonal distinctions can be made. On one hand, there are helpers with context and helpers without context. On the other hand, there are attribute helpers and operation helpers. The operation helpers can be further subdivided into operation helpers with parameters and without parameters. We defined different metrics for measuring the number of helpers of each type. Since helpers may be defined in the transformation module or in library units, we also measure the *number of helpers per unit*. This provides an indication of the division of helpers among units.

Similarly to lazy matched rules and called rules, helpers need to be invoked explicitly. Therefore, again, it may be the case there are some helpers present in a model transformation that are never invoked. To detect such unused helpers, we propose the metric *number of unused helpers*.

Helpers are identified by their name, context, and, in case of operation helpers, parameters. It is possible to overload helpers, i.e., define helpers with the same name but with a different context. To measure this kind of overloading we define the metrics *number of overloaded helpers* and *number of helpers per helper name*. Overloading is typically used to define similar operations on different datatypes. Of course it is also possible to define multiple different operations on the same datatype, i.e., in a different context. Therefore, we propose to measure the *number of helpers per context*.

Helpers are often used to manipulate collections. Therefore, we measure the *number of operations on collections per helper*. Also, conditions are often used in helpers. The metric *helper cyclomatic complexity* is related to McCabe's cyclomatic complexity [161], it measures the amount of decision points in a helper. Currently, only `if` statements are considered as decision points. In the future, it could be extended to take into consideration other constructs that influence the flow of control, such as the `for` loop. Also the complexity of OCL expressions could be taken into account when measuring the complexity of a helper [162].

Similar to transformation rules, helpers also allow the definition of local variables. We define the metrics *number of helpers with local variables*, and *number of local variables per helper* to measure the use of local variables in helpers. Again, we also measure the *number of unused local variables defined in helpers* to detect obsolete variable definitions. Similar to called rules, operation helpers have parameters. To get more insight in the use of parameters we propose to measure the *number of parameters per operation helper*. Also parameters may be unused. To detect this form of dead code, we propose the metric *number of unused parameters per operation helper*.

Table 5.2 shows all the helper metrics we defined.

5.3.3 Dependency Metrics

Transformation rules and helpers also depend on each other. Transformation rules can invoke (other) lazy matched rules, called rules, and helpers. Helpers can invoke (other) helpers and called rules. These dependencies are measured using the metrics *rule fan-out*, *helper fan-out*, *lazy rule fan-in*, *called rule fan-in*, and *helper fan-in*. Rules can also refer to each other in an implicit way, i.e., using a `resolveTemp()` expression. This dependency is measured using the metrics *number of calls to* `resolveTemp()` and *number of calls to* `resolveTemp()` *per rule*.

No.	Metric
28.	Number of helpers
29.	Number of helpers with context
30.	Number of helpers without context
31.	Number of attribute helpers
32.	Number of operation helpers
33.	Number of operation helpers with parameters
34.	Number of operation helpers without parameters
35.	Number of helpers per unit
36.	Number of unused helpers
37.	Number of overloaded helpers
38.	Number of helpers per helper name (overloadings)
39.	Number of helpers per context
40.	Number of operations on collections per helper
41.	Helper cyclomatic complexity
42.	Number of helpers with local variables
43.	Number of local variables per helper
44.	Number of unused local variables defined in helpers
45.	Number of parameters per operation helper
46.	Number of unused parameters per operation helper

Table 5.2 Helper metrics

An ATL model transformation can consist of multiple units that depend on each other. For measuring this dependency, we defined four metrics. The metrics *number of imported units* and *number of times a unit is imported* are used to measure the import dependencies of units. To measure how the internals of units depend on each other, we defined the metrics *unit fan-in* and *unit fan-out*. These metrics measure the number of calls from rules and helpers in a unit to helpers in another unit and vice versa.

The last dependency category, i.e., the dependency of rules and helpers on built-in functions is measured by the metric *number of calls to built-in functions*. Built-in functions are OCL functions and also additional ATL operations such as `replaceAll()`. The built-in functions `println()` and the `debug()` deserve special attention. The `debug()` function is used to print information to the console that can be used for debugging. In practice, we see that sometimes the `println()` function is used for a similar purpose. The occurrence of these two functions may indicate that the model transformation is still under development.

Table 5.3 shows all the dependency metrics we defined.

No.	Metric
47.	Rule fan-out
48.	Helper fan-out
49.	Rule fan-in
50.	Lazy matched rule fan-in
51.	Called rule fan-in
52.	Helper fan-in
53.	Number of calls to `resolveTemp()`
54.	Number of calls to `resolveTemp() per rule`
55.	Number of imported units
56.	Number of times a unit is imported
57.	Unit fan-in
58.	Unit fan-out
59.	Number of calls to built-in functions
60.	Number of calls to `println()`
61.	Number of calls to `debug()`

Table 5.3 Dependency metrics

5.3.4 Miscellaneous Metrics

We defined six more metrics that do not fit the discussed categories. The metric *number of units* measures the number of units that make up a model transformation. Apart from the unit that contains the transformation rules, i.e., the module, units can only contain helpers. Therefore, the metric *number of helpers per unit* can be used to measure the size of a unit. These metrics can provide insight in the size and modularity of a model transformation. It may be the case that there are library units imported from which no helper is invoked in the transformation. To detect this, we define the metric *number of unused units*.

The last two metrics provide insight in the context of the model transformation. It is to be expected that model transformations involving more models are more complex. Therefore, we propose to measure the *number of input models* and the *number of output models*.

Recall that in the body of a rule or a helper, local variables can be defined. Since these variables are local to a rule or helper, they need to be redefined in another rule or helper if a local variable of the same type is required. This may lead to inconsistencies in variable naming, i.e., variables of the same type may have different names in different rules or helpers. To detect such inconsistencies we defined metrics that measure the *number of types per variable name*.

Table 5.4 shows all the miscellaneous metrics we defined.

No.	Metric
62.	Number of units
63.	Number of helpers per unit
64.	Number of unused units
65.	Number of input models
66.	Number of output models
67.	Number of types per variable name

Table 5.4 Miscellaneous metrics

5.4 Tool

We implemented a tool that enables automatic collection of the metrics presented in Section 5.3 from ATL model transformations. The architecture of the tool is shown in Figure 5.1.

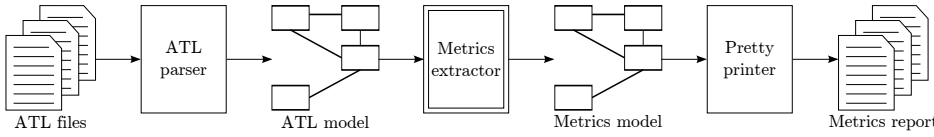


Figure 5.1 Metrics extraction tool architecture

An ATL model transformation consists of a module and possibly a number of libraries. The files containing this module and libraries are parsed by the ATL parser. This results in ATL models representing the model transformation. These models are the input for the metrics extractor. This metrics extractor is itself a model transformation implemented in ATL. It consists of one matched rule that matches on an ATL module, and a number of lazy matched rules, each for calculating the value of one of the metrics. An example of such a lazy matched rule is depicted in Listing 5.3.

The output of the metrics extractor is a model that contains the metrics data. The metamodel that describes such metrics models is depicted in Figure 5.2. The Object Management Group (OMG) has defined a software metrics metamodel as well [163]. Their metamodel is intended to serve as a format for the interchange of measurement data derived from software artifacts. Since we require a metamodel for the presentation of metrics only, it is too elaborate for our purposes. Therefore, we defined one ourselves. Note, however, that model transformations can be defined to transform between the two formats. A metrics model can be provided as input to a pretty printer. This pretty printer is a model-to-text transformation implemented in Xpand [164]. The output of the pretty printer is a comma separated value file that can be read by a spreadsheet application.

```
lazy rule NumTransformationRules{
  from  m_in : ATL!Module
  to    s_out: Metrics!SimpleIntegerMetric(
                        Metric  <- 'Number of Transformation Rules',
                        Value   <- thisModule.getAllRules()
                                                    ->size()
               )
}

helper def: getAllRules(): OrderedSet(ATL!Rule) =
  ATL!Rule.allInstances();
```

Listing 5.3 Metric extraction rule

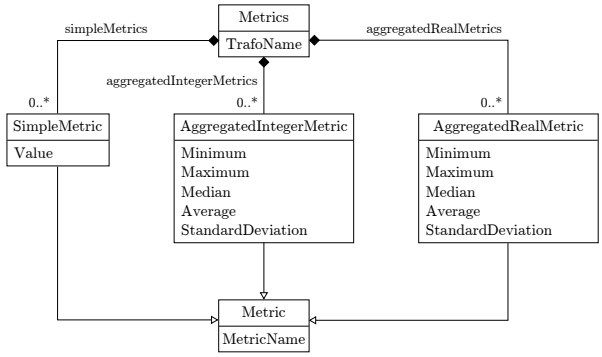


Figure 5.2 Metrics metamodel

In Section 5.3 we presented two types of metrics, viz., metrics that are measured over the entire transformation and metrics that are measured on a smaller scale, i.e., per unit, per rule, or per helper. The former type of metric has as single value for the entire transformation. We refer to this type of metrics as *simple metrics*. The latter type of metric has multiple values for the entire transformation, viz., one for every element that is measured (unit, rule, or helper). To assess the transformation as a whole we do not present all of these values. Instead, we give average, minimum, maximum, median and standard deviations for these metrics. We refer to this type of metrics as *aggregated metrics*.

The metrics extractor gathers data from an ATL model. Therefore, there is a problem with identifying calls to helpers. The call to a helper consists of only its name. However, a helper is defined by its name, context, and parameters. To distinguish between calls to helpers with the same name, more information is required. Unfortunately, this information is only available at run-time. We therefore identify a call to a helper by its name only. We realize that this is a threat to the validity of our measurements, since some of the metrics may be calculated

incorrectly. However, from our own experience and from the transformations available in the ATL zoo [159], we can conclude that there are few transformations that have overloaded helpers. Moreover, if there are overloaded helpers, this is indicated by a metric.

5.5 Empirical Study

The quality of a model transformation is not measurable directly. Therefore, we resort to metrics, which can be measured directly. Before these metrics can be used for assessing the quality of model transformations, we have to establish a relation between the metrics and quality attributes relevant for model transformations. To explore this relation, we conducted an empirical study similar to the one presented in Section 4.5. In the empirical study we used a collection of seven ATL model transformations with different characteristics. For each of these transformations, metrics data were collected using the metrics collection tool presented in Section 5.4. Quality attributes of the same collection of transformations were quantitatively assessed by nineteen ATL experts using a questionnaire. To establish a relation between the metrics and the quality attributes, we analyzed the correlations between the metrics data and the expert feedback. In this section, we describe the design and results of the empirical study.

5.5.1 Objects

We used seven ATL model transformations as objects in our empirical study. These transformations were acquired from various sources and they have different characteristics. Here, we shortly describe their purpose.

The first transformation is used to perform some type checking and reference resolving of a PicoJava program. PicoJava is a subset of the full Java language [165]. The transformation was developed in a research project. The second transformation generates a relational database model taking as input a UML class model. It also generates a trace model specifying links between source and target model elements. Transformation users can configure the transformation behavior by means of a configuration model. The transformation has been used as a case study in a research project [166]. The third transformation has been used for educational purposes. Students were asked to develop a transformation that generates code from a simple state machine language. The purpose of the fourth transformation is to transform an R2ML model into an XML model with R2ML syntax elements. The Rule Markup Language (R2ML) is a general web rule markup language used to enable sharing rules between different rule languages. The transformation language is part of the ATL transformation zoo [159]. The fifth transformation copies a UML2 model. The transformation is part of a

collection of MDE case studies [167]. The sixth transformation is part of a chain of refining model transformations presented in Chapter 8. The purpose of the transformation is to enable reliable communication over a lossy channel between two communicating objects. It was developed as part of a research project. The last transformation is the transformation used for conducting the experiment presented in Section 5.4. It takes an ATL model transformation and extracts the metrics presented in Section 5.3 from it. Table 5.5 provides an overview of the objects of our empirical study along with some of their characteristics.

Transformation	LOC	# Rules	# Helpers	Purpose	Ref.
PicoJavaType	227	12	14	PicoJava type checking and reference resolving	
R2ML2XML	1125	55	1	Generate an XML document of an R2ML model	[159]
SM2NQC	158	13	1	Generate NQC code from state machines	
UML2DB	5152	84	76	Generate a relational database model from a UML class model	[166]
UML2Copy	4158	199	1	Copy a UML 2 model	[167]
Lossy2Lossless	1003	37	2	Enable reliable communication over an unreliable channel	[67]
ATL2Metrics	2110	93	28	Extract metrics from ATL transformations	[63]

Table 5.5 Characteristics of the analyzed model transformations

5.5.2 Participants

The participants in the study were nineteen experienced users of ATL with various backgrounds. Some of them are part of the ATL development team, whereas others use ATL only occasionally. None of the authors of the article this chapter is based on participated as participant in this study. The developers of some of the objects presented in Section 5.5.1 were among the participants. To avoid biased results, measures were taken such that the participants did not evaluate a model transformation they developed themselves.

5.5.3 Task

The task of the participants was to quantitatively evaluate the quality of one or more of the objects, i.e., ATL model transformations. They were asked to fill out a questionnaire consisting of twenty-one questions each addressing one of the quality attributes. To enable checking the consistency of the answers provided by

the participants, the questionnaire contained at least three similar, but different questions for every quality attribute. For instance, in one of the questions the participants were asked to rate the understandability of the model transformation and in another one they were asked to indicate how much effort it would cost them to comprehend the model transformation. In each question, the participants had to indicate their evaluation on a seven-point Likert scale [143]. For all objects, the same questionnaire was used.

It was likely that the participants had no previous knowledge of the transformations they needed to evaluate. Therefore, we provided them with a brief description explaining that purpose We also provided them with the input and output metamodels of the transformation, such that they could run the transformation if desired. The participants could use as much time for the evaluation task as they needed.

In addition to the quantitative evaluation task, we used the questionnaire to obtain qualitative statements from the participants. They were requested to indicate what characteristics of an ATL model transformation in their opinion influences each of the quality attributes.

In total, there were nineteen participants in the empirical study. All participants evaluated at least one of the objects. There were two participants that evaluated three objects. This leads to a total of twenty-two evaluations.

5.5.4 Quality of the Analyzed Model Transformations

Each of the transformations has been manually evaluated by at least two participants using the questionnaire. Recall that at least three similar, but different questions were asked for every quality attribute, For each participant, the evaluation of a particular quality attribute we used in our analysis is the mean of the answers he provided to all questions addressing that quality attribute. The results of the manual evaluation can be found in Table 5.6. The table shows in the second column the number of observations for a particular transformation, i.e., the number of participants that evaluated that transformation. In columns 4 to 9, the table shows per model transformation and per quality attribute two values. The first value is the mean evaluation of the participants. This value is used for establishing the relation between the metrics and the quality attributes. The second value is the standard deviation of the evaluations. The standard deviation gives an indication of the consistency of the evaluations of the participants, i.e., it indicates disagreement. Since the standard deviations are in general low ($< 1,5$), we can conclude that the evaluations of the participants are relatively consistent.

	# Observations		Understandability	Modifiability	Reusability	Completeness	Consistency	Conciseness
PicoJavaType	4	Average	5,06	4,88	4,92	6,00	6,25	5,00
		Std. Dev.	1,34	1,15	1,24	0,73	0,75	1,71
R2ML2XML	4	Average	4,56	4,94	4,25	5,71	6,00	4,58
		Std. Dev.	1,59	0,85	2,22	0,83	0,74	1,51
SM2NQC	2	Average	4,50	4,63	4,50	5,88	5,67	5,67
		Std. Dev.	1,41	1,06	1,05	0,64	0,82	1,03
UML2DB	2	Average	2,13	2,25	3,17	4,29	5,33	2,50
		Std. Dev.	0,64	1,04	1,60	1,80	1,03	0,55
UML2Copy	3	Average	5,42	4,42	3,33	6,00	5,11	3,44
		Std. Dev.	0,67	1,62	1,32	0,60	1,54	2,01
Lossy2Lossless	2	Average	3,88	4,25	3,67	6,29	5,67	2,67
		Std. Dev.	1,81	0,89	1,21	0,76	1,03	0,82
ATL2Metrics	5	Average	4,78	4,63	5,21	5,53	5,47	3,93
		Std. Dev.	1,59	1,16	1,31	1,43	0,99	1,75

Table 5.6 Quality of the analyzed model transformations

5.5.5 Relating Metrics to Quality Attributes

To establish the relation between metrics and quality attributes we analyze the correlation between them. The metrics were collected using the metrics collection tool presented in Section 5.4. For the metrics that require aggregation, we used the mean. An example of a metric requiring aggregation is *number of parameters per called rule*. The value we used in the analysis represents thus the *mean number of parameters per called rule*.

The data that has been acquired from the questionnaire is ordinal. Therefore, we use a non-parametric rank correlation test [59]. Since we have a small data set and we expect a number of tied ranks, we use Kendall's τ_b rank correlation test to acquire the correlations [144]. This test returns two values, viz., significance and correlation coefficient. The significance of the correlation indicates the probability that there is no correlation between two variables even though correlation is reported, i.e., the probability for a coincidence. The correlation coefficient indicates the strength and direction of the correlation. A positive correlation coefficient means that there is a positive relation between metric and quality attribute and a

negative correlation coefficient implies a negative relation. For more information on the interpretation on these values, the reader is referred to [145]. Table 5.7 shows the correlations that were acquired. Since we are performing an exploratory study and not an in-depth study, we accept a significance level of 0,10. The significant correlations are marked. The columns labeled C.C. list correlation coefficients and the columns labeled Sig. list (two-tailed) significance values.

In Section 4.5, we presented the results of a similar empirical study for model transformations developed with the ASF+SDF term rewriting system. In that study we found that the size of a transformation expressed in terms of the number of transformation functions correlates negatively with understandability and modifiability. Therefore, we expect to find similar correlations for ATL as well. In Table 5.7 it is shown that no significant correlation is found between the metrics that measure the amount of transformation rules and helpers, and the quality attributes understandability and modifiability. However, the participants indicated in the qualitative part of the empirical study that they see size as a negative influence on both the understandability and modifiability of an ATL model transformation. Note that only one of the nineteen ATL experts also participated in the empirical study for ASF+SDF. Most of the participants indicated that the amount of called rules in particular has a negative impact on understandability and modifiability. The correlation coefficient between the metric number of called rules and understandability and modifiability is negative, however insignificant. A reason that was mentioned by the participants for this negative influence is that called rules, and also `do`-sections, are resorted to when a specific solution to part of a transformation problem is required. This is also addressed as a reason for low reusability of called rules. Table 5.7 shows that significant negative correlations have been found between the metrics called rule fan-in and lazy rule fan-in and a number of quality attributes. The participants mentioned that the use of, specifically, called rules leads to more complex rule interaction and coupling between rules. Although this argument holds for lazy matched rules as well, it is mentioned less often.

ATL allows defining helpers in separate units, or libraries as they are called in ATL terminology. Table 5.7 shows that the metric number of units correlates negatively with the quality attributes understandability, modifiability, completeness, and conciseness. Modularizing software is generally considered to be beneficial for its quality. Therefore, a positive correlation would have been expected here. It must be noted that the participants mentioned the use of libraries as an influence on quality only with respect to conciseness and reusability. However, no significant correlation has been found between the metric number of units and reusability. In Table 5.7 it is also shown that some other metrics related to the use of libraries, viz., unit fan-in, and unit fan-out correlate in a negative way with the quality attributes understandability, modifiability, completeness, and conciseness. A high

	Understandability		Modifiability		Completeness		Consistency		Conciseness		Reusability	
Metric	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.	C.C.	Sig.
# Transformation rules	,024	,885	-,086	,604	-,082	,623	-,364	,031	-,273	,099	-,092	,582
# Matched rules	,014	,931	,000	1,000	,034	,839	-,029	,861	-,129	,435	-,383	,022
# Lazy matched rules	-,081	,633	,041	,812	-,201	,243	-,058	,741	,051	,765	,239	,168
# Called rules	-,128	,482	-,263	,149	-,158	,389	-,308	,098	-,365	,046	-,347	,060
# Unused lazy matched rules	-,025	,893	,138	,460	-,152	,419	,026	,892	,076	,687	,217	,251
# Unused called rules	-,128	,482	-,263	,149	-,158	,389	-,308	,098	-,365	,046	-,347	,060
# Elements per output pattern	-,375	,026	-,215	,202	-,228	,180	,124	,472	-,146	,389	-,122	,474
# Parameters per called rule	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249
# Unused parameters per called rule	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249
# Unused input pattern elements	-,032	,854	,059	,737	-,033	,854	-,056	,758	,033	,854	,297	,097
# Direct copies	,227	,197	,040	,822	,322	,070	-,059	,745	-,125	,478	-,196	,271
# Rules with filter	-,005	,977	-,038	,818	-,005	,977	-,049	,771	-,129	,435	-,402	,016
# Rules per input pattern	-,109	,507	-,114	,489	-,130	,434	,029	,861	-,014	,931	,315	,059
# Rules with local variables	-,013	,944	-,051	,780	,032	,861	-,137	,460	-,178	,328	,302	,100
# Variables per rule	-,038	,834	-,076	,676	,070	,700	-,111	,550	-,242	,184	,225	,220
# Rules with `do`-section	,000	1,000	-,153	,385	-,057	,747	,018	,922	-,108	,540	-,173	,332
# Helpers	-,178	,296	-,229	,180	-,201	,243	-,047	,786	-,246	,152	,187	,281
# Unused helpers	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249
# Helpers per helper name (overloadings)	-,033	,855	-,066	,714	-,054	,769	,220	,237	,060	,741	,007	,971
Helper cyclomatic complexity	-,248	,142	-,154	,364	-,357	,037	-,026	,882	-,175	,304	,126	,461
# Variables per helper	,006	,972	,063	,727	,013	,944	-,111	,550	,178	,328	,328	,074
Rule fan-out	-,223	,175	-,124	,453	-,333	,046	-,157	,351	-,235	,157	,024	,885
Helper fan-out	,020	,907	,183	,278	-,105	,537	,302	,081	,264	,120	,136	,426
Lazy rule fan-in	-,356	,037	-,143	,404	-,397	,021	,005	,976	-,021	,905	-,062	,719
Called rule fan-in	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249
Helper fan-in	,138	,402	,000	1,000	-,024	,885	-,236	,162	-,196	,236	-,005	,977
# Calls to `resolveTemp()`	-,352	,049	-,358	,045	-,159	,380	-,088	,632	-,236	,189	-,179	,323
# Calls to `resolveTemp` per rule	-,326	,068	-,306	,087	-,106	,558	-,061	,741	-,236	,189	-,153	,399
# Imported units	-,252	,164	-,080	,661	-,323	,078	,110	,554	-,027	,883	-,223	,225
# Times a unit is imported	-,318	,088	-,138	,459	-,379	,045	,086	,654	-,084	,656	-,247	,192
Unit fan-in	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249
Unit fan-out	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249
# Units	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249
# Helpers per unit	-,178	,296	-,229	,180	-,201	,243	-,047	,786	-,246	,152	,187	,281
# Input models	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249
# Output models	-,407	,029	-,407	,029	-,391	,038	-,122	,524	-,345	,066	-,218	,249

C.C. : Correlation coefficient
Sig. : Significance (two-tailed)

Table 5.7 Kendall τ_b correlations

value for unit fan-in and fan-out indicates a high coupling between the units that comprise a model transformation. In traditional software development, it is considered to be desirable to have low coupling between modules [168]. A reason for this is that having less interconnections between units reduces the time needed by developers to understand the details of other units. Moreover, a change in a unit can cause a ripple effect, i.e., the effect of the change is not local to the unit [169]. Similarly, an error in one unit can affect other units. The metrics number of imported units and number of times a unit is imported also relate to the use of libraries. However, they correlate less with the quality attributes.

Unused elements are usually not beneficial for the understandability and modifiability of a transformation because they clutter it. Since unused elements are in principal superfluous, they have an obvious negative effect on conciseness. For the metrics number of unused helpers, number of unused called rules, and number of unused parameters of called rules such correlations have been found. However, no significant correlations can be found for the metrics number of unused lazy rules and number of unused input pattern elements.

The participants mentioned that simple transformations with one-on-one mappings tend to be the most complete. The number of direct copies measures one-on-one copies. This metric correlates positively with completeness, supporting the claim of the participants. Having simple, or small, transformations was mentioned as having a positive influence on reusability as well. Some of the participants indicated that model transformations should be split up in a chain of smaller transformations, because this increases reusability and, as mentioned before, also understandability. A negative influence on reusability and also modifiability that was mentioned is the use of the `resolveTemp()` expression. The correlations presented in Table 5.7 partly support this. The reason that was mentioned for this influence is that the use of `resolveTemp()` expressions increases coupling between rules. This is also an explanation for the negative correlation that was found between understandability and the metrics number of calls to `resolveTemp()` and number of calls to `resolveTemp()` per rule.

The number of input and output models that a transformation takes correlates in a negative way with the quality attributes understandability, modifiability, and completeness. When a transformation takes multiple models as input and generates multiple output models, it is to be expected that the transformation rules of that transformation are more complex, and thereby less understandable and modifiable, since information from multiple sources needs to be combined and assigned to the correct targets. The data partially support this. A significant, positive correlation exists between the metrics number of output models and number of elements per output pattern. Since the transformations we used for our study all have one element per input pattern, no correlation can be found between this metric and the metric number of input models. In the qualitative part of

the empirical study, the participants indicated that the number, as well as the complexity of the involved metamodels has a negative effect on understandability, modifiability, and also completeness. The reason for this is that the metamodels need to be understood before the transformation can be understood and modified. Moreover, it is hard to detect incompleteness in a transformation if the metamodels and their interrelations are not fully understood.

In the qualitative part of the empirical study, more feedback was acquired regarding the quality of ATL model transformations of which some cannot be related to metrics directly. The participants indicated that lay-out of the code of a model transformation, as well as the use of proper naming for rules, helpers, and variables will increase the understandability and modifiability of the transformation. Enforcing proper naming by means of a coding convention was mentioned as a positive influence on consistency as well. Proper comments and additional documentation of the requirements and design of the transformation were mentioned as positive influences on completeness and, again, understandability. According to the participants, the use of helpers has a positive influence on almost all quality attributes, albeit that they should not become too complex. Helpers prevent duplication of code, which increases consistency and conciseness. It was mentioned that navigation of the source model should be delegated to helpers rather than implementing it as part of a rule. Besides being beneficial for the quality attributes considered in this paper, experiments have shown that this has a positive effect on performance as well [121]. Rule inheritance has, according to the participants, a positive influence on the quality attributes understandability, conciseness, and consistency. The use of rule inheritance can lead to rules that are more concise and have a common pattern, which are in general more understandable. Although, a deep inheritance tree should be avoided, since it decreases understandability again. Similarly to what holds for inheritance trees in object-oriented software, a rule deeper in the rule inheritance tree may be more fault-prone because it inherits a number of properties from its ancestors [155]. Moreover, in deep hierarchies it is often unclear from which rule a new rule should inherit from.

5.5.6 Threats to Validity

When conducting empirical studies, there are always threats to the validity of the results of the study [147]. Here, we address the potential threats to validity we identified for the empirical study we performed.

An important issue that should be taken into account for empirical studies in general is the representativeness of the experimental design with respect to practice. The objects used in the study are seven ATL model transformations that have been developed and applied for various purposes. Therefore, they have different characteristics. It must be noted, however, that there is only one

transformation that is largely imperative, i.e., 57 of the 84 transformation rules of the UML2DB transformation are called rules and all of the 84 rules have a `do`-section. In Figure 5.3, a scatter plot is depicted of the data concerning the number of called rules in the objects and their reusability. Since almost all data points are on the left side of the graph, the significant negative correlation that has been found between the number of called rules and reusability is mainly explained by the two outlying data points that originate from the UML2DB transformation. Even though we use a ranked correlation test to reduce the influence of outlying values, these correlation would not have been found if this transformation would not have been in the object set. Therefore, we cannot base our conclusions solely on the correlations we found. To address this threat to validity, we also collected qualitative data. The participants were requested to indicate the characteristics of an ATL model transformation that in their opinion influences each of the quality attributes. These qualitative statements were used to support or refute the results of the quantitative analysis.

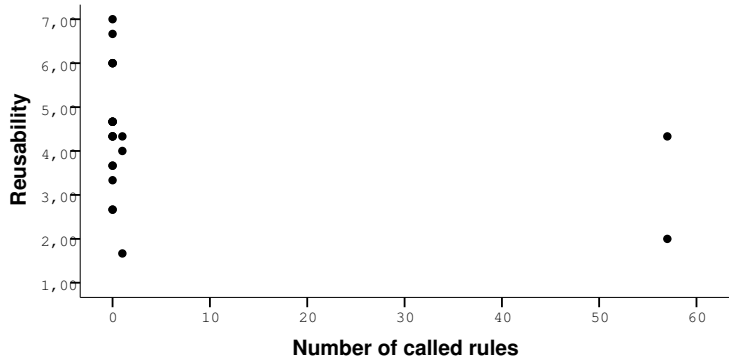


Figure 5.3 Relating the number of called rules to reusability

The participants in our study have different backgrounds. Some of them are part of the ATL development team, whereas others use ATL only occasionally. In the questionnaire, the participants were requested to rate their knowledge of ATL. Most of the participants rated their knowledge as *high* or *very high*. Therefore, we do not consider participant experience as a threat to the validity of this study.

The task the participants had to perform was evaluating different quality attributes of model transformations. Evaluating model transformations is not a typical task for a model transformation developer. Moreover, participants are not always the most careful readers [170]. Both these issues may decrease the validity of the data. To address both threats to validity, we posed for each of the quality attributes at least three similar but different questions. The results showed that

the responses provided by the participants to each of the similar questions were relatively consistent. Also the answers among the participants were relatively consistent. Therefore, we minimized these threats to validity.

Recall that we investigated the possible confounding effect of the size of a model transformation on the results of the empirical study we conducted for ASF+SDF (see Section 4.5.6). Since hardly any quality attribute correlates significantly with any of the metrics that indicates the size of a model transformation, we conclude that the size of a model transformation had no confounding effect on the results of our empirical study.

5.6 Conclusions

In this chapter, we presented a set of 66 metrics for measuring model transformations developed with the model transformation language ATL. We performed an empirical study aimed at establishing whether the metrics are valid predictors for the quality attributes presented in Chapter 3. The quality of seven model transformations developed with ATL was manually assessed by nineteen experienced ATL users. Metrics were collected from the same set of model transformations using the tool we created. We analyzed the correlation between the expert data and the metric data. A number of the correlations we found were acknowledged by the participants of our empirical study.

Chapter 6

Comparing Metric Sets for Model Transformations

In Chapters 4 and 5, we defined metric sets for ASF+SDF and ATL respectively. We also defined metric sets for the model transformation languages QVTO and Xtend. In this chapter, we present a comparison of the metric sets we defined for ASF+SDF, ATL, QVTO, and Xtend. We will address similarities among these metric sets and we will also point out the differences among them.

6.1 Introduction

In Chapters 4 and 5, we defined metric sets for the model transformation languages ASF+SDF and ATL respectively. For two other model transformation languages, viz. Xtend and QVTO, we also defined metric sets and implemented tools to automatically collect them [171]. In Section 1.1.2.3, we showed that all four model transformation languages studied in this thesis have different characteristics. However, they all are designed for a similar task, i.e., model transformation. Therefore, it is to be expected that there is overlap between the metric sets. We noted earlier that the metric sets for the different languages contain conceptually similar metrics. In this section, we will address the similarities and we will also point out the differences among the metrics sets we defined. We have divided the overlapping metrics into six categories, viz., size metrics, function complexity

metrics, modularity metrics, inheritance metrics, dependency metrics, consistency metrics, and input/output metrics. For each of these categories, we describe which metrics of all four languages fit that category. Finally, we discuss a number of metrics that are language-specific. Again, we address in this chapter research question RQ_3.

> **RQ_3**: *How can metrics be used to assess the quality of model transformations?*

The remainder of this chapter is structured as follows. In Sections 6.2 to 6.8, we describe the overlapping metrics. A number of language-specific metrics are described in Section 6.9. Section 6.10 concludes this chapter. For related work, the reader is, again, referred to Section 4.6.

6.2 Size Metrics

The first category contains metrics for measuring the *size* of a model transformation. The size of a model transformation is determined by the *number of transformation functions* it comprises. All four languages have different types of transformation functions. In ATL there are rules, in QVTO there are mappings, in Xtend there are extensions, and in ASF+SDF there are transformation functions. Table 6.1 shows the metrics we have defined for measuring the number of transformation functions for each of the respective languages.

ATL	QVTO	Xtend	ASF+SDF
#Rules	#Mappings	#Extensions	#Transformation functions
#Helpers	#Helpers		

Table 6.1 Size metrics

ATL, Xtend, and ASF+SDF have different types of transformation functions, viz., ATL has matched rules, (unique) lazy matched rules, and called rules, Xtend has expression extensions, create extensions, and Java extensions, and ASF+SDF has non-traversal and traversal functions. We defined metrics for measuring the number of transformation functions for each of these function types, however, they are not shown in the table.

Besides the number of transformation functions, the size of a transformation is also affected by the *number of helper functions*. Again, metrics have been defined to measure the number of helper functions for each of the helper function types there are in ATL and QVTO. Note that Xtend and ASF+SDF do not have separate helper functions.

Another metric applicable for measuring the size of a model transformations is the *number of modules* it comprises. However, we decided to treat this metric as a modularity metric (see Section 6.4).

6.3 Function Complexity Metrics

The complexity of a model transformation is affected by the complexity of its parts, i.e., transformation and helper functions. The metrics we defined for measuring the complexity of these functions for the various languages are shown in Table 6.2.

ATL	QVTO	Xtend	ASF+SDF
#Elements per input/ output pattern			
#Parameters per called rule/operation helper	#Parameters per mapping/ helper	#Parameters per extension	Function val-in
#Bindings per rule			
#Operations on collections per rule/helper	#Operations on collections per mapping/ helper	#Operations on collections per extension	
#Variables per rule/ helper	#Variables per mapping/ helper	#Local variables per extension	
Helper cyclomatic complexity	Mapping/helper cyclomatic complexity	Extension cyclomatic complexity	

Table 6.2 Function complexity metrics

The complexity of a transformation function is among others determined by the number of (model) elements it takes as input and returns as output. In ATL this is measured using the metrics *number of elements per input pattern* and *number of elements per output pattern*. Called rules do not have an input pattern, so the number of (model) elements a called rule takes as input is measured using the metric *number of parameters per called rule*. Similarly, in QVTO this is measured using the metric *number of parameters per mapping*. For input model elements only *in* and *inout* parameters should be considered and for output model elements only *out* and *inout* parameters. For Xtend and ASF+SDF the number of parameters of a transformation function is measured by the metrics *number of parameters per extension* and *function val-in* respectively. Since transformation functions in ASF+SDF and also in Xtend can return only one output model element, there is no metric for measuring this. The complexity of helper functions for both ATL and QVTO can be measured using the metric *number of parameters per helper* as well.

In the body of a transformation function, the assignment of transformed source model elements to target model elements is performed. In ATL, target model elements are typically assigned values using bindings. Therefore, the *number of*

bindings per rule affects the complexity of a transformation rule in ATL. Since model transformations often need to deal with collections, i.e., sets of model elements, metrics have been defined to measure the *number of operations on collections* separately. A well known complexity measure is McCabe's *cyclomatic complexity* [161]. It measures the number of decision points in a function. Since transformation and helper functions also contain decision points, this metric is applicable to model transformations as well. This metric is not defined for ASF+SDF, since it has no constructs to explicitly define decision points. Although, it can be argued that every equation and every condition is a decision point.

ATL, QVTO, and Xtend allow defining local variables in transformation and helper functions. These are typically used to provide separation of concerns, i.e., to split a calculation in orderly parts. Therefore, the *number of local variable definitions* affects the complexity of a transformation function as well.

6.4 Modularity Metrics

All four languages have support for structuring model transformations by packaging transformation rules in modules, albeit that ATL has some restrictions on this. Table 6.3 shows the metrics related to modularity.

ATL	QVTO	Xtend	ASF+SDF
#Units	#Modules	#Modules	#Modules
#Imported units	#Imported modules	#Imported modules	#Import declarations per module
#Times a unit is imported	#Times a module is imported	#Times a module is imported	#Times a module is imported
#Helpers per unit	#Mappings/helpers per module	#Extensions per module	#Functions per module

Table 6.3 Modularity metrics

An indication of the modularity of a model transformation is the *number of modules/units* in that transformation. Note that this metric gives an indication of the size of a transformation as well and can therefore also be considered as a size metric. Measuring the number of modules alone is not enough to assess modularity. In a proper modular transformation, functionality should be well spread over the modules it comprises. Therefore, we defined metrics that measure the *number of transformation and helper functions per module* to determine the balance of the modules that comprise a model transformation. When the standard deviation of these metrics is high, the modules are not properly balanced. This could be intentional, but it could also be an indicator that the division of functions over modules should be reconsidered. Finally, in a proper modularized transformation, similar to traditional software, coupling between modules should

be low [168]. Coupling can be measured using the import metrics, they can be used to assess which modules rely heavily on other modules as well as which modules is heavily relied on. However, the import metrics alone do not suffice for assessing coupling between modules. When a module m_1 imports another module m_2, this is no guarantee that transformation or helper functions in module m_1 rely on transformation or helper functions in module m_2, i.e., the import may be obsolete. To acquire more insight in the coupling between modules, metrics that measure the dependencies between functions in modules are required. These dependency metrics are discussed in Section 6.6.

6.5 Inheritance Metrics

ATL and QVTO both have support for rule inheritance. The use of inheritance may affect the quality of a model transformation in a similar way as it affects object-oriented software [155]. A rule deeper in the rule inheritance tree may be more fault-prone because it inherits a number of properties from its ancestors. Moreover, in deep hierarchies it is often unclear from which rule a new rule should inherit from. To acquire insights into the rule inheritance relations in ATL and QVTO transformation, we defined the metrics shown in Table 6.4. QVTO has different forms of inheritance, i.e., *inherit*, *merge*, and *disjunct*. We defined metrics to measure all three forms.

ATL	QVTO	Xtend	ASF+SDF
#Abstract rules	#Abstract mappings		
#Children per matched rule	#Mapping inherits		
#Rule inheritance trees	#Mapping merges		
Depth of rule inheritance tree	#Mapping disjuncts		
Width of rule inheritance tree	#Mappings per disjunct		
#Overloaded helpers	#Overloaded mappings/helpers	#Overloaded extensions	#Signatures per function
	#Mappings per mapping name (overloadings)	#Extensions per extension name (overloadings)	#Equations per function
#Helpers per helper name (overloadings)	#Helpers per helper name (overloadings)		

Table 6.4 Inheritance metrics

Related to inheritance is overloading. Therefore, metrics regarding overloading are also included in this category. All languages support overloading of functions, except for ATL rules. For all four languages we defined metrics for measuring the *number of overloadings per overloaded function*. For ATL, QVTO, and Xtend

we also defined metrics for measuring the *number of overloaded functions*. This could also be measured for ASF+SDF by measuring the number of transformation functions that have more than one signature or equation.

6.6 Dependency Metrics

Transformation functions generally depend on other transformation functions. To measure this dependency, we measure *fan-in* and *fan-out* of transformation functions [134]. Fan-in of a transformation function is the number of times it is invoked by other transformation functions. Fan-out of a transformation function is the number of times it invokes other transformation functions. We consider fan-in/fan-out metrics for four types of transformation elements, i.e., transformation functions, helper functions, library functions, and modules. Since the fan-in and fan-out metrics for modules are indicators for coupling between modules, they are related to modularity as well. The metrics are shown in Table 6.5.

ATL	QVTO	Xtend	ASF+SDF
#Calls to rules	#Calls to mappings per mapping	#Calls to extensions	Function fan-in
#Calls from rules	#Calls from mappings per mapping	#Calls from extensions	Function fan-out
#Calls to helpers	#Calls to helpers	#Calls to extensions in other modules	Module fan-in
#Calls from helpers	#Calls from helpers	#Calls from extensions in other modules	Module fan-out
#Calls to helpers in other units	#Calls to mappings/helpers in other modules	#Calls to extensions in the standard library	
#Calls from helpers in other units	#Calls from mappings/helpers in other modules		
#Calls to built-in functions			

Table 6.5 Dependency metrics

6.7 Consistency Metrics

Metrics can be used to detect all kinds of inconsistencies in model transformations. We consider inconsistencies such as code smells and style violations. The metrics we defined for that purpose are shown in Table 6.6.

A typical inconsistency is an unused model transformation element. This can be a helper function, a transformation function, or an entire module. On a smaller scale there are parameters and local variables that may have been defined but are unused. We have defined metrics to detect all of these unused elements.

ATL	QVTO	Xtend	ASF+SDF
#Unused units	#Unused modules		
#Unused rules	#Unused mappings	#Unused extensions	
#Unused helpers	#Unused helpers		
#Unused input pattern elements			
#Unused parameters	#Unused parameters	#Unused parameters	
#Unused local variables	#Unused local variables	#Unused local variables	#Unused variables per module
#Types per variable name	#Types per variable name	#Types per variable name	#Types per variable name
#Calls to `println()`	#Calls to `log()`	#Calls to logging expressions	
#Calls to `debug()`	#Calls to `assert()`	#Calls to `syserr()`	

Table 6.6 Consistency metrics

All four languages allow definition of local variables. In ATL, QVTO, and Xtend these are local to a transformation function, in ASF+SDF these are local to a module. This means that a variable needs to be redefined if a variable of the same type is to be used in other transformation functions. This may lead to inconsistencies in variable naming, i.e., a variable name in one module can be related to a different type in another function. To detect such inconsistencies we defined metrics that measure the *number of types per variable name*.

A number of functions are available in all three languages that support logging of some kind. Typically, this is used for debugging purposes. The occurrence of calls to these functions may therefore indicate that the model transformation is still under development.

6.8 Input/Output Metrics

The last category of metrics is aimed at providing insight in the context of the transformation. They are shown in Table 6.7.

ATL	QVTO	Xtend
#Input/output models	#Input/output models	
#Involved metamodels	#Imported metamodels	#Imported metamodels
	#Imported metamodels per module	#Imported metamodels per module

Table 6.7 Input/output metrics

For ASF+SDF we did not consider the input and output (meta)models of the transformation. The number of metamodels involved in an ASF+SDF transformation could, for example, be measured by measuring the number of SDF modules that have no related ASF module that defines equations. However, it

may be the case that a module defines both a metamodel and transformation functions. In this case, by measuring the number of SDF modules that have no related ASF module, an incorrect metric value would be calculated. Since, there are more exceptions that need to be dealt with, decerning metamodels is too error prone. Therefore, Table 6.7 does not contain metrics related to metamodels for ASF+SDF. From the results of the empirical study presented in Section 5.5 can be concluded that model transformations involving more models and metamodels are more complex. Therefore we defined metrics that measure the number of models and metamodels involved in a transformation.

6.9 Language-specific Metrics

All four languages have different characteristics. We defined metrics that measure aspects of model transformations that are specific for each of the languages. Here, we will address only a few of them.

In ATL, a transformation function can have multiple input and output model elements. The metric *rule complexity change* measures the amount of output model elements that are generated per input model element. The input pattern of an ATL matched rule can be constrained using filter conditions. The metric *number of rules with a filter condition on the input pattern* measures this. Using such filter conditions a rule matches only on a subset of the model elements defined by the input pattern.

In QVTO it is possible to define intermediate classes and properties. These are used in the scope of the transformation only. These intermediate classes and properties may be useful for splitting calculations. Therefore, we measure the *number of intermediate classes/properties*. In QVTO and in ATL there are constructs for explicit trace resolution. These constructs may affect the understandability of a model transformation. Therefore, we defined metrics to measure the *number of trace resolution calls*.

Xtend has support for aspect-oriented programming using so-called *arounds*. We measure this using the metric *number of arounds*.

ASF+SDF is a combination of two languages, viz., ASF for term rewriting and SDF for syntax definition. Therefore, there are a number of metrics related to those activities that do not apply to model transformation in general. Alves and Visser defined a number of metrics for language development with SDF [153]. In Section 4.6, we already noted that these metrics cannot be used for measuring model transformations since the focus is different. There are also metrics specific for ASF+SDF that are applicable to developing model transformations as well. An example of this is the *number of start-symbols*. Also all metrics related to equations and their conditions are specific to ASF+SDF (see Section 4.3).

6.10 Conclusions

In Chapters 4 and 5, we defined metric sets for ASF+SDF and ATL respectively. We also proposed metric sets for measuring model transformations developed with QVTO and Xtend. All four model transformations we proposed metric sets for serve a similar purpose, i.e., performing model transformation. Therefore, conceptually similar metrics are defined for all of the languages. Since the languages have different characteristics, metrics specific for each of the languages are defined as well. In this chapter, we addressed similarities among these metric sets and we also pointed out the differences among them. It is to be expected that for other model transformation languages metric sets can be defined that overlap with the metric sets presented in this chapter.

Chapter 7

Visualization Techniques for Model Transformations

An approach to increase the quality of software, is to support their development and maintenance process by means of visualization techniques. Numerous such techniques support the maintenance process of traditional software. However, currently there are few visualization techniques available tailored towards analyzing model transformations. We present two complementary visualization techniques for analyzing model transformations. These techniques are mainly focused on increasing the understanding of model transformations. The first technique can be used to visualize the components of a model transformation and the dependencies between them. The second technique can be used to visualize the relation between a model transformation and the metamodels it is based on.

7.1 Introduction

In Chapters 4 and 5, we focused on assessing the quality of model transformations. One of the reasons for assessing the quality of model transformations is that model transformations of higher quality are, in principal, more maintainable. This is important, since model transformations are becoming more prominent and should not become the next *maintenance nightmare*. To facilitate a maintenance process and also the remainder of a development trajectory, often visual analysis

techniques are employed. Numerous such techniques exist for traditional software. However, currently there are few visualization techniques available tailored towards analyzing model transformations. Therefore, we address in this chapter research question RQ_4.

> **RQ_4**: *What visualization techniques can be employed to support the development and maintenance process of model transformations?*

The metrics we presented in Chapters 4 and 5 can be used to quickly acquire insights into the characteristics of a model transformation. For a more in-depth study, different techniques are required. Therefore, we present in this chapter four complementary visualization techniques for analyzing model transformations. These techniques are mainly focused on facilitating model transformation comprehension, since a significant proportion of the time required for maintenance, debugging, and reusing tasks is spent on understanding [115]. The first two visualization techniques are aimed at facilitating structure and dependency analysis. These techniques can be used to make the relations between the different components that comprise a model transformation explicit. Structure and dependency analysis has already been employed for analyzing different kinds of software artifacts. In this chapter, we show that it can be used for model transformations as well. Since these visualization techniques focus on the internals of a model transformation, they are related to internal quality (see Section 3.2). The other two visualization techniques are aimed at metamodel coverage analysis. Metamodel coverage analysis is the analysis of the relation between a model transformation and the metamodels it is defined on. These visualization techniques are specific for model transformations and therefore do not exist for traditional software artifacts. Since these visualization techniques focus on the externals of a model transformation, they are related to external quality. The visualization techniques we propose are not specific for a particular model transformation language. Some of the techniques may, however, be less applicable for certain languages. For example, a language with only implicit invocations, such as typical graph transformation languages, has no need for the dependency analysis technique.

The remainder of this chapter is structured as follows. In Section 7.2, we discuss the four visualization techniques for model transformations. Section 7.3 describes the toolset we use for automating the analysis. In Section 7.4, we demonstrate how the techniques can be applied in practice. Related work is described in Section 7.5. Section 7.6 concludes this chapter.

7.2 Visualization Techniques

In this section, the visualization techniques for analyzing model transformations are described. The structure and dependency visualizations are described in Section 7.2.1. The metamodel coverage visualizations are described in Section 7.2.2.

7.2.1 Structure and Dependency Analysis

In this section, two visualizations are described aimed at making the relations between the different parts that comprise a model transformation explicit.

7.2.1.1 Structure Analysis

Most model transformation languages have support for structuring model transformations by packaging transformation rules into modules [116]. We propose to visualize the import graph of a model transformation, i.e., the modules comprising the model transformations and their import relations. A (small) example of this visualization is depicted in Figure 7.1. The model transformation that is visualized here is described in Section 8.5.2.6. The arrows should be interpreted as *imports* relations. Such a high-level overview can act as a guide for navigating the model transformation, for instance during maintenance tasks. The visualization depicted in Figure 7.1 shows only the import relation between a pair of modules. In the remainder of this section, we will show another visualization where calls between modules are visualized as well.

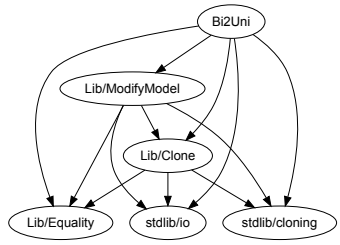


Figure 7.1 Structure visualization

7.2.1.2 Dependency Analysis

In Chapters 4 and 5, we discussed a number of metrics that can be used to acquire insights into the dependency relation of modules and transformation functions comprising a model transformation. These metrics measure the fan-in and fan-out of modules and transformation functions. While these metrics provide quick insights, they are merely numbers. Therefore, we propose another analysis technique that makes the call relation between transformation functions visible. For this purpose we use the tool ExTraVis [172]. A partial screen shot of the tool displaying the call relation of an, in this case Xtend, model transformation is depicted in Figure 7.2. The outer ring displays the modules that comprise the transformation. The second ring displays the different kinds of transformation function types per module. The inner ring displays the transformation functions

that comprise the transformation grouped per function type. The circle in the middle displays the calls that are being made between transformation functions. Callers and callees are connected to each other by an edge. The caller is on the green end of the edge, the callee is on the red end of the edge.

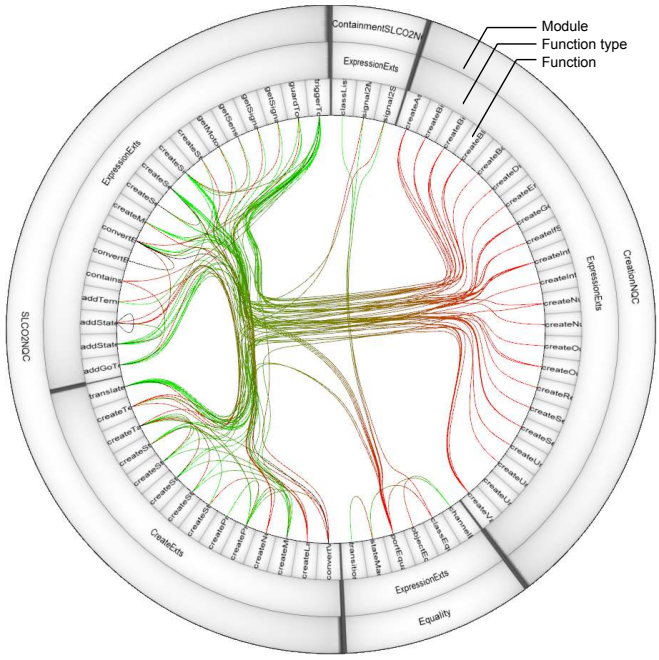


Figure 7.2 Trace visualization

Visualization of dependency data has various useful applications in the development and maintenance process of a model transformation. It may increase the understanding of a model transformation since the way transformation functions depend on each other and interact are made explicit. Also, undesired calls between transformation functions can be identified just by looking at the visual representation of the call graph. Similarly, transformation functions that are never called, and which may therefore be obsolete, can easily be recognized. We will show an example of this in Section 7.4.1. Metrics data can indicate whether there are transformation functions with a high fan-in or fan-out value. Using the visualization, fan-in/fan-out analysis can be performed in more detail. Not only can it be observed that there are some transformation functions with high fan-in/fan-out, the exact functions that cause this can be pointed out. Since the visualization groups transformation functions per module, calls between modules

can be examined as well. In this way, modules with low cohesion and high coupling [173] can be revealed.

7.2.1.3 Dynamic Dependency Analysis

ExTraVis has the option to provide additional detail with the call relation, such as for example the order and frequency of the calls. This feature could be exploited to analyze the dynamics of a model transformation by adding runtime data to the call relation. A declarative transformation language like ATL or QVTR does not require explicit rule invocation. Therefore, this technique may be less applicable for these languages. However, for imperative transformation languages it can have similar additional benefits as for traditional programming languages, such as feature location and feature comprehension. Since runtime data is input dependent, in the case of model transformation input model dependent, feature location may be useful for deriving characteristics of a possibly huge input model.

7.2.2 Metamodel Coverage Analysis

A model transformation is defined on the metamodel level, i.e., it transforms models conforming to a source metamodel to models conforming to a target metamodel. For some transformations it is required to transform all elements of the source metamodel, e.g., in case of language migration, whereas for other transformations it suffices to transform only a subset of the elements of the source metamodel, e.g., in case of partial refinement. Similarly, some transformations generate model elements for every metamodel element in the target metamodel, whereas other transformations generate model elements for a subset of the metamodel elements only. To acquire insight in the parts of the source and target metamodel that are *covered* by a model transformation, we propose two visualization techniques for coverage analysis.

7.2.2.1 Metamodel Coverage

Figure 7.3 shows an example of metamodel coverage visualization. The figure shows a metamodel, where the metaclasses and references that are covered by the transformation are colored grey. Attributes that are covered are underlined. An input metamodel element is covered if it serves as input for a transformation function in the transformation. An output metamodel element is covered if it is generated as output by a transformation function in the transformation. The metamodel elements that are considered in the coverage analysis are metaclasses, attributes, and references. The name of a metaclass is postfixed with *(a)* when it is an abstract metaclass.

Coverage analysis can be used to analyze the completeness of a transformation. Using the visualization, it can be observed whether all the metamodel elements

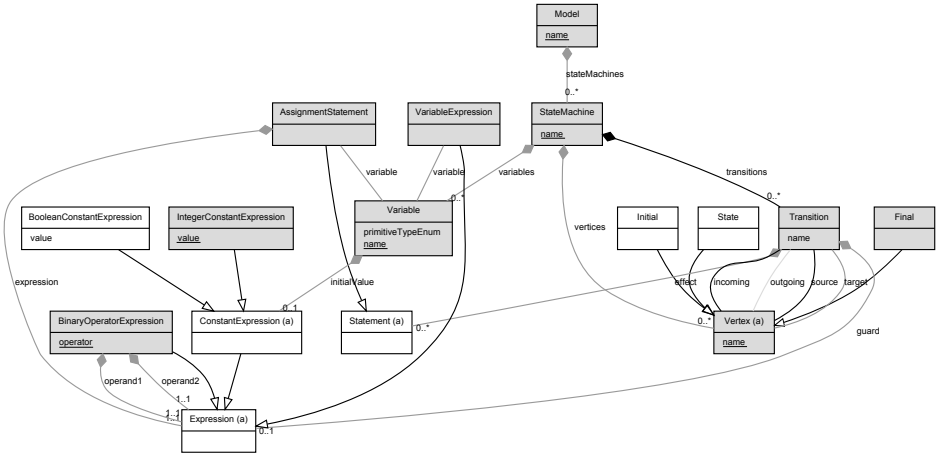


Figure 7.3 Metamodel coverage visualization

from the source model that should be transformed are in fact covered by the transformation. Conversely, it can be observed whether all metamodel elements from the target metamodel that should be generated are covered by the transformation. Of course the coverage visualization can also be used to detect metamodel elements that should *not* be covered by a transformation. Coverage analysis can also be used to facilitate the construction of test sets for model transformations. It can be used to determine the model elements the test set should focus on. Test sets do not have to contain model elements that are not covered by a model transformation, since they will not be transformed anyway.

There may be several reasons why a particular metamodel element is not covered by a transformation. First, coverage of a metamodel element may not be required for the transformation task at hand. If the requirements of the transformation dictate that only part of the metamodel needs to be transformed, it is not necessary that the remainder of the metamodel is covered by transformation functions. Second, the transformation may be incomplete, e.g., a transformation function for the metamodel element that is not covered has not yet been implemented. Third, the metamodel element may be an abstract metaclass. Some model transformation languages do not allow an abstract metaclass to be the input of a transformation function since they cannot be instantiated. However, there are transformation languages that do allow this. For example, in ATL it is possible to use an abstract metaclass as input pattern of an abstract transformation rule that can be extended to transform the non-abstract specializations of the abstract metaclass. In this case, the abstract class is covered by the transformation, however, by a rule that will not be executed itself. Last, a bidirectional reference

may have been used in one direction only. A bidirectional reference between two metaclasses is a reference that is navigable from both metaclasses. It may have a different name in each of the metaclasses. Two examples of this are depicted in Figure 7.4. The references are navigable from metaclass *Vertex* using the names *incoming* and *outgoing*, and from metaclass *Transition* using the names *target* and *source* respectively. In our visualization, we treat a bidirectional reference as two distinct references (see Figure 7.3). In this way, it can be determined in which direction the reference is used in the transformation. Therefore, it is possible that the reference is not covered by the transformation in the other direction.

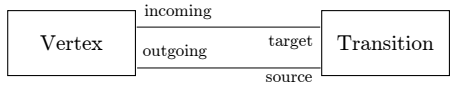


Figure 7.4 Bidirectional references

7.2.2.2 Metamodel Coverage Relation

Projecting coverage on a metamodel diagram is one way of visualizing metamodel coverage that may provide useful insights into a model transformation. It is however impossible to trace the coverage back to the transformation, i.e., the relation between a transformation element and the metamodel element it covers is invisible in the diagram. Therefore, we introduce another coverage visualization technique to overcome this matter. Figure 7.5 shows an example of a visualization where the relation between a transformation element and the metamodel elements it covers is made explicit. The model transformation that is visualized here is described in Section 8.5.3.1. The visualization shown here is part of a screen shot from the tool TreeComparer [174]. TreeComparer has originally been designed for comparing hierarchically organized data sets. Therefore, the metamodels and transformation are structured hierarchically in the image.

In the top part of the image, the metamodels are shown. The top row is for grouping all metamodels together, it is labeled *Metamodels*. The second row separates the input metamodels from the output metamodels. The input metamodels are shown to the left, the output metamodels to the right. The columns are labeled *Input* and *Output* respectively. The third row shows the names of the metamodels as they are referred to in the transformation. In this case, there is one input metamodel labeled *MM_IN* and one output metamodel labeled *MM_OUT*. The fourth row shows the packages that are present in the metamodels, in this case both the input and the output metamodel contain one package only, viz., *slco* and *promela* respectively. The fifth row shows the metaclasses that are present in the metamodels. In some cases, the columns are colored red and they stretch to the sixth row as well. In these cases, the metaclass is not covered

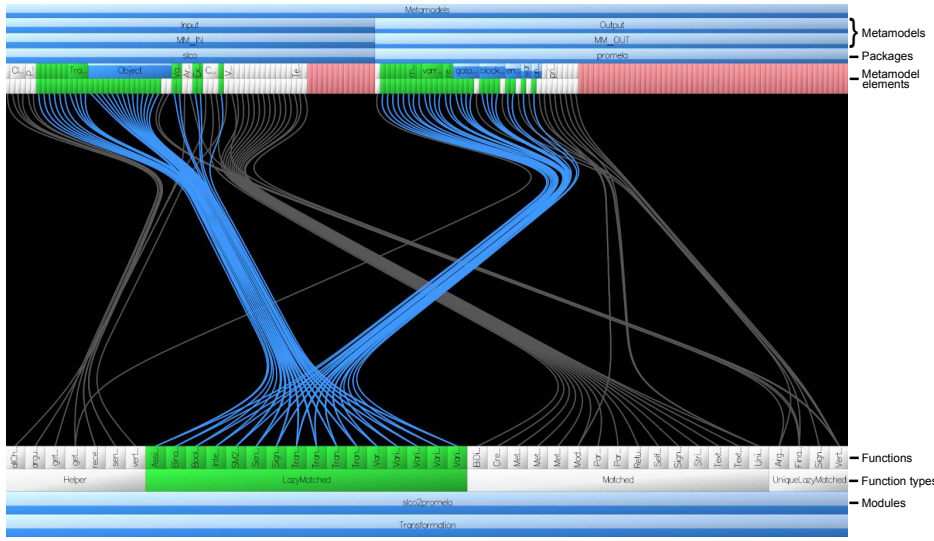


Figure 7.5 Metamodel coverage relation visualization

by the transformation. The TreeComparer tool supports many-to-one relations only. In a model transformation the coverage relation is typically many-to-many. Therefore, there is a dummy in the sixth row for every transformation rule that covers a metaclass.

In the bottom part of the image, the model transformation is shown. The bottom row is for grouping all the modules of the transformation, it is labeled *Transformation*. The second row shows the modules that comprise the transformation. In this case, there is only one module, named *slco2promela*. The third row is used for grouping the different kinds of transformation elements. In this case, an ATL model is visualized. The transformation elements that are available in ATL are helpers, matched rules, lazy matched rules, unique lazy matched rules, and called rules. These are shown on the third row. The fourth row shows the actual transformation elements.

There is a line between a transformation element and a metaclass if and only if the metaclass is covered by the transformation element. Note that in this visualization attributes and references are not considered.

TreeComparer has support for selecting a part of the metamodel elements, transformation elements, or relations. In Figure 7.5, for example, a selection is made of all the input and output metamodel elements covered by the lazy matched rules of an ATL model transformation. The tool also allows zooming in on a selection to get a more detailed view of that selection. This selection and zooming

functionality enables detailed study of coverage of part of a transformation or the involved metamodels. Highlighting the transformation functions that cover a (group of) metamodel element(s) facilitates navigation of a transformation, which may increase the understanding of a transformation. Moreover, visualization of the coverage of certain parts of a transformation can be used by developers to determine whether the selected transformation functions cover the required parts of the involved metamodels. In this way, coverage relation visualization helps in finding errors during development.

Model transformations can be defined on multiple input or output metamodels. It may be the case that another transformation has to be developed for one of the involved metamodels. By selecting this metamodel, the transformation functions that cover this metamodel are highlighted. In this way, coverage relation visualization can be used to identify parts of a transformation eligible for reuse. Similarly, when a metamodel is developed based on an existing one, this visualization technique can be used to identify reusable transformation functions.

The visualization shows coverage of both the source and the target metamodel element of a transformation function. In this way, the relation between source and target metamodel elements is visualized. This relation can be used to predict the effect of modifications to a source model on the corresponding target model when the model transformation is applied to the modified source model. This is referred to as impact analysis [175]. Impact analysis has a number of useful applications. When a source model needs to be refactored for some reason, multiple alternative refactorings may be applicable to achieve the same goal. Impact analysis can be applied to determine which of the alternatives is least invasive for the target model. In this case, impact analysis can also be used to facilitate determining the parts of the target model that have to be retested to ensure its correctness.

Coverage relation analysis can also be used to assist in the process of co-evolution of metamodel and transformation [176]. It can be used to determine whether the changes to a metamodel affect the transformation, i.e., whether the changes are breaking or non-breaking [177]. If the meta-classes that are covered by a transformation do not change, there is no problem and the transformation is still usable with the evolved metamodel. However, if there are meta-classes that are covered by a transformation that do change or are removed, then the transformation will have to evolve as well. The coverage relation visualization technique can be used to determine which transformation functions are affected by changed or removed meta-classes. In Section 7.4.2, we will elaborate a use case where coverage analysis is used in the process of co-evolution of metamodel and transformation.

Another application of the visualization is facilitating model traceability, i.e., tracing the source model elements that a target model element is generated from. When an error manifests in a target model, e.g., a run-time error in an

executable target model, this may be the result of an erroneous source model. The visualization can be consulted to establish which source metamodel elements are possible origins of a target metamodel element. This information can be used to determine the source model elements that a target model element is generated from. In this way, the visualization facilitates providing feedback of results from analyzing a target model to a source model. Recall that the source model of a model transformation is typically a domain-specific model. Application of this visualization technique, therefore, makes implementing tools for a DSL, such as a type checker or a simulator, less urgent, since the results of dedicated tools for these purposes can easily be traced back to the source model. Note, however, that this is merely a palliative and that tools specific for the source language are always preferred.

The two coverage visualization techniques described in this section are complementary rather than competing alternatives. The first coverage visualization technique, i.e., where coverage is projected on a diagram representing an input or output metamodel of a transformation has a familiar layout. Besides the coverage of metaclasses, it also shows coverage of attributes and references. However, there is no visible relation between a metamodel element and the transformation element it is covered by. This shortcoming is solved by the coverage relation visualization technique, where these relations are made explicit. However, this visualization is less detailed. It only shows coverage of metaclasses.

7.3 Toolset

We have implemented a toolset such that the visualization techniques presented in Section 7.2 can be applied automatically. The transformation languages we have implemented the toolset for are ATL, QVTO, and Xtend. Figure 7.6 depicts the extensible architecture of the toolset. In the first step, the code of a model transformation is parsed, resulting in a model that represents the abstract syntax tree (AST) of that transformation. This model can be represented in different formats depending on the transformation language. For ATL and QVTO this is an Ecore model, for Xtend the model is represented using POJOs[1]. In the second step, the data required for the desired visualization technique is extracted from the AST model and represented as another model. For ATL this extraction is implemented as an ATL model transformation, for QVTO as a QVTO model transformation, and for Xtend as a Java program. Note that for the coverage visualization techniques, the source and target metamodel of the transformation need to be provided as input as well. In the third and last step, the analysis model is transformed into the input format of the tool used for the analysis. Typically,

[1] Plain Old Java Objects

this step is performed using some form of pretty-printing. We used a combination of Xpand templates and Java programs to perform the pretty printing.

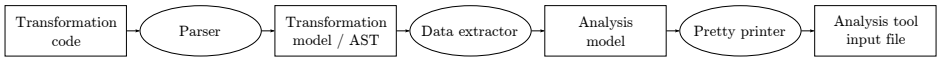


Figure 7.6 Tool set architecture

The advantage of this architecture is that it is extensible. To enable the visualization techniques for a new model transformation language, only the data extractors have to be implemented. The pretty-printers can be reused. In principal, also a parser needs to be implemented. However, a parser is typically provided with the implementation of the language. When implementing a new analysis technique, extractors need to be implemented for each of the languages as well as a pretty-printer for that technique.

7.4 Case Studies

We describe our experiences with the visualization techniques in an ongoing MDE project. In Chapter 8, we describe our experiences with the development of a DSL. This language is accompanied with a set of model transformations, implemented using Xtend and ATL, that can be used to refine models created with the language. These model transformations are the subjects of our study.

7.4.1 Detecting Obsolete Transformation Elements

Metrics were extracted from one of the Xtend transformations using our metrics extraction tool for Xtend (see Section 6). Table 7.1 shows an excerpt of the generated metrics report.

Metric	Value
# Modules	4
# Extensions	94
# Uncalled Extensions	18
# Uncalled Modules	2

Metric	Min.	Max.	Mean	Median	Std. Dev.
# Extensions per Module	3	58	23,5	16,5	24,45
# Unique Extension Names per Module	2	40	17	13	17,17
Extension Fan In	0	14	2,63	1	3,44
Extension Fan Out	0	30	2,63	1	5,04
Module Fan In	0	48	12,25	0,5	23,84
Module Fan Out	0	49	12,25	0	24,5

Table 7.1 Excerpt of the metrics report

The metrics indicate that the transformation contains eighteen extensions (transformation functions) that are never called by other extensions. These extensions are divided over two modules, since the metrics indicate that there are two modules that contain extensions that are never called from other modules and that there are no modules that contain no extensions. From a maintenance perspective it is useful to determine which extensions and modules are uncalled, such that the transformation can be adapted if required. For this purpose, the dependency graph can be employed. The dependency graph of the transformation is depicted in Figure 7.7.

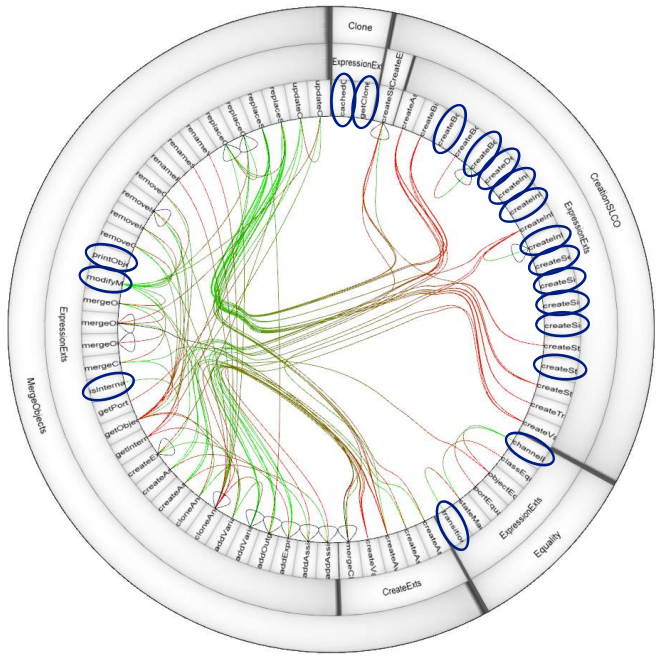


Figure 7.7 Using dependency analysis to detect obsolete transformation elements

The eighteen extensions that are never called can easily be recognized using the ExTraVis tool. For illustratory purposes, they have been marked in the figure. Note that some of these extensions do have incoming calls. These are recursive calls, i.e., calls to itself, that are excluded when determining uncalled extensions. Note that ExTraVis has an option to hide self calls. When zooming in to module level with the ExTraVis tool (not shown here) it can easily be observed that the modules named MergeObjects (left) and Clone (top) contain extensions that are never called by extensions from different modules. From the import graph depicted

in Figure 7.8 can be concluded that the transformation is built up hierarchically. The MergeObjects module is the top module that only calls functions lower in the import hierarchy. Since it does not receive calls from modules lower in the hierarchy, the module is considered uncalled. As for the other uncalled module and extensions, we presented our findings to the developer of the transformation. He indicated that the Clone module is a leftover from an earlier version of the transformation and that it can (and should) be removed. The CreationSLCO module (right) is a module that is used by six other transformations as well. We also analyzed these transformations. It turned out that four of the extensions in the CreationSLCO are never called by any of these transformations. Also these turned out to be obsolete and should be removed.

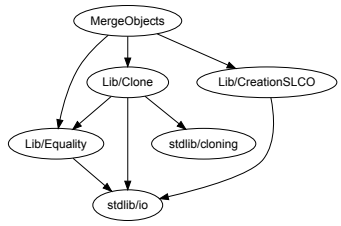


Figure 7.8 Structure visualization

7.4.2 Metamodel and Transformation Co-evolution

During every iteration in the language development, the language evolves. This is reflected in the metamodel. Since the refining model transformations are based on that metamodel, they have to evolve as well. To identify the transformation rules and helpers that are affected by the evolution, we employed the coverage relation visualization technique described in Section 7.2.2.2.

Figure 7.9 shows the relation between the evolved metamodel on top and a model transformation that needs to co-evolve on the bottom. On the top right, there are a number of metaclasses that have no outgoing edges. This means these metaclasses are not covered by any of the transformation functions or helpers in the transformation. These metaclasses are all the metaclasses that were added to the metamodel due to the evolution of the language. Since, in this case, all metaclasses need to be covered by the transformation, it is easily observed for which metaclasses transformation rules should be added. On the bottom of the figure there are ten matched rules that have no outgoing edges. Matched rules have to match on a metaclass. This implies that these matched rules match on metaclasses that are not present in the metamodel. Therefore, they are obsolete and may be removed from the transformation. After inspection

of the transformation it turned out that these rules all match on metaclasses that were removed from the metamodel due to the evolution of the language.

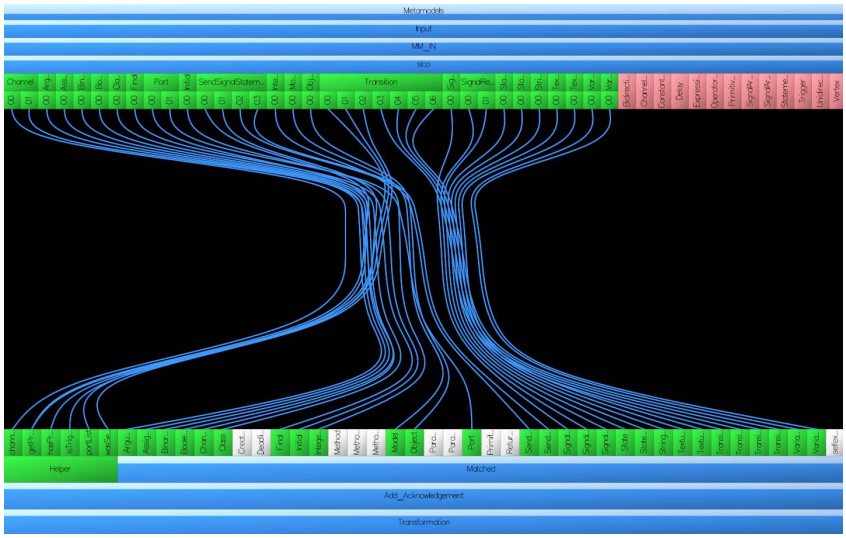


Figure 7.9 Using coverage analysis for transformation co-evolution

Already in this example with a relatively small metamodel and ditto model transformation, the coverage relation visualization turned out to be of great help. We expect that it will prove even more beneficial in cases with larger metamodels and model transformations. The danger with larger data sets is that the visualization will become cluttered. However, TreeComparer was developed for visualizing huge data sets.

7.5 Related Work

Anastasakis et al. recognize that the quality of model transformations is crucial for the success of MDE [178]. Suitable validation and analysis techniques should therefore be applied to them. They consider model transformations as a special kind of models that can be subject to existing model analysis techniques. The authors use Alloy, a textual declarative modeling language based on first-order relational logic that is accompanied with a fully automated analysis tool. A model transformation and its source and target metamodel are transformed into an Alloy model and subsequently analyzed using the analysis tool. The analysis tool can among others be used to simulate the model transformation and to check whether certain assertions hold.

Verification of metamodel coverage has been studied by others before. Wang et al. see metamodel coverage as an aspect of validation and verification that should be considered for model transformations [179]. They state that it is important because it allows identification of the scope of a model transformation. They base coverage on the core MOF structural constructs, viz., class, feature, inheritance, and association. The coverage analysis we propose does not consider inheritance coverage, although this can be derived from the coverage visualization. In the paper, they present a prototype implementation for the Tefkat transformation language [180].

McQuillan and Power address metamodel coverage in the context of testing model transformations [181]. They base their coverage criteria on criteria for coverage of UML class diagrams. The main difference between our approach and the two aforementioned approaches is that we use visualization techniques to present coverage instead of listing the (un)covered metamodel elements.

Also Planas et al. present a metamodel coverage technique, however for ATL only [182]. Their technique is aimed at determining full coverage of a metamodel by a set of ATL matched rules. They state that a set of ATL rules is source-covering when all elements of the source metamodel may be navigated through the execution of these rules. Similarly, they state that a set of ATL rules is target-covering when all elements of the target metamodel may be created and initialized through the execution of these rules. The result of their analysis is a yes or no answer stating whether all source and target model elements are covered. In their paper, they also present a technique for determining the *weak executability* of an ATL rule. An ATL rule is weakly executable when it can be safely applied without breaking the target metamodel integrity constraints.

Von Pilgrim et al. present a technique for visualizing chains of model transformations [183]. In this visualization, diagrammatic representations of models are shown on two-dimensional planes in a three-dimensional space. Lines between these planes connect source model elements to target model elements. This work is related to our metamodel coverage visualization technique. The main difference is that our visualization is based on the relation between transformation and metamodels, whereas their visualization focuses on relations between models based on a transformation. They do not visualize the model transformation themselves. If we apply our visualization to models instead of metamodels, we can perform similar traceability analyses. However, they are capable of visualizing transformation chains, whereas our visualization is currently limited to one transformation only. Since their visualization provides diagrammatic representations of models, this may lead to scalability issues when huge models or long transformation chains need to be analyzed.

Küster acknowledges that there is a strong need for techniques and methodologies that deal with developing model transformations because of their wide

application area and the future importance of MDE. In [184], he presents an approach to the validation of model transformations, focusing on termination and confluence. These two properties together ensure that a model transformation always generates a unique target model given a source model. Termination and confluence are properties of fundamental importance in rewriting systems and therefore also in rule-based model transformation languages. To ensure termination and confluence, a set of criteria is provided that has to be checked during the development of a transformation. The author provides proofs for each of the criteria.

7.6 Conclusions

We have addressed the necessity for analysis techniques for model transformations to, among others, assist in the maintenance process. In this chapter, we proposed four such techniques. First, we proposed two different ways to visualize dependencies between the components of a transformation. Second, we proposed to analyze the coverage of the metamodels that a transformation is defined on by means of two different visualizations. The proposed techniques complement each other rather than being competing alternatives. A toolset has been implemented to automate the visualization techniques.

We have actively used both visualization techniques in an MDE project. They have shown to be of great assistance when dealing with evolution or maintenance of model transformations. To reap the full benefits of the techniques, they should be embedded in (existing) model transformation tools.

Chapter 8

Fine-grained Model Transformations

Traditionally, the state-space explosion problem in model checking is handled by applying abstractions and simplifications to a verification model. We propose a model-driven engineering approach that works the other way around. Instead of abstracting from a concrete model, we propose to refine an abstract model to concretize it. We propose to use fine-grained model transformations to enable model checking of models that are as close to the implementation model as possible. To demonstrate our approach we developed a DSL aimed at modeling the structure and behavior of distributed communicating systems. We also implemented two model transformations to different formalisms, viz., one for execution, and one for model checking. The DSL and the formalisms for execution, and model checking have different semantic characteristics. Therefore, we implemented a number of fine-grained model transformations that bridge the semantic gaps between our DSL and both formalisms. We describe in this chapter how the development and evolution of the model transformation has been facilitated by the acquired insights and developed techniques presented earlier in this thesis.

8.1 Introduction

In Chapter 1, we noted that one of the goals of MDE is to increase the quality of software. Generating reliable code from models specified using a DSL is one aspect of this. To increase the reliability of generated code, formal methods

such as verification can be used. Model checking is an automated verification technique that checks whether a formally specified property holds for a model of a system [128]. An exhaustive state space search is performed by an automated model checker to determine whether a property holds in a finite state model of a system. Often, this state space is huge and model checking is no longer a feasible approach for verification. Traditionally, abstractions and simplifications are applied to the model to enable model checking in such cases [185–187]. We propose an MDE approach to enable model checking that works the other way around. Instead of starting with a large model and iteratively simplifying it, we start with a small model and iteratively refine it.

In a typical MDE development process, domain-specific models are refined using model transformations until a model is acquired with enough details to implement a system [188]. To increase the reliability of the final system, model checking can be employed. Because of the aforementioned state-space explosion problem, model checking the final system may be infeasible. We propose to define a model transformation that transforms the (refined) domain-specific models to models suitable for model checking. Using this model transformation, model checking can be applied to the domain-specific models in every stage of the refinement process. While model checking the final system may be infeasible, this approach enables verification of intermediate models close to the implementation.

In this chapter, we demonstrate this approach using a DSL for modeling systems consisting of concurrent, communicating objects. This DSL has an intuitive graphical syntax to model the structure and behavior of a system. It offers constructs such as synchronous communication over lossless channels to enable development of concise models. To execute models, we implemented a chain of transformations that transform models specified using our DSL to NQC, a restricted version of C [189]. The semantic properties of this implementation platform differ from those of our DSL, i.e., some construct that are available in our DSL have no direct counterparts on the implementation platform. For instance, NQC does not offer constructs such as synchronous communication and lossless channels. Instead, communication is asynchronous and takes place over a lossy channel. To enable transformation from our DSL to the implementation platform, the semantic gaps between the two platforms need to be bridged. Therefore, we added a number of constructs to our DSL and implemented a number of transformations that can be used to stepwise refine models to align the semantic properties of the DSL with the implementation platform. These transformations replace the constructs in a model that are not offered by the implementation platform by constructs that it does offer. The observable behavior of the models should be preserved by the model transformations. A final transformation transforms the resulting model to executable code.

To enable model checking of the (intermediate) domain-specific models, we also implemented a model transformation from our DSL to Promela, the input language for the model checker Spin [190]. Our first experiments showed that verification of the models generated by the refining transformations using Spin was infeasible due to state-space explosion. We concluded that the change induced on the models by the transformations was too large, i.e., the model transformations were too coarse-grained. Therefore, we split up the coarse-grained transformations into more fine-grained ones. The impact of such a fine-grained transformation on a model is smaller, i.e., the model does not change as drastically. This is reflected in the increase of the state-space size that is searched by Spin. Using this approach, intermediate models generated by the fine-grained transformations can be model checked almost all the way up to the models that can be executed, because the state space stays within reasonable bounds.

The design decisions we took during implementation of the model transformations have been influenced by the insights we acquired from the research presented earlier in this thesis. For instance, we chose to bridge the platform gaps one at the time instead of all at once from start, since this leads to smaller model transformations, which, in principal, should be more understandable and modifiable (see sections 4.5 and 5.5). Usage of the DSL and development of the model transformations produced new insights that lead to evolution of the DSL. Evolution of the DSL implies that the model transformations based on it have to evolve as well. This co-evolution process was supported by the visualization techniques presented in Chapter 7. This leads to the following research question.

> **RQ$_5$**: *How can the acquired insights and developed techniques be applied to facilitate the development and maintenance of model transformations?*

The remainder of this chapter is structures as follows. Our approach to enable model checking of models as close to the implementation model as possible is discussed in Section 8.2. In Section 8.3, we introduce SLCO, the DSL we used for our experiments. The languages we used for execution and for model checking, viz., NQC and Promela respectively, are discussed in Section 8.4. Section 8.5 describes the transformations for refining models created using the DSL as well as transformations for transforming them to different formalisms. The experiments we conducted are presented in Section 8.6. Related work is described in Section 8.7. Section 8.8 concludes this chapter.

8.2 Exploring the Boundaries of Model Verification

Our goal is to generate reliable code from models specified using a DSL. To increase the reliability of generated code, formal methods such as verification

can be used. To ensure that the same model is verified and executed, models specified using the DSL should automatically be transformed to models suitable for these purposes. In this way, these models do not have to be created by hand. Such transformations thus enable the use of formal methods without having to create models suitable for that purpose separately. This has the advantage that engineers do not have to learn the syntax and semantics of different languages. Moreover, manual transformation is a slow and error-prone task.

Often, the DSL and the implementation platform have different semantic characteristics. Therefore, the semantic gap between the two formalisms needs to be bridged (see Chapter 2). We propose to use model transformations to refine a DSL model in such a way that the semantic properties of the DSL and the implementation platform are aligned. In this way, an abstract DSL model becomes concrete and transformation from the refined (concrete) DSL model to the implementation model is merely a syntactical transformation.

To enable verification of a DSL model, a transformation from the DSL to a formalism for verification, e.g., a model checking formalism, should be implemented. Using this transformation, it is possible to verify whether both the abstract and the concrete DSL models fulfill their requirements. From the experiments presented in Section 8.6, we concluded that verification of an abstract model poses no problems. However, verification of a concrete model is infeasible. The verification takes too much time and needs too many resources.

The transformations used to refine the abstract DSL models produce intermediate models. These models can also be transformed to a verification formalism. By verifying the intermediate DSL models, it is possible to verify models that are more concrete. This approach is schematically depicted in the top half of Figure 8.1. The check marks indicate models that can be verified, whereas the crosses indicate models that cannot be verified. Our experiments, which are presented in Section 8.6, showed that it is possible to verify some of the intermediate models. However, the most concrete model that can be verified is still not concrete enough. The reason for that is that the change induced on the models by the transformations is too large, i.e., the transformations are too coarse-grained. Therefore, we propose to use more fine-grained transformations to enable verification of more concrete models. This can be achieved by splitting existing transformations into smaller parts. In this way, more intermediate models are generated that can be verified. This approach is schematically depicted in the bottom half of Figure 8.1. Using this approach, it is possible to verify models that are closer to the concrete model. By replacing the coarse-grained transformations T_{rs} and T_{abp} from Figure 8.1 by the more fine-grained transformations T_{arg}, T_{uni}, T_{ll}, T_{time}, T_{ex}, T_{merge}, and T_{int}, for instance, the state space of the intermediate model M_{ex} can be explored, instead of that of the less concrete model M_{rs}. The aforementioned transformations are explained in Sections 8.5.1 and 8.5.2. The

example shown in Figure 8.1 is an illustration of one of the experiments presented in Section 8.6. In different cases, the transformation steps as well as which of them can be verified will vary.

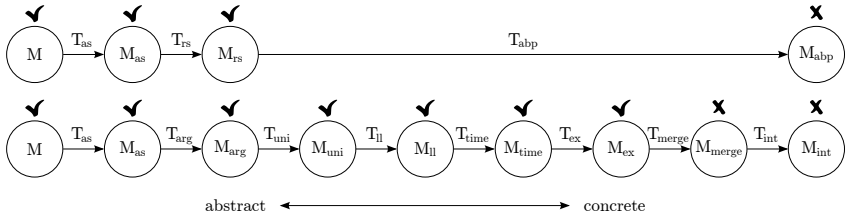


Figure 8.1 Verification of intermediate models

The most concrete model that can be model checked may still not be close enough to the implementation model. An attempt can be made to split the transformations into even smaller parts. If this is not possible, another possibility may be to perform partial refinement, i.e., to apply the model transformation to part of the model only. To obtain a complete target model, the remainder of the source model should be copied to the target model. Obviously, the model should be fitted for partial refinement for this technique to be applicable. Since the refinement, in this case, is applied to a small part of the model, this will most likely result in models that give rise to smaller state spaces. Using partial refinement, the boundaries of what can be verified using model checking can be explored even further.

Using fine-grained transformations has some positive side-effects. The change induced on a model by a fine-grained transformation is smaller than the change induced on it by a coarse-grained one, i.e., fine-grained model transformations affect a smaller part of the model. Therefore, it is to be expected that fine-grained model transformations are smaller than course-grained ones. In Sections 4.5 and 5.5, we observed that smaller model transformations are considered to be more understandable and modifiable. Also, fine-grained transformations have proven to be more reusable than course-grained ones during our experiments. Furthermore, fine-grained transformations enable shuffling the order in which they are applied. This order affects the output model, i.e., some sequences of transformations may lead to more efficient implementations than others. However, it may be that not all orderings are allowed because the precondition of some transformations may be in conflict with the postcondition of others.

8.3 Domain-Specific Language

We demonstrate the approach presented in Section 8.2 using a DSL we developed. This DSL is called *Simple Language of Communicating Objects* (SLCO). In Section 8.3.1, we describe this DSL. Over time, our DSL has evolved. In Section 8.3.2, we motivate and describe some of the changes to the language.

8.3.1 Simple Language of Communicating Objects

SLCO provides constructs for specifying systems consisting of objects that operate in parallel and communicate with each other. Since SLCO uses concepts that are computer science oriented, it is a horizontal DSL [191]. Figure 8.2 shows the SLCO metamodel.

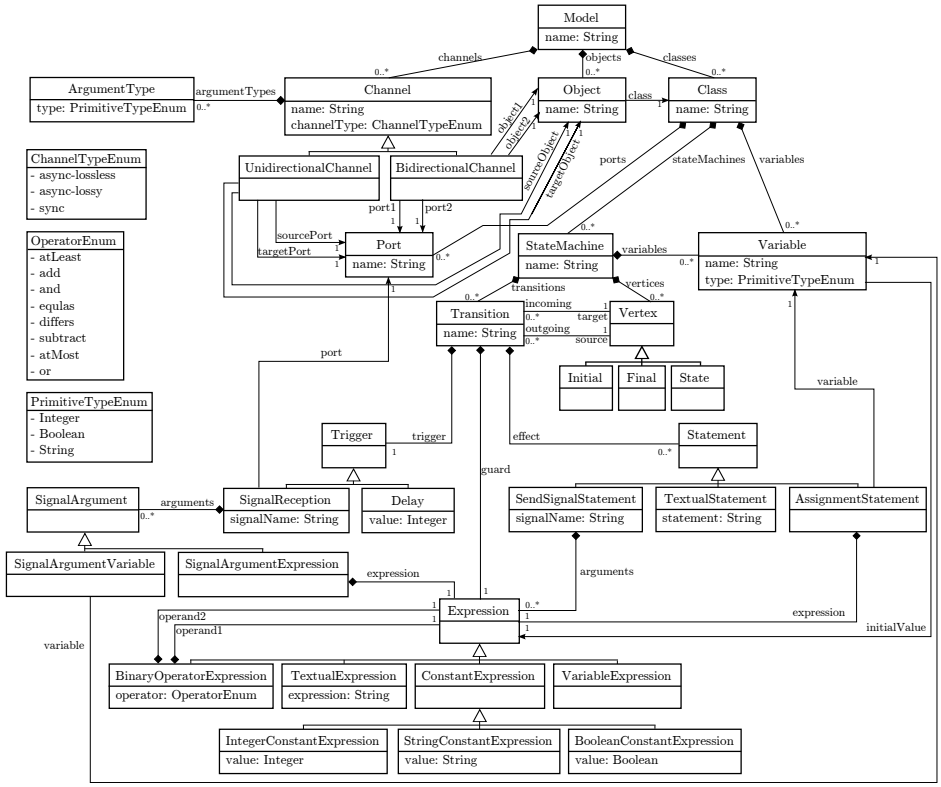


Figure 8.2 SLCO metamodel

The semantics of SLCO is formally defined by means of rewrite rules [192].

Here, we will discuss the semantics of SLCO informally only. An SLCO model consists of a number of classes, objects (instances of these classes), and channels. Channels are used to connect a pair of objects such that they can communicate with each other. An example of this is shown in Figure 8.3. The objects *Left*, *Middle*, and *Right*, which are instances of classes S and M, can communicate over channels *M2L* and *M2R*. A class describes the structure and behavior of

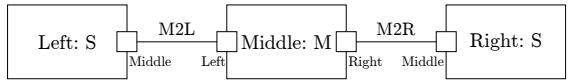


Figure 8.3 Three objects connected by channels in SLCO

its instances. A class has ports and variables that define the structure of its instances, and state machines that describe their behavior. Ports are used to connect channels to objects. Figure 8.3 shows that both instances of class S have a port *Middle* connecting them to channels *M2L* and *M2R*, and that the instance of class M has two ports, *Left* and *Right*, connecting it to the same channels. A state machine consists of variables, states, and transitions. A transition has a source and a target state, possibly a guard or a trigger, and a number of statements that form its effect. A guard is a boolean expression that must hold to enable the transition from source state to target state. There are two types of triggers, viz., a deadline and a signal reception. If the amount of time specified by a deadline has passed or if a signal is received, the transition that has such a trigger is enabled. There are two ways in which reception of a signal can be handled. A signal has arguments. First, the values of the arguments can be stored in variables. Second, expressions can be defined that define the values the arguments should have in order to be received. This is referred to as conditional signal reception. When a transition is made from one state into another state, the statements that constitute the effect of the transition are executed. There are statements for assigning values to variables and for sending signals over channels. Figure 8.4 shows an example of a state machine.

8.3.2 DSL Evolution

During the development of the language and implementation of the transformations presented in Section 8.5, SLCO has evolved. This evolution can be divided into four categories. We describe for each of these categories the main influences on the evolution of our DSL.

The problem domain for which we designed a DSL consists of concurrent, communicating objects. Therefore, the most important requirement for SLCO is that it can describe such objects at an appropriate level of abstraction. For example, we chose to offer synchronous communication primitives, since this leads

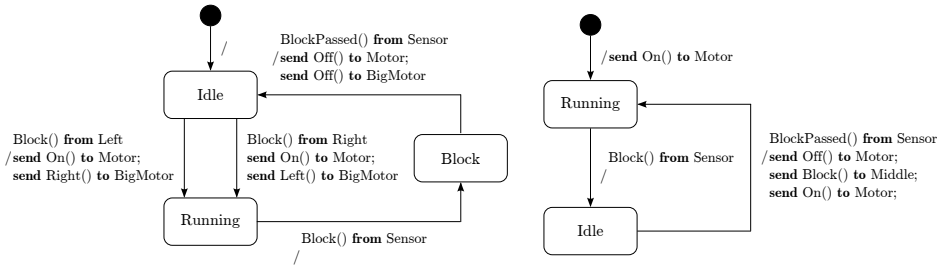


Figure 8.4 Two state machines in SLCO

to concise models. Also for this reason, we opted for a concrete syntax that is graphical and resembles the well-known syntax of the UML [72]. These choices were made to effectively describe problems in the *problem domain*. The influence of the problem domain on the evolution of a DSL is to be expected, since the language is focused on that domain.

In our case, the *target platforms* also have an impact on the evolution, since our language does not have its own execution platform, nor is it embedded in another language. If a transformation to, for example, an execution platform is impossible due to language mismatches, it is impossible to execute a model created using the DSL. Therefore, our DSL has evolved to eliminate unresolvable mismatches with the target platforms. Communication between RCX's occurs using infrared signals. These signals are asynchronous and the channel used for this type of communication is unreliable. Initially, communication in SLCO could only occur by exchanging synchronous signals over reliable channels. To enable a straightforward mapping from SLCO to NQC, SLCO was extended with asynchronous communication primitives and lossy channels. A transition with both a trigger and a guard, has no counterpart in Promela that matches the semantics of SLCO. We dealt with this problem by disallowing a transition to have both a trigger and a guard. In practice, this restriction posed no problems since we never used a trigger in combination with a guard in any of our models.

A number of design decisions have been made to increase the *quality of models*, in particular their understandability and modifiability. For example, while creating models by hand, we noticed that it was tedious to initialize variables explicitly as part of a state machine describing the behavior of a class. For this reason, we enabled specifying an initial value for a variable as part of its declaration. We also noticed that some variables were used only locally by state machines, but were defined globally as variables of the class containing the state machines. To improve the readability of the models, we introduced the concept of local variables. However, this increase in readability is accompanied by a decrease in modifiability. We use the Eclipse Modeling Framework to implement our DSL

and transformations. In this framework, a tree view editor is a commonly used tool for editing models. When referring to a variable in this tree view editor, the container of a variable is not shown in the list of variables that can be referred to. This makes it hard to distinguish between two variables with the same name from a different container. A prototype of a graphical editor we developed, suffers from the same problem. The many changes required for increasing the understandability and modifiability of our DSL can partially be explained by our lack of experience in designing DSLs. We found that it is hard to predict what language features will improve understandability and modifiability without actually using the language. These are, however, important quality attributes that should be taken into account to enhance the chance for acceptance of the language in practice. We expect that there are more quality attributes [112] that influence the evolution of a language, which will become apparent when the language is used more extensively.

Enhancing the *quality of the transformations* also had its effect on the language. To increase the understandability and modifiability of the transformations, certain implicit language features have been made explicit. For example, the arguments of all signals sent over a channel must have the same type. A channel is characterized by this type, its directionality, its synchronicity, and its reliability. Initially, the allowed types of the arguments and the directionality of a channel were left implicit. However, the types of the arguments of the signals sent over a channel must be specified explicitly in Promela. When transforming SLCO to Promela, the characteristics of a channel had to be derived from the statements that use the channel for sending and receiving. To avoid this, we decided to make the type of channels explicit, which lead to smaller transformations.

The evolution of the DSL has lead to changes in the abstract syntax of the language, i.e., in the metamodel. Since the transformations described in Section 8.5 are based on this metamodel, they have to evolve along. In Section 7.4.2, we described how the metamodel coverage relation visualization has helped in this co-evolution process.

8.4 Target Platforms

To enable verification and execution of SLCO models, a verification platform and an execution platform are required. In this section, we briefly introduce the platforms and languages we chose for verification and for execution. We also describe how these platforms differ from SLCO.

8.4.1 Execution — NQC

We use the Lego Mindstorms platform for the execution of SLCO models. The key part of this platform is a programmable Lego brick, called RCX. This RCX has an

infrared port for communication and is connected by wires to sensors and motors for interaction with its environment. We deliberately opted for the outdated RCX brick instead of the newer and more advanced NXT brick to investigate the strength of our transformational approach when dealing with a very primitive execution platform. The language we use to program these programmable bricks is called Not Quite C (NQC) [189]. NQC is a restricted version of C, combined with an API that provides access to the various features of the Lego Mindstorms platform such as sensors, outputs, timers, and the infrared port.

8.4.2 Verification — Promela

Model checking is an automated verification technique that checks whether a formally specified property holds for a model of a system [128]. An exhaustive state space search is performed by an automated model checker to determine whether a property holds in a finite state model of a system. Model checking is commonly used for verifying concurrent systems. Before automated model checking can be applied, two tasks need to be performed. First, the system to be verified needs to be specified in a formalism that is accepted by an automated model checker. Second, the properties that the system should satisfy have to be formulated in a logical formalism that is accepted by an automated model checker. Usually a form of temporal logic is used for this. Hereafter, an automated model checker can be used to verify whether the property holds in the model. If this is not the case, a counterexample in the form of a trace violating the property is often provided.

We use the model checker Spin [190]. Spin can check a model for deadlocks, unreachable code, or determine whether it satisfies an LTL property. The modeling formalism for Spin is Promela. The Promela language has constructs for modeling selections and loops. These are based on Dijkstra's guarded commands. Promela has primitives for message passing between processes over channels either using queues, or handshaking. This enables the modeling of both asynchronous and synchronous communication respectively. The syntax of expressions and assignments in the Promela language is inherited from C.

8.4.3 Platform Characteristics

The discussed platforms and languages have different characteristics. These differences are shown in Table 8.1.

The first column lists the languages. The second column indicates whether communication is synchronous or asynchronous on the corresponding platform. If communication is synchronous, both sender and receiver need to be available before a signal can be sent. In this way, sender and receiver synchronize on communication. If communication is asynchronous, a sender can send a signal

Platform	(A)synchronous communication	Lossless/lossy communication	# Concurrent objects	Datatypes	Connectivity for communication
SLCO	both	both	∞	Boolean, Integer, String	Point-to-point
NQC	asynchronous	lossy	limited	Integer	Broadcast
Promela	both	lossless	∞	Integer, Channel, mtype, User-defined types	Point-to-point

Table 8.1 Platform characteristics

and proceed with its execution even though the receiver is not yet ready to receive the signal. The third column indicates whether channels are lossless or lossy. If a channel is lossless, a signal that is sent will always arrive at the receiving end. If a channel is lossy, a signal that is sent may get lost. The fourth column lists the amount of objects that can be instantiated simultaneously. In SLCO, this amount is unlimited. For Lego Mindstorms, however, this number is limited in practice. Because every object should be deployed on an RCX, the amount of concurrent objects is bounded by the available number of RCX bricks. The fifth column shows the datatypes that are available on the corresponding platforms. The sixth column shows whether signals are broadcasted or sent using point-to-point communication. If signals are broadcasted, each signal can be received by multiple objects. In the case of point-to-point communication signals are sent from one object to exactly one other object.

8.5 Model Transformations

In Section 8.4.3, we explained that the characteristics of the platforms differ. To execute SLCO models, these semantic platform gaps need to be bridged. Therefore, we defined a number of transformations that transform an SLCO model to a refined SLCO model with equivalent observable behavior. Each of these transformations eliminates one of the platform gaps. An SLCO model that uses synchronous communication only, for example, can be transformed to an equivalent SLCO model that uses asynchronous communication only.

First, we describe two coarse-grained transformations that bridge platform gaps, as well as a transformation that ensures that the precondition of one of those other transformations is met. Afterwards, we describe the more fine-grained versions of those transformations. Finally, we describe the model transformations that transform SLCO models to Promela models and NQC models.

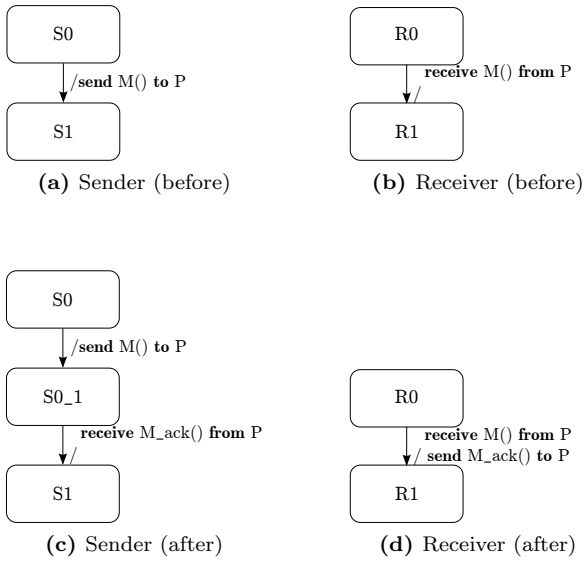
8.5.1 Coarse-grained Transformations

The coarse-grained transformations deal with only two out of the five platform gaps. Model developers are responsible for creating input models that do not introduce problems concerning the other three platform gaps. There is no transformation that can be used to reduce the number of objects, so the model developer is responsible for creating input models that contain as many objects as can be deployed. Note that the coarse-grained transformations do not introduce objects themselves. The coarse-grained transformations also do not introduce datatypes that can not be used in NQC. If the input model does not use these datatypes, the transformations will result in a deployable model. Because there is no transformation that deals with the problem of identifying the sender of a message that has been broadcasted, only input models with two communicating parties are allowed.

8.5.1.1 Synchronized Communication over Asynchronous Channels

The transformation that replaces communication using synchronous signals by communication using asynchronous signals ensures that the behavior of the model is still as desired by adding acknowledgment signals for synchronization. Whenever a signal is sent, the receiving party sends an acknowledgement indicating that the signal has been received. The sending party waits until the acknowledgement has been received. In this way, synchronization is achieved. The effect of the transformation on a part of an SLCO model is depicted in Figure 8.5.

This solution to achieve synchronized communication over asynchronous communication channels is not generally applicable. Suppose that state $S0$ of the state machine depicted in Figure 8.5a and state $R0$ of the state machine depicted in Figure 8.5b both have an additional outgoing transition to another state without a trigger, guard, or effect. In this case, there is a non-deterministic choice in both state machines between synchronous communication with the other state machine and taking this additional transition. Since communication is synchronous, both parties need to be available for communication to occur. Therefore, either both state machines will participate in the communication or both state machines will take the additional transition. In the asynchronous variant (figures 8.5c and 8.5d), it is not required that both parties are available to start communication. Therefore, the sender (Figure 8.5c) can start communication by taking the transition from state $S0$ to state $S0\_1$, while the receiver takes the additional transition. Now, the sender is in state $S0\_1$ and, since the receiver is no longer in state $R0$, it will never receive the acknowledgement signal from the receiver. Therefore a deadlock occurs that was not present in the original model. This example shows that the proposed transformation does not lead to a behavioral equivalent model when non-deterministic choice is involved, i.e., it is not generally applicable.



(a) Sender (before) **(b)** Receiver (before)

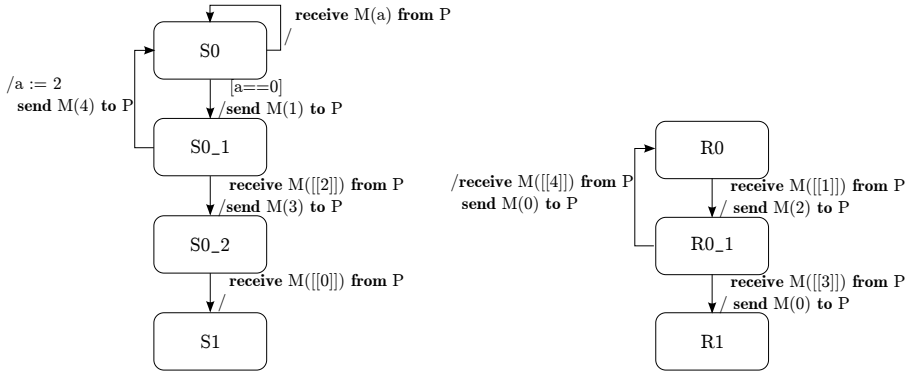
(c) Sender (after) **(d)** Receiver (after)

Figure 8.5 Synchronized communication over asynchronous channels

To overcome this issue, a different solution is required. The solution we chose is based on the four-phase handshake protocol described by Feijen and van Gasteren [193]. Their version of the protocol uses shared variables. We cannot use shared variables, since communicating state machines are not part of the same object. Therefore, we adapted the protocol in such a way that it works with signals that are being exchanged between state machines. The protocol allows to cancel the sending of a message, which enables making another (non-deterministic) choice. Note that the protocol allows indefinite cancelation of messages, which may result in a livelock. The state machines depicted in figures 8.5a and 8.5b are transformed to the state machines depicted in figures 8.6a and 8.6b. Note that most of the arguments of the signal reception triggers are between double square brackets ([[]]). This indicates conditional signal reception. Note also that a (local) variable a is used in this solution at the sender side. The initial value of this variable is 0.

8.5.1.2 Lossless Communication over Lossy Channels

Lossless communication over lossy channels is implemented using a variant of the alternating bit protocol (ABP) [194]. This protocol ensures that each signal that is sent, is eventually received, assuming that not all signals get lost. We



(a) Sender (after) **(b)** Receiver (after)

Figure 8.6 Synchronized communication over asynchronous channels with handshake

implemented a model transformation that adds the ABP to a model by adding new state machines implementing the protocol to objects that communicate over a lossy channel. These new state machines communicate with the existing state machines in these objects using shared variables.

8.5.1.3 Exclusive Access to Ports

To ensure that a model meets the precondition for the transformation that adds the ABP to a model, we use a third transformation. When multiple state machines communicate over the same port, the previous transformation may only be applied if at most one of the state machines sends a message over this port at the same time. This requires exclusive access to ports. To ensure this, we implemented a model transformation that adds a token server to each object were exclusive acces to ports is required. This token server is implemented as an additional state machine that is added to the objects directly. The token server and the existing state machines pass information to each other using shared variables.

8.5.2 Fine-grained Transformations

Our experiments, which are presented in Section 8.6, showed that the state spaces belonging to the most concrete models acquired from the coarse-grained model transformations are too large for model checking. The reason for this is that the change induced on the models by the coarse-grained model transformations is too large. Therefore, we implemented a number of more fine-grained transformations to minimize the influence of each transformation on the size of the state space. In Sections 4.5 and 5.5, we observed that smaller model transformations

are considered to be more understandable and modifiable. Since fine-grained transformations affect a smaller part of the model, it is to be expected they are smaller than course-grained ones Therefore, they are expected to be of higher quality in terms of understandability and modifiability.

Recall that the coarse-grained transformations bridge only two of the platform gaps discussed in Section 8.4.3. Model developers are responsible for creating input models that do not introduce problems concerning the other platform gaps. To provide model developers with more freedom in modeling, we implemented fine-grained model transformations for bridging each of the platform gaps. The combination of splitting coarse-grained transformations into smaller transformations and introducing new transformations, make that it is impossible to compare the coarse-grained transformations and the fine-grained transformations in a straightforward way. There is, however, one exception to this. The transformation that replaces synchronous signals by asynchronous signals is left unchanged, since we found no way to make it more fine-grained. The other fine-grained transformations are described below. Which of the transformations need to be applied and also the order depends on the characteristics of the input model and also on the available number of RCX's.

8.5.2.1 Lossless Communication over a Lossy Channel

Lossless communication over lossy channels is again implemented using the ABP. In this case, the ABP that is added by the transformation is implemented as a number of concurrent objects that are connected to the communicating objects using lossless channels. In contrast, the coarse-grained version of this transformation adds the ABP as a number of state machines to the communicating objects.

8.5.2.2 Reducing the Number of Objects

The transformation that reduces the number of objects merges objects by creating a new object that contains all the variables, ports and state machines contained in the original objects. If two state machines that were originally contained in two different objects communicate over a lossless, unidirectional, synchronous channel, this form of communication is replaced by communication using shared variables. Also here, the four-phase handshake protocol from Feijen and van Gasteren is used to ensure synchronized communication between the two state machines [193]. This transformation is used to make the number of objects match the number of available RCX's, since an object is deployed on an RCX. Typically, this transformation is used to merge the objects that represent the ABP for an object with that object.

8.5.2.3 Replacing Strings by Integers

On the NQC platform, strings are not available as datatype. Therefore, we implemented a transformation that replaces all string constants in a model by integer constants.

8.5.2.4 Making the Sender of a Signal Explicit

When multiple objects broadcast signals with the same name and the same number of arguments over the same medium, the receiving object cannot determine the origin of such a signal. This situation arises when multiple RCX controllers communicate with each other, because they communicate using infrared. To enable a receiving controller to determine the origin of each signal it receives, a number identifying the sending object is appended to the names of each signal.

8.5.2.5 Making all Signal Names Equal

To keep the transformation that adds the ABP as simple as possible, our implementation of the ABP takes signals with a predefined name as input. Before an instance of the ABP can be added to substitute an asynchronous, lossless, unidirectional channel, the names of the signals that are sent over this channel have to be changed to this predefined name. We implemented a transformation that performs this change and ensures that the original names of the signals are passed as a parameter of the signals that replace them. For instance, a signal $S()$, will be transformed to a signal $Signal(S)$.

8.5.2.6 Replacing a Bidirectional Channel by Two Unidirectional Channels

Our implementation of the ABP can only substitute asynchronous, lossless, unidirectional channels. When the ABP needs to be added to a bidirectional channel, the channel needs to be transformed to two unidirectional channels. Therefore, we implemented a transformation that replaces a bidirectional channel between two objects O_1 and O_2 by two unidirectional ones and ensures that communication from object O_1 to object O_2 occurs over one channel and that communication from object O_2 to object O_2 occurs over the other channel.

8.5.2.7 Duplicating a Channel for Each State Machine that Uses It

The four-phase handshake protocol we employ when merging objects does not work properly if multiple state machine send information over the same port at the same time. When two objects are connected by a unidirectional channel and multiple state machines within one of the objects send signals over this channel, the channel must be replaced by multiple channels before these objects can be merged. We implemented a transformation that introduces a new channel with

the same properties as the original channel for each state machine that sends signals over this channel.

8.5.2.8 Reducing the Number of Channels

When two objects are connected by more than one channel, these channels can be merged into one channel. Therefore, we implemented a transformation that changes the names of the signals that are sent over the new channel, to distinguish between identical signals that were previously sent over different channels. Merging channels can be used to optimize a model, because it is a way to reduce the number of instances of the ABP that need to be added.

8.5.2.9 Adding Delays to Transitions

To prevent objects from sending signals continuously, we implemented a transformation that adds delay triggers to the transitions that send signals as part of their effect. This transformation also optimizes a model, because it reduces the number of messages that are being sent, which in turn reduces the number of collisions between messages sent via infrared.

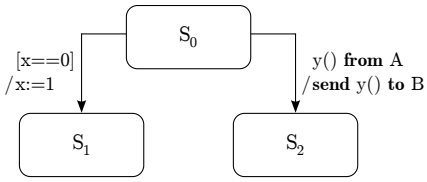
8.5.3 Cross-platform Transformations

In this section, we discuss the model transformations from SLCO to the formalisms for verification, and execution.

8.5.3.1 SLCO to Promela

The transformation from SLCO to Promela transforms SLCO models containing synchronous or asynchronous, lossless channels to Promela models. Every state machine contained in a class instantiated as an object in an SLCO model is transformed to a Promela proctype. Channels between objects are transformed to channels between proctypes. State machines can be implemented using an imperative programming style in multiple ways [195]. We have chosen to implement them as jump tables using goto-statements. An example of a source model and corresponding target model, illustrating part of the transformation is shown in Figure 8.7. State S_0 depicted in Figure 8.7a is transformed to the code depicted in Figure 8.7b.

A state is transformed to a labeled selection statement. The name of the state is used for the label. Every outgoing transition of state S_0 is transformed to an alternative of the selection statement. The semantics of the selection statement is such that it will non-deterministically execute one of the alternatives for which the guard is executable and it will block if none of the guards is executable. The guard is the first statement or expression of an alternative. The guard and statements



(a) Partial state machine

```
S0:
 if
 ::x == 0 -> x = 1;
   goto S1;
 ::A?y -> B!y
   goto S2;
 fi;
```

(b) Corresponding Promela code

Figure 8.7 Transforming SLCO to Promela

of a transition are transformed to Promela code in a straightforward way. When the guard is executable, the guard itself and the statements following it can be executed. In case the guard is an expression, it is executable if it evaluates to `true`. In case the guard is a statement, it is executable if the statement is. When the guard and the statements are executed, the transition to a state has been completed and the state machine is *in* the target state of the transition. Therefore, a goto statement that jumps to the label representing the target state of a transition is added after the transformed statements in the code. Because we use an untimed version of Spin, we abstract from time by transforming deadline triggers to skip statements. A signal reception trigger is transformed to a receive statement. A receive statement blocks until it is able to receive a message over a channel. Since a statement can be used as a guard in the selection statement, a signal reception trigger can be transformed to a guard. If a transition has no guard nor a trigger, only the statements that constitute the effect are transformed.

Specification of Temporal Logic Properties

In Section 8.4.2, it was noted that model checking can be used to verify whether a property, in this case an LTL property, holds in a Promela model. An LTL property typically refers to elements in a Promela model. This can pose complications in our case, since we make use of Promela models generated from SLCO models. This makes specifying an LTL property a hard and error-prone task.

To ease the process of specifying an LTL formula, we developed a metamodel for modeling LTL properties. This metamodel is depicted in Figure 8.8. The metaclass `AtomicProposition` has a reference to the variable metaclass of the SLCO metamodel. In this way, it is possible to model LTL properties using, besides the temporal logic constructs, also the elements of the SLCO model. We also implemented a model-to-text transformation to generate an LTL property in textual form that can be used by the model checker Spin.

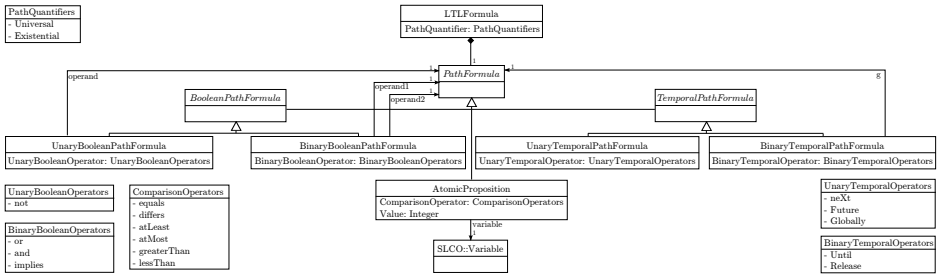


Figure 8.8 Temporal logic metamodel

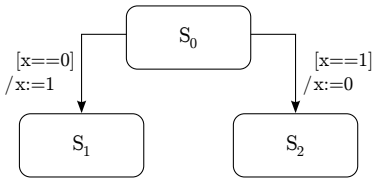
8.5.3.2 SLCO to NQC

The transformation from SLCO to NQC transforms SLCO models with asynchronous, lossy channels to NQC models. To enable execution of the NQC code, an additional requirement should be fulfilled, viz., the number of concurrent objects in the SLCO model should match the number of available RCX'.

Every state machine contained in a class instantiated as an object in an SLCO model is transformed into an NQC task. The transformation from state machines to NQC code resembles the transformation from state machines to Promela described in Section 8.5.3.1. However, there is an important difference. The semantics of the if-statement in NQC is different from the one in Promela. First, the if-statement in NQC is a conditional statement rather than a selection construct. This means that only one alternative can be specified per if-statement. Therefore, every outgoing transition of a state is transformed into a separate if-statement. Second, the if-statement in NQC is non-blocking. This means that the condition of an if-statement is evaluated only once and if it does not evaluate to `true`, the statement is treated as a skip-statement. To ensure that the guards are continuously being evaluated and that triggers are picked up, a goto-statement is added after the if-statement that jumps back to the if-statement. Last, the if-statement in NQC requires an expression as condition and not a statement as is the case with Promela. For transitions with guards this is no problem, since these are expressions in SLCO as well. When a transition has no guard or trigger, the transition is always enabled. Therefore, the expression `true` is used in this case. Triggers should also be transformed into expressions. A deadline is transformed into an expression that checks whether a timer has exceeded the deadline value. This timer is started when the transition is taken to the state from which an outgoing transition has a deadline trigger. A signal reception is transformed into an expression that checks whether the message that is expected is ready to be received. Note that this is possible since an RCX stores the last message that has been received.

An RCX can only send and receive integer values over its infrared port. Since signals in SLCO consist of the signal itself and possibly a number of arguments, they have to be encoded such that they can be represented as a single integer. This means that before a signal is sent, it has to be encoded and that after a signal has been received it has to be decoded. The actual sending and receiving of messages is done via API calls.

An example of a source model and corresponding target model, illustrating part of the transformation from SLCO to NQC is depicted in Figure 8.9.



(a) Partial state machine

```
S0:
 if(x == 0){x = 1;
            goto S1;
           }
 if(x == 1){x = 0;
            goto S2;
           }
 goto S0;
```

(b) Corresponding NQC code

Figure 8.9 Transforming SLCO to NQC

8.6 Experiments

We performed a number of experiments to determine the size of the state space of intermediate models generated by our chains of refining transformations. By transforming intermediate SLCO models to Promela models, we obtain models of which the state space can be explored using the model checker Spin. For the experiments described in this section, we configured Spin to explore the state spaces by means of a depth-first search with a maximum search depth of $1 \cdot 10^8$ transitions and using at most $4 \cdot 10^4$ megabytes of memory. After describing the models that serve as inputs for our experiment, we show that an approach using coarse-grained transformations quickly leads to models with very large state spaces. Then, we present the results of our experiments using fine-grained transformations. Finally, we discuss how applying transformations to a part of the applicable model elements only can also be used to explore the state space of less abstract versions of models.

8.6.1 Cases

We apply the refining transformations described in Section 8.5 to three different models. The first model is a producer-consumer model. It consists of one producer

object that repeatedly sends synchronous signals over a port and one consumer object that receives these signals over a port. A channel connects the port of the producer to the port of the consumer.

The second model describes the behavior of a system consisting of three interoperating conveyor belts. The state machine depicted on the left of Figure 8.4 (page 138) shows the behavior of one of the three conveyor belts in this system. The two other conveyor belts have the same behavior. Therefore, only one class and state machine have to be defined for this. Note, however, that it needs to be instantiated twice, once for each of the conveyor belts. The state machine that models this behavior is depicted on the right of Figure 8.4. Since we had only two RCX's available at the time and there is no coarse-grained transformation that merges objects, we used a slightly different model for our experiments. Instead of instantiating the class twice, we changed the class such that it contains the same state machine twice. In this way, the class models the behavior of two conveyor belts, which makes that it has to be instantiated only once. Therefore, it can run on one RCX, leaving the other one available for the object controlling the third conveyor belt. In the complete model, there is a third class that models the environment of the system. This class, as well as the state machines contained in it, are not described here.

The third model is related to the first model. It consists of two producer objects and one consumer object. Two channels connect the ports of each of the producers to the two ports of the consumer. The model is depicted in Figure 8.10.

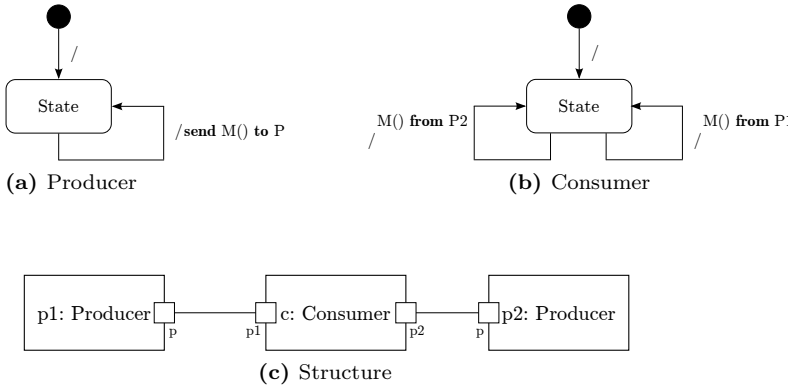


Figure 8.10 Two producers and one consumer

8.6.2 Results

Application of the coarse-grained transformations to the first model, i.e., the one with one producer and one consumer, leads to the state-space sizes shown in Table 8.2. Replacing synchronous communication by asynchronous communication approximately doubles the size of the state space. Note that for all three models we used in our experiments it suffices to apply the (simple) transformation that adds acknowledgement signals only, i.e., the four-phase handshake protocol is not required. Adding the state machines that implement the ABP to both the producer object and the consumer object leads to a significant increase of the size of the state space. Although the resulting state space of the most concrete model is much larger than the one of the intermediate model, it is still small enough for verification given the aforementioned configuration of Spin.

Model	# States	# Transitions
Original	4	6
Asynchronous signals	8	11
Lossless communication	76 066 432	542 196 960

Table 8.2 One producer and one consumer – coarse-grained transformations

To further illustrate the effects of the coarse-grained transformations on the size of the state space, we applied them to the second model, i.e., the interoperating conveyor belts model, as well. This model is slightly more complex than the first model. Recall that one of the classes of the interoperating conveyor belts model contains two state machines. These two state machines communicate over the same port. Therefore, the model transformation that adds a token server to the model has to be applied before the transformation that adds the ABP can be applied. Table 8.3 shows that adding a token server leads to a state space that can still be model checked. The final row in the table indicates that for the most concrete model it is impossible to explore the entire state space before the search depth is exceeded or all available memory is depleted. This shows that the output of this transformation is not suited for model checking, even though the input model is still relatively small.

Among others, the results of these experiments lead us to implement the more fine-grained versions of the transformations presented in Section 8.5.2. Table 8.4 shows the effect of the fine-grained transformations on the size of the state spaces of the intermediate models for the first model. Table 8.5 shows the effect on the state space of the second model. The transformations that ensure that all signals have a fixed name, replace bidirectional channels by two unidirectional channels, ensure that each state machine within an object communicates with the ABP over an exclusive channel, and replace strings by integers have no effect on the size of

Model	# States	# Transitions
Original	494	1 294
Asynchronous signals	748	1 980
Token server	10 090	33 820
Lossless communication	–	–

Table 8.3 Interoperating conveyor belts – coarse-grained transformations

the state space. Note that the effect of adding the ABP on the state-space size is much smaller when applying the fine-grained transformation than when applying the coarse-grained transformation. The reason for this is that the fine-grained version adds additional objects that communicate over synchronous channels, whereas the coarse-grained one adds additional state machines that communicate using shared variables. In general, communication over synchronous channels does not increase the state space, whereas communication using shared variables does. However, later on the objects that represent the ABP are merged with the objects that need the ABP. Then communication is no longer performed using synchronous channels, but using shared variables, which leads to an increase in the size of the state space.

Model	# States	# Transitions
Original	4	6
Asynchronous signals	8	11
Fixed signal names	8	11
Unidirectional channels	8	11
Lossless communication	114 388	596 367
Delays	1 009 856	5 902 673
Merged objects	83 251 840	592 242 910
Integers instead of strings	83 251 840	592 242 910

Table 8.4 One producer and one consumer – fine-grained transformations

For the first model, each intermediate model has a state space that can be explored given the aforementioned configuration of Spin. For the second model, merging objects leads to a state-space that is too large to explore. Even though the most concrete model is still unsuited for model checking, fine-grained transformations enable exploring an intermediate model that is more concrete than the one produced by coarse-grained transformations.

Model	# States	# Transitions
Original	494	1 294
Asynchronous signals	748	1 980
Fixed signal names	748	1 980
Unidirectional channels	748	1 980
Lossless communication	19 148 872	141 049 260
Delays	167 466 690	1 334 614 400
Exclusive channels	167 466 690	1 334 614 400
Merged objects	–	–

Table 8.5 Interoperating conveyor belts – fine-grained transformations

8.6.3 Exploring the Boundaries

Both model one and model two consist of two objects that eventually have to communicate over an unreliable channel. Therefore, in both experiments, only two instances of the ABP are added, viz., one for communication in each direction. Table 8.6 shows the effect of the fine-grained transformations on the size of the state spaces of the intermediate models for the third model, i.e., the model consisting of two producers and one consumer. To achieve lossless communication over a lossy channel in this case, four instances of the ABP have to be added, because communication takes place in both directions between two pairs of objects.

Model	# States	# Transitions
Original	8	17
Asynchronous signals	33	68
Fixed signal names	33	68
Unidirectional channels	33	68
Lossless communication	–	–

Table 8.6 Two producers and one consumer – fine-grained transformations

Adding four instances of the objects that implement the ABP leads to an explosion of the state space. This makes it infeasible to verify properties of this model using state space exploration given the aforementioned configuration of Spin. Table 8.7 shows the effect of adding an instance of the ABP to respectively one, two, and three channels in the third model, while leaving the other channels untouched. By replacing communication over only a subset of the four channels in the model by communication via the ABP, a model is obtained with a state

space that is significantly smaller than the state space corresponding to the model in which communication over all channels is replaced. In this way, verification of a model that resembles the implementation more closely than the original, more abstract, model is possible. The same approach can be used to merge only some of the objects in the model of the interoperating conveyor belts. Generally, in cases where it is impossible to verify the completely refined model, applying a refining transformation to a part of the applicable elements in the model only can be used to verify intermediate models that resemble the implementation as close as possible, in cases where it is impossible to verify the completely refined model.

Model	# States	# Transitions
Original	8	17
Asynchronous signals	33	68
Fixed signal names	33	68
Unidirectional channels	33	68
one ABP instance	5 188	21 335
two ABP instances	527 108	3 224 435
three ABP instances	105 715 260	879 085 750

Table 8.7 Two producers and one consumer – fine-grained transformations

8.6.4 Discussion

For our experiments, we used the model checker Spin to illustrate the effect of both coarse-grained and fine-grained transformations on state spaces. However, our approach is not limited to one particular model checker. The refining transformations we implemented take SLCO models as input and produce SLCO models as output. Support for another model checker (or a similar tool) can be added by implementing a single transformation from SLCO to the formalism supported by that tool.

To clearly show the influence of our refining transformations, we used no additional reduction or abstraction techniques. However, our approach can be combined with such techniques in practical situations. Using one of the standard state vector compression modes offered by Spin [196], for instance, it is possible to explore larger state spaces. Using this compression method, the state space of the timed version of the model of the three conveyor belts can be explored using approximately $15 \cdot 10^3$ megabytes, instead of $31 \cdot 10^3$ megabytes.

Typically, model checking is used to verify whether a property holds for a model of a system. Because the refining transformations modify the model, properties under investigation may have to change as well. After adding communication

via the ABP to a model, for example, there are unfair traces in the state space representing the behavior that all signals are discarded by the lossy channel. To consider only the fair traces, a fairness constraint has to be added to the property.

8.7 Related Work

Multiple proposals are presented in literature to enable model checking of huge specifications. Clarke et al. suggest four different abstraction techniques and demonstrate their practicality on a number of examples [186]. Another possibility, applied by Chan et al., is to model check only a part of the system [185]. They also applied simplifications to the model to avoid constructs that could not be handled properly by their model checker. Wing and Vaziri-Farahani enabled quick verification in a case study by applying abstractions to both the model and the verification properties [187]. They state that the choice of what abstractions to apply takes some 'good' judgement. All of the aforementioned approaches work by applying abstraction and simplification to concrete models. Our approach works the other way around, we refine an abstract model to a more refined one. Note that our approach does not preclude the use of abstractions and simplifications on the (intermediate) models.

An approach related to the one described in this chapter is described by Weißenbacher and Herzner [197]. In their approach, an abstraction algorithm is applied to a model. Next, verification techniques such as model checking are applied on the acquired abstract model to verify whether a specified property holds. If this property holds, it can be concluded that the property holds in the original model as well. In this case, the process stops. If this property does not hold, the next step is to determine whether the behavior specified by this property is absent in the original model a well. If this is the case, then it can be concluded that the property indeed does not hold in the original model. This means that a valid counterexample has been found and the process stops. If this is not the case, i.e., if the behavior specified by the property is absent in the abstract model, but present in the original model, an invalid counterexample has been found. This may result from the coarseness of the abstraction. In this case, the abstract model should repeatedly be refined until the invalid counterexample is eliminated. The invalid counterexample is eliminated if the counterexample is found in the original model as well or if it does not occur in the refined abstract model anymore. Their approach is related to the application of partial refinement to a model.

The B-method [198] is developed as a means to refine abstract specifications into implementations. By fulfilling a number of proof obligations and thus proving that each refinement step is sound, it can be proven that an implementation adheres to the corresponding initial specification. Using the B-method, reliable code is derived starting from one initial specification, whereas our approach

focusses on automatically generating reliable code from every possible model that can be described using our DSL.

8.8 Conclusions

In this chapter, we proposed an approach using model checking to increase the reliability of code generated from models specified in a domain-specific language. A model transformation from a domain-specific language to a language suitable for model checking can be defined to enable model checking of domain-specific models. Using this model transformation, model checking can be applied on the domain-specific models in every stage of the refinement process. To deal with the state-explosion problem we advocate to use fine-grained model transformations to stepwise refine domain-specific models. Fine-grained transformations tend to be smaller than coarse-grained transformations. Therefore, their quality in terms of understandability and modifiability is higher. Another advantage of fine-grained transformations is that they may be more reusable than coarse-grained ones. When fine-grained transformations still do not allow model checking of models that are sufficiently concrete, it may be helpful to apply a transformation to part of a model only.

We conducted experiments to validate our approach on multiple cases with a DSL we defined. The results show that it is possible to validate models that are more concrete when fine-grained transformations are applied.

Throughout the development process, the DSL has evolved. We identified four main influences on the evolution of our DSL, viz., the problem domain, the target platforms, model quality, and model transformation quality. The problem domain, model quality, and transformation quality continuously influence the evolution of a language throughout the development process. The problem domain should always be taken into consideration when adapting the language to ensure that the abstractions provided by the language fit the domain. Opportunities to adapt the language to improve the quality of models and model transformations become apparent as experience with the language grows. Because quality is a subjective concept, quality attributes can be in conflict. For example, we added local variables to state machines to increase understandability, which had a negative effect on modifiability.

Chapter 9

Conclusions

In this thesis, we studied how the quality of model transformations can be assessed and improved. This chapter summarizes the contributions of this thesis and provides directions for further research.

9.1 Contributions

Model-driven engineering gets an increasing amount of attention from industry and is therefore becoming more prominent. Since model transformations play a pivotal role in this paradigm, they are becoming more and more important as well. Model transformations are in many ways similar to traditional software artifacts. To prevent them from becoming the next maintenance nightmare, they need to comply to similar quality standards. The central research question of this thesis is therefore as follows.

> **RQ:** *How can the quality of model transformations be assessed and improved, in particular with respect to development and maintenance?*

To answer this research question, we first considered a model transformation that transforms models from one semantic domain to models in another semantic domain. Model transformations that need to bridge a semantic gap are expected to be more complex than those that merely transform syntax. The complexity of a model transformation has repercussions on its quality. Quality, in general, is

a broad and subjective concept. Therefore, we defined quality in the context of model transformations and indicated what quality attributes we consider relevant with respect to their development and maintenance. We also described different approaches for assessing this quality. Thereafter, we focused on a widely-applied approach for assessing the quality of all kinds of software artifacts, viz., metrics. We defined metrics sets for measuring model transformations developed with the model transformation languages ASF+SDF, ATL, QVTO, and Xtend. For two of these metric sets, viz., the ones for ASF+SDF and for ATL, we conducted an empirical study aimed at establishing a relation between the metrics and aforementioned quality attributes. The identified relations can be used as basal directives for improving the quality of model transformations. Another approach for improving the quality of model transformations is to support their development and maintenance process with visualization techniques. Visualization techniques have proved their use for traditional software artifacts. However, hardly any are currently available for model transformations. Therefore, we developed four visualization techniques for model transformations. The development and validation of the metric sets and visualization techniques has led to insights regarding the development of model transformations. We applied these insights, as well as the techniques and tools we developed during the development of a number of model transformations that bridge a number of semantic gaps.

In the remainder of this section, we revisit the research questions and address the main contributions of this thesis.

9.1.1 Model Transformation between Semantic Domains

Model transformations facilitate reuse of models that have been developed earlier in a software development process. In Chapter 2, we described a model transformation that transforms process algebra models to state machine models. The goal of this transformation is to reuse analysis models that have been developed during a software development process for code generation, which state machines are well suited for. However, the semantic properties of process algebra and state machines are so far apart that straightforward transformation is impossible. This led to the following research question.

> **RQ$_1$**: *How should model transformations that transform between models from different semantic domains be approached?*

In general, to bridge a semantic gap when transforming models from one formalism into another, it first has to be identified. Therefore, two steps should be taken. First, the semantics of the source and the target formalism should be well understood. Second, additional requirements on the (static) semantics should be made explicit. When bridging a semantic gap is not straightforward, it is recommended to address a simplified version of the source metamodel first. Therefore,

we started with plain process algebra, i.e., without time, data, stochastic, etc., to acquire understanding of code generation from, and model transformations based on process algebras. Another possibility is to relax the semantic requirements on the transformation. In our case, a semantic gap emerged as a result of the requirements on the transformation, viz., the transformation should preserve both structural and behavioral properties. We relaxed the former requirement. Our transformation preserves structure for all operators except for two operators that are seldomly used in modeling. Note that it can also occur that the semantic domains of the source and target language are so far apart that the semantic gap cannot be bridged at all.

9.1.2 Quality of Model Transformations

Recall that model transformations need to adhere to similar quality standards as traditional software to prevent them from becoming the next maintenance nightmare. Since quality in general is a broad and subjective concept, it needs to be defined in the context of model transformation. This led to the following research question that was addressed in Chapter 3.

> **RQ$_2$**: *How can quality be defined in the context of model transformations?*

A model transformation can be considered as a transformation definition or as the process of transforming a source model to a target model. Accordingly, model transformation quality can be defined in two different ways. The quality of the definition is referred to as the internal quality of a model transformation. The quality of the process of transforming a source model to a target model is referred to as the external quality of a model transformation. For both definitions, different attributes of quality play a role. We discussed seven quality attributes and their relevance for model transformations, in particular with respect to development and maintenance. Six of these quality attributes mostly relate to internal quality, viz., understandability, modifiability, reusability, modularity, consistency, and conciseness. One mostly relates to external quality, viz., completeness.

Since a model transformation can be considered in two different ways, there are also two ways to assess its quality (both internal and external). Quality can be assessed by performing measurements on the transformation definition. We refer to this type of quality assessment as direct quality assessment. Direct quality assessment is the preferred method for assessing the internal quality of model transformations. Although, some external quality attributes can be assessed using direct quality assessment as well. Quality can also be assessed by performing measurements in the environment of the transformation, e.g., by comparing properties of source and target models, or by measuring execution time. We refer to this type of quality assessment as indirect quality assessment. Indirect

quality assessment is the preferred method for assessing the external quality of model transformations. It is not suited for assessing attributes of internal quality.

9.1.3 Metrics for Model Transformations

Many methods exist for assessing the quality of software artifacts, e.g., testing, auditing, or collecting metrics. Software metrics have been proposed for measuring various kinds of software artifacts. This seems a viable approach for direct quality assessment of model transformations as well. However, hardly any research has been performed in this area. This led to the following research question.

> **RQ$_3$**: *How can metrics be used to assess the quality of model transformations?*

In Chapters 4, 5, and 6, we defined metric sets for measuring model transformations developed with four model transformation formalisms, each with different characteristics, viz., for ASF+SDF, ATL, Xtend, and QVTO. We also developed tools that enable collecting metrics from model transformations automatically. While these metric sets can be used to indicate bad smells in the code of model transformations, they cannot be used for assessing quality yet. A relation has to be established between the metric sets and quality attributes. For two of these metric sets, viz., the ones for ASF+SDF and for ATL, we conducted an empirical study aimed at establishing such a relation. From these empirical studies we learned what metrics serve as predictors for the different quality attributes of model transformations. These insights can be taken into consideration when developing model transformations to attain model transformations of higher quality.

9.1.4 Visualization of Model Transformations

Visualization techniques are often employed to facilitate a maintenance process and also the remainder of a development trajectory. Numerous such techniques exist for traditional software artifacts. This seems a viable approach for model transformations as well. However, currently there are few visualization techniques available tailored towards analyzing model transformations. This led to the following research question.

> **RQ$_4$**: *What visualization techniques can be employed to support the development and maintenance process of model transformations?*

One of the most time consuming processes during software maintenance is acquiring understanding of the software. We expect that this holds for model transformations as well. Therefore, we described in Chapter 7 four complementary visualization techniques for facilitating model transformation comprehension. The first two techniques are aimed at visualizing the dependencies between the

components of a model transformation. The other two techniques are aimed at visualizing the coverage of the source and target metamodels by a model transformation. We implemented a toolset such that both visualization techniques can be applied automatically.

To visualize the dependencies between the components that comprise a model transformation, we proposed two different visualizations. The first one is a visualization of the import graph of a model transformation, i.e., the modules comprising the model transformations and their import relations. The import graph provides a high-level overview of a model transformation that can help understanding the relations between the different modules. The second one is a visualization of both the structure of a model transformation in terms of the division of transformation functions over modules, and the dependency between transformation functions in terms of the way they invoke each other. This visualization provides additional insights in a model transformation, since the relation between the elements it comprises are made explicit. Moreover, it can help in identifying transformation functions that are frequently invoked or never invoked by other transformation functions.

To visualize the coverage of the source and target metamodels by a model transformation, we also proposed two different visualizations. The first one provides a diagrammatic representation of a metamodel, either a source or a target metamodel of a model transformation, in which the metaclasses, attributes, and references that are covered by the model transformation are highlighted. This visualization facilitates analyzing the completeness of a model transformation. Moreover, it can assist in the process of defining test sets. The advantage of this visualization is that the diagrammatic representation is comparable to the representation used by metamodel developers. The disadvantage, however, is the lack of traceability, i.e., the relation between a metamodel element and the transformation function that covers it is not visible. The second visualization provides a solution to this. In this visualization, the relation between metamodel elements and the transformation functions they are covered by is made explicit. This is useful for finding errors during the development of model transformations, and also for identifying parts of a model transformation eligible for reuse by other (related) transformations. This visualization also facilitates the process of co-evolution of metamodel and transformation, since it is easy to identify what transformation functions cover added, removed, or changed metaclasses.

9.1.5 Improving the Quality of Model Transformations

The development of the metric sets, and in particular the empirical studies, have led to insights considering the development of model transformations. The visualization techniques we propose are aimed at facilitating the development

of model transformations. To validate whether these insights and techniques are actually beneficial for developing model transformations, we formulated the following research question.

> **RQ$_5$**: *How can the acquired insights and developed techniques be applied to facilitate the development and maintenance of model transformations?*

In Chapter 8, we described the development of a chain of model transformations that bridges a number of semantic gaps. We chose to solve this transformational problem not with one model transformation, but with a number of smaller model transformations. This should lead to smaller transformations, which are more understandable. However, the model transformations proved to be insufficiently fine grained to enable verification of intermediate models produced by them. Therefore, we split the model transformations into even smaller parts.

The language on which the model transformations are defined, was subject to evolution. In particular the coverage visualization proved to be beneficial for the co-evolution of the model transformations.

9.2 Directions for Further Research

In this thesis, we addressed tools and techniques for assessing and improving the quality of model transformations, in particular with respect to development and maintenance. This section presents direction for further research in this area.

The insights we acquired from developing the metric sets and, in particular, the empirical studies we conducted have led to basal directives that we used for developing the transformations described in Chapter 8. These basal directives should be extended and converted to guidelines for developing model transformations. Applying these guidelines should lead to model transformations of higher quality. However, research should be performed to confirm whether this is in fact the case. Such research can also lead to further insights into the relation between metrics and quality attributes and also to new guidelines.

Both the metrics and the visualization techniques described in Chapter 7 are static analysis techniques, i.e., they do not require execution of the model transformation under study. Static analysis is a frequently applied technique in software development in general. Therefore, there are a lot more analysis techniques available that could be beneficial for model transformations as well. One such technique that could be considered is code clone detection. A tool like CCFinder [199] may be used for this. Code clone detection can be employed in two different ways. First, a number of different model transformations can be compared to identify reused code. Second, code clones within one model transformation can be identified. Both approaches can be used to indicate

improvements for modularity and reusability of a model transformation. Apart from static analysis, there are also dynamic analysis techniques that can be applied to model transformations. Dynamic analysis of model transformations can lead to additional insights. We already addressed this in Section 7.2.1.3, where we presented ideas on extending the structure and dependency analysis techniques with runtime data. Also the coverage visualization technique can be extended. Currently, the relation between source metamodel element and transformation function, as well as the relation between transformation function and target metamodel element are visualized. In this way, the relation between source metamodel elements and target metamodel elements is also visualized. Instead of visualizing the relation between model transformation and metamodels, the relation between model transformation and models could be visualized. In this way, the relation between source and target model element is visualized. This can for example be used for impact analysis, i.e., to study the effects of a change in a source model on the corresponding target model, or for origin tracking, i.e., to trace the source of, for example, an error in a target model. The versatility of this visualization can be further improved by supporting visualization of transformation chains.

In Section 3.4.2.1, we described a method aimed at assessing the correctness of model transformations using model checking. In Chapter 8, we showed how this method can be facilitated using fine-grained model transformations. This is a promising first step towards assessing the correctness of model transformations. However, this method provides a partial answer only. Moreover, it is solely applicable to model transformations that transform behavioral models and only if these models can be transformed into models suitable for model checking. Finally, since it involves model checking, it suffers from the state-space explosion problem. To assess full correctness of model transformations, other techniques are required, such as proving correctness. However, to construct such proofs, it is required that the semantics of the source and target language are formally defined. Since model transformations are typically defined on DSLs, this implies that the semantics of DSLs need to be formally defined during their development and that ad hoc informal semantics does not suffice. An approach to formally verify that a transformation ensures semantic equivalence between a source model and the corresponding target model is proposed by Giese et al. [131].

Model transformations are similar to traditional software artifacts. Therefore, they should be treated in a similar way, especially when MDE will be widely adopted by industry. Similarly to traditional software development, there should be a solid development trajectory for model transformations. Moreover, methods and techniques are required for specifying model transformations. The arrangement of such a development trajectory, as well as determining the requirements for a specification formalism for model transformations are topics for further research.

Bibliography

[1] É. Lévénez. Computer Languages History. `http://www.levenez.com/lang/`. (Cited on page 1.)

[2] D. C. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, February 2006. (Cited on page 2.)

[3] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the Sixteenth Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280, San Diego, CA, USA, November 2001. IEEE Computer Society. (Cited on page 2.)

[4] R. Magritte. La Trahison des Images. Oil on canvas painting, 1929. Los Angeles County Museum of Art (LACMA). ≪©2011 Photo SCALA, Florence≫. (Cited on page 2.)

[5] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, June 2000. (Cited on pages 3, 4, and 5.)

[6] M. Fowler. *Domain-Specific Languages*. The Addison-Wesley Signature Series. Addison-Wesley, first edition, 2010. (Cited on page 3.)

[7] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, September 2003. (Cited on pages 3, 7, and 8.)

[8] A. van Deursen and P. Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance*, 10(2):75–92, March–April 1998. (Cited on page 4.)

[9] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A Software Engineering Experiment in Software Component Generation. In *Proceedings of the Eighteenth International Conference on Software Engineering (ICSE 1996)*, pages 542–552, Berlin, Germany, March 1996. IEEE Computer Society. (Cited on page 4.)

[10] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005. (Cited on page 4.)

[11] J. M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, 10(5):564–574, September 1984. (Cited on page 4.)

[12] G. Arango. Domain Analysis: From Art Form To Engineering Discipline. *SIGSOFT Software Engineering Notes*, 14(3):152–159, April 1989. (Cited on page 4.)

[13] S. Mauw, W. T. Wiersma, and T. Willemse. Language-Driven System Design. In *Proceedings of the Thirtyfifth Hawaii International Conference on System Scienses (HICSS 2002)*, pages 3637–3646, Big Island, HI, USA, January 2002. IEEE Computer Society. (Cited on page 5.)

[14] E. Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, September–October 2003. (Cited on page 5.)

[15] I. Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, Enschede, The Netherlands, May 2005. (Cited on pages 5 and 7.)

[16] OMG. Meta Object Facility specification. formal/ 2002-04-03, Object Management Group, April 2004. (Cited on page 5.)

[17] L. Tratt. Model transformations and tool integration. *Software and Systems Modeling*, 4(2):112–122, May 2005. (Cited on pages 6 and 7.)

[18] R. Mannadiar and H. Vangheluwe. Debugging in Domain-Specific Modelling. In B. A. Malloy, S. Staab, and M. G. J. van den Brand, editors, *Proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *Lecture Notes in Computer Science*, pages 276–285, Eindhoven, The Netherlands, October 2010. Springer Verlag. (Cited on page 6.)

[19] OMG. MDA Guide, Version 1.0.1. omg/ 2003-06-01, Object Management Group, June 2003. (Cited on page 6.)

[20] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model-Driven Architecture: Practice and Promise*. Addison-Wesley Object Technology Series. Addison-Wesley, 2003. (Cited on pages 6 and 8.)

[21] T. Mens and P. van Gorp. A Taxonomy of Model Transformation. In G. Karsai and G. Taentzer, editors, *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 125–142, Tallinn, Estonia, September 2006. Elsevier. (Cited on pages 7, 51, 53, and 87.)

[22] A. Thue. Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. In *Skrifter Utgit Av Videnskapsselskapet I Kristiania, I. Matematisk-Naturvidenskapelig Klasse*, volume 10, pages 1–34. May 1914. (Cited on page 7.)

[23] P. Stevens. A Landscape of Bidirectional Model Transformations. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Revised Papers of the Second International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer Verlag, Braga, Portugal, July 2007. (Cited on page 7.)

[24] J. Bézivin. Model Driven Engineering: An Emerging Technical Space. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Revised Papers of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer Verlag, Braga, Portugal, July 2005. (Cited on page 7.)

[25] OMG. MOF 2.0/XMI Mapping, Version 2.1.1. formal/ 2007-12-01, Object Management Group, December 2007. (Cited on pages 9, 37, and 38.)

[26] W3C. XSL Transformations (XSLT) Version 2.0. W3C Recommendation REC-xslt20-20070123, World Wide Web Consortium, January 2007. (Cited on page 9.)

[27] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly Media, third edition, 2000. (Cited on page 9.)

[28] M. Lutz. *Programming Python*. O'Reilly Media, second edition, 2001. (Cited on page 9.)

[29] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. The Facets of Ruby Series. Pragmatic Programmers, third edition, 2009. (Cited on page 9.)

[30] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The Missing Link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, number 2505 in Lecture Notes in Computer Science, pages 90–105, Barcelona, Spain, October 2002. Springer Verlag. (Cited on pages 9 and 41.)

[31] J. Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE*, V(2):21–24, April 2004. (Cited on page 10.)

[32] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *Proceedings of the Second OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, USA, 2003. (Cited on page 10.)

[33] F. Jouault and I. Kurtev. Transforming Models with ATL. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, number 3844 in Lecture Notes in Computer Science, pages 128–138, Montego Bay, Jamaica, October 2005. Springer Verlag. (Cited on pages 10 and 11.)

[34] F. Jouault, F. Allilaire, J. Bézivin, and I Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008. (Cited on pages 10 and 76.)

[35] OMG. Request for Proposal: MOF 2.0 Query/Views/Transformations RFP. ad/ 2002-04-10, Object Management Group, April 2002. (Cited on page 10.)

[36] F. Jouault and I. Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC 2006)*, pages 1188–1195, Dijon, France, April 2006. ACM. (Cited on page 10.)

[37] A. Vallecillo, J. Gray, and A. Pierantonio, editors. *Theory and Practice of Model Transformations, Proceedings of the First International Conference on Model Transformation (ICMT 2008)*, volume 5063 of *Lecture Notes in Computer Science*, Zurich, Switzerland, July 2008. Springer Verlag. (Cited on pages 10, 14, and 172.)

[38] R. F. Paige, editor. *Theory and Practice of Model Transformations, Proceedings of the Second International Conference on Model Transformation*

(ICMT 2009), volume 5563 of *Lecture Notes in Computer Science*, Zurich, Switzerland, June 2009. Springer Verlag. (Cited on pages 10, 14, and 173.)

[39] L. Tratt and M. Gogolla, editors. *Theory and Practice of Model Transformations, Proceedings of the Third International Conference on Model Transformation (ICMT 2010)*, volume 6142 of *Lecture Notes in Computer Science*, Málaga, Spain, July 2010. Springer Verlag. (Cited on pages 10 and 14.)

[40] E. Visser and J. Cabot, editors. *Theory and Practice of Model Transformations, Proceedings of the Fourth International Conference on Model Transformation (ICMT 2011)*, volume 6707 of *Lecture Notes in Computer Science*, Zurich, Switzerland, June 2011. Springer Verlag. (Cited on pages 10, 14, 174, and 179.)

[41] OMG. Object Constraint Language, Version 2.2. formal/ 2010-02-01, Object Management Group, February 2010. (Cited on pages 11 and 82.)

[42] ATL User Guide. `http://wiki.eclipse.org/ATL/User_Guide`. (Cited on page 11.)

[43] E. Clayberg and D. Rubel. *Eclipse Plug-ins*. The Eclipse Series. Addison-Wesley, third edition, 2009. (Cited on page 11.)

[44] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, second edition, 2008. (Cited on page 11.)

[45] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. formal/ 2008-04-03, Object Management Group, April 2008. (Cited on page 11.)

[46] SmartQVT Website. `http://smartqvt.elibel.tm.fr/`. (Cited on page 12.)

[47] Eclipse M2M Website. `http://www.eclipse.org/m2m/`. (Cited on page 12.)

[48] A. Haase, M. Vlter, S. Efftinge, and B. Kolb. Introduction to openArchitectureWare 4.1.2. In *Model-Driven Development Tool Implementers Forum (MDD-TIF 2007) (co-located with TOOLS 2007)*, 2007. (Cited on page 12.)

[49] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing, 1996. (Cited on pages 12, 37, and 58.)

[50] M. F. van Amstel, M. G. J. van den Brand, Z. Protić, and T. Verhoeff. Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap? In Vallecillo et al. [37], pages 61–75. (Cited on pages 12, 17, and 67.)

[51] L. J. P. Engelen and M. G. J. van den Brand. Integrating Textual and Graphical Modelling Languages. In T. Ekman and J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 105–120, York, England, March 2010. Elsevier. (Cited on pages 12 and 67.)

[52] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF —Reference Manual—. *SIGPLAN Notices*, 24(11):43–75, 1989. (Cited on pages 12 and 58.)

[53] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, chapter 1, pages 1–66. ACM Press, 1989. (Cited on pages 13 and 58.)

[54] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In R. Wilhelm, editor, *Proceedings of the Tenth International Conference on Compiler Construction (CC 2001)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370, Genova, Italy, April 2001. Springer Verlag. (Cited on pages 13, 37, and 68.)

[55] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In R. N. Horspool, editor, *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer Verlag. (Cited on page 13.)

[56] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani. MDE Adoption in Industry: Challenges and Success Criteria. In M. R. V. Chaudron, editor, *Models in Software Engineering: Workshops and Symposia at MoDELS 2008 Reports and Revised Selected Papers*, volume 5095 of *Lecture Notes in Computer Science*, pages 54–59, Toulouse, France, September–October 2008. Springer Verlag. (Cited on pages 13 and 45.)

[57] A. Pierantonio, A. Vallecillo, B. Selic, and J. Gray. Special issue on model transformation. *Science of Computer Programming*, 68(3):111–113, October 2007. (Cited on page 14.)

[58] P. Mohagheghi and V. Dehlen. Where Is the Proof? – A Review of Experiences from Applying MDE in Industry. In I. Schieferdecker and A. Hartman, editors, *Proceedings of the Fourth European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2008)*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443, Berlin, Germany, June 2008. Springer Verlag. (Cited on page 14.)

[59] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Co., second edition, 1996. (Cited on pages 15, 52, 57, 71, and 97.)

[60] M. F. van Amstel, C. F. J. Lange, and M. G. J. van den Brand. Metrics for Analyzing the Quality of Model Transformations. In G. Falcone, Y.-G. Guéhéneuc, C. F. J. Lange, Z. Porkoláb, and H. A. Sahraoui, editors, *Proceedings of the Twelfth ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering (QAOOSE 2008) (co-located with ECOOP 2008)*, pages 41–51, Paphos, Cyprus, July 2008. (Cited on page 17.)

[61] M. F. van Amstel. The Right Tool for the Right Job: Measuring Model Transformation Quality. In *Proceedings of the Fourth IEEE International Workshop on Quality Oriented Reuse of Software (QUORS 2010) in proceedings of Thirty-Fourth Annual International Computer Software and Applications Conference (COMPSAC 2010)*, pages 69–74, Seoul, South Korea, July 2010. IEEE Computer Society. (Cited on page 17.)

[62] M. F. van Amstel, C. F. J. Lange, and M. G. J. van den Brand. Using Metrics for Assessing the Quality of ASF+SDF Model Transformations. In Paige [38], pages 239–248. (Cited on page 18.)

[63] M. F. van Amstel and M. G. J. van den Brand. Quality Assessment of ATL Model Transformations using Metrics. In M. D. Del Fabro, F. Jouault, and I. Kurtev, editors, *Proceedings of the Second International Workshop on Model Transformation with ATL (MtATL 2010)*, volume 711 of *CEUR Workshop Proceedings*, pages 19–33, Málaga, Spain, June 2010. CEUR-WS.org. (Cited on pages 18 and 95.)

[64] M. F. van Amstel and M. G. J. van den Brand. Using Metrics for Assessing the Quality of ATL Model Transformations. In I. Kurtev, M. Tisi, and D. Wagelaar, editors, *Proceedings of the Third International Workshop on Model Transformation with ATL (MtATL 2011)*, volume 742 of *CEUR*

Workshop Proceedings, pages 20–34, Zurich, Switzerland, July 2011. CEUR-WS.org. (Cited on page 18.)

[65] M. F. van Amstel, M. G. J. van den Brand, and P. H. Nguyen. Metrics for Model Transformations. In *Proceedings of the Ninth Belgian-Netherlands Software Evolution Workshop (BENEVOL 2010)*, Lille, France, December 2010. (Cited on page 18.)

[66] M. F. van Amstel and M. G. J. van den Brand. Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In Visser and Cabot [40], pages 108–122. (Cited on page 19.)

[67] M. F. van Amstel, M. G. J. van den Brand, and L. J. P. Engelen. An Exercise in Iterative Domain-Specific Language Design. In A. Capiluppi, A. Cleve, and N. Moha, editors, *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL 2010)*, ACM International Conference Proceeding Series, pages 48–57, Antwerp, Belgium, September 2010. ACM. (Cited on pages 19 and 95.)

[68] M. F. van Amstel, M. G. J. van den Brand, and L. J. P. Engelen. Using a DSL and Fine-grained Model Transformations to Explore the Boundaries of Model Verification – Extended Abstract. In *Proceedings of the Seventh Workshop on Advances in Model Based Testing (A-MOST 2011) (co-located with ICST 2011)*, pages 63–66, Berlin, Germany, March 2011. IEEE Computer Society. (Cited on page 19.)

[69] M. F. van Amstel, M. G. J. van den Brand, and L. J. P. Engelen. Using a DSL and Fine-grained Model Transformations to Explore the Boundaries of Model Verification. In *Proceedings of the Third Workshop on Model-Based Verification & Validation From Research to Practice (MVV 2011) (co-located with SSIRI 2011)*, pages 120–127, Jeju Island, Korea, June 2011. IEEE Computer Society. (Cited on page 19.)

[70] A. L. Rubio. Metamodeling and Formalisms for Representation of Behavior. In *Twelfth Workshop for PhD students in Object-Oriented Systems (co-located with ECOOP 2002)*, Málaga, Spain, June 2002. (Cited on page 21.)

[71] J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Science*, 335(2–3):131–146, May 2005. (Cited on pages 22 and 23.)

[72] OMG. Unified Modeling Language: Superstructure specification, version 2.1.1. formal/ 2007-02-05, Object Management Group, August 2007. (Cited on pages 22, 25, 28, and 138.)

[73] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. The Addison-Wesley Object Technology Series. Addison-Wesley, second edition, 2005. (Cited on pages 22 and 25.)

[74] M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, December 2002. (Cited on page 22.)

[75] H. Fecher and J. Schöborn. UML 2.0 State Machines: Complete Formal Semantics via Core State Machines. In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology, Proceedings of the Eleventh International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2006) and the Fifth International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2006), Revised Selected Papers*, volume 4346 of *Lecture Notes in Computer Science*, pages 244–260, Bonn, Germany, August 2006. Springer Verlag. (Cited on page 22.)

[76] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming*, 51(1):43–75, April–May 2002. (Cited on page 22.)

[77] J. Lilius and I. Porres Paltor. The Semantics of UML State Machines. TUCS Technical Report 273, Turku Centre for Computer Science, May 1999. (Cited on page 22.)

[78] J. A. Bergstra and J. W. Klop. Algebra Of Communicating Processes. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 89–138. North-Holland, 1986. (Cited on pages 22 and 23.)

[79] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990. (Cited on pages 22 and 23.)

[80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980. (Cited on page 23.)

[81] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978. (Cited on page 23.)

[82] J. Davies and S. A. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138(2):243–271, February 1995. (Cited on page 23.)

[83] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal Specification Language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS 2006)*, number 06351 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik, August–September 2006. (Cited on page 23.)

[84] L. Cheng. *Timed Processes: Models, Axioms and Decidability.* PhD thesis, University of Edinburgh, Edinburgh, United Kingdom, 1993. (Cited on page 23.)

[85] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60(1–3):109–137, January–March 1984. (Cited on pages 24, 25, and 36.)

[86] J. C. M. Baeten, T. Basten, and M. A. Reniers. Process Algebra: Equational Theories of Communicating Processes. Lecture notes (DRAFT), 2007. (Cited on page 28.)

[87] A. J. H. Simons. On the Compositional Properties of UML Statechart Diagrams. In *Proceedings of the Third Workshop on Rigorous Object-Oriented Methods (ROOM 2000)*, Workshops in Computing, pages 1–12, York, UK, January 2000. BCS. (Cited on page 30.)

[88] Telelogic Rhapsody 7.1.1. `http://modeling.telelogic.com/products/rhapsody/index.cfm`. (Cited on pages 37 and 38.)

[89] K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Communications of the ACM*, 11(6):419–422, June 1968. (Cited on page 37.)

[90] ArgoUML v0.24. `http://argouml.tigris.org/`. (Cited on page 38.)

[91] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, R., and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the Twenty-second International Conference on Software Engineering (ICSE 2000)*, pages 439–448, Limerick, Ireland, June 2000. ACM. (Cited on page 40.)

[92] A. Sabetta, D. C. Petriu, V. Grassi, and R. Mirandola. Abstraction-Raising Transformation for Generating Analysis Models. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 217–226. Springer Verlag, October 2005. (Cited on page 41.)

[93] J. Sawada. ACL2VHDL Translator: A Simple Approach to Fill the Semantic Gap. In *Proceedings of the Fifth International Workshop of the ACL2 Theorem Prover and its Applications*, November 2004. (Cited on page 41.)

[94] R. Grangel, M. Bigand, and J.-P. Bourey. A UML Profile as Support for Transformation of Business Process Models at Enterprise Level. In J.-P. Bourey, R. Grangel Seguer, X. Franch, Ela Hunt, and Remi Coletta, editors, *Proceedings of the First International Workshop on Model Driven Interoperability for Sustainable Information Systems (MDISIS 2008)*, volume 340 of *CEUR Workshop Proceedings*, pages 73–87. CEUR-WS.org, June 2008. (Cited on page 41.)

[95] A. Rountev, O. Volgin, and M. Reddoch. Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams. In M. Ernst and T. Jensen, editors, *Proceedings of the Sixth ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE 2005) (co-located with ESEC/FSE 2005)*, pages 96–102, Lisbon, Portugal, September 2005. ACM. (Cited on page 41.)

[96] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30(2):181–202, 1993. (Cited on page 41.)

[97] J. J. Pardo, V. Valero, F. Cuartero, and D. Cazorla. Automatic Translation of a Timed Process Algebra into Dynamic State Graphs. In *Proceedings of the Eith Asia-Pacific Conference on Software Engineering (APSEC 2001)*, pages 63–70, Macau, China, December 2001. IEEE Computer Society. (Cited on page 42.)

[98] A. Ferrara. Web Services: a Process Algebra Approach. In *Proceedings of the Second international conference on Service oriented computing (ICSOC 2004)*, pages 242–251, New York, NY, USA, November 2004. ACM. (Cited on page 42.)

[99] S. Doxsee and W. B. Gardner. Synthesis of C++ Software from Verifiable CSPm Specifications. In J. Rozenblit, T. O'Neill, and J. Peng, editors, *Proceedings of the Twelfth IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 2005)*, pages 193–201, Greenbelt, MD, USA, April 2005. IEEE Computer Society. (Cited on page 42.)

[100] IEEE Standards Board. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, The Institute of Electrical and Electronics Engineers, September 1990. (Cited on page 45.)

[101] P. Mohagheghi and J. Aagedal. Evaluating Quality in Model-Driven Engineering. In *Proceedings of the International Workshop on Modeling in Software Engineering (MISE 2007)*, pages 1–6, Minneapolis, MN, USA, May 2007. IEEE Computer Society. (Cited on pages 46 and 57.)

[102] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992. (Cited on page 46.)

[103] N. Ashley. *Measurement as a Powerful Software Management Tool*. International Software Quality Assurance Series. McGraw-Hill, 1995. (Cited on page 46.)

[104] D. A. Garvin. What Does "Product Quality" Really Mean? *MIT Sloan Management Review*, 26(1):25–43, October 1984. (Cited on page 46.)

[105] J. J. M. Trienekens and E. P. W. M. van Veenendaal. *Software Quality from a Business Perspective*. Kluwer, 1997. (Cited on page 47.)

[106] B. Kitchenham and S. L. Pfleeger. Software Quality: The Elusive Target. *IEEE Software*, 13(1):12–19, January 1996. (Cited on page 47.)

[107] J. M. Juran, F. M. Gryna, and R. S. Bingham. *Quality Control Handbook*. McGraw-Hill, third edition, 1974. (Cited on page 47.)

[108] P. B. Crosby. *Quality Is Free: The Art of Making Quality Certain*. McGraw-Hill, 1979. (Cited on page 47.)

[109] ISO – International Organization for Standardization. *ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model*. Geneva, Switzerland, 2001. (Cited on page 48.)

[110] R. Plösch, H. Gruber, A. Hentschel, Ch. Körner, G. Pomberger, S. Schiffer, M. Saft, and S. Storck. The EMISQ method and its tool support-expert-based evaluation of internal software quality. *Innovations in Systems and Software Engineering*, 4(1):3–15, April 2008. (Cited on page 48.)

[111] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in Software Quality. Final Technical Report RADC-TR-77-369, Volume 1 (of three), Rome Air Development Center, Rome, NY, USA, November 1977. (Cited on page 48.)

[112] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978. (Cited on pages 48, 50, and 139.)

[113] ISO – International Organization for Standardization. *ISO/IEC 25000 SQuaRE - Software Product Quality Requirements and Evaluation*. Geneva, Switzerland, 2005. (Cited on page 48.)

[114] G. Canfora, L. Mancini, and M. Tortorella. A Workbench for Program Comprehension during Software Maintenance. In A. Cimitile and H. A. Müller, editors, *Proceedings of the Fourth Workshop on Program Comprehension (WPC 1996)*, pages 30–39, Berlin, Germany, March 1996. IEEE Computer Society. (Cited on page 49.)

[115] M.-A. D. Storey, K. Wong, and H. A. Muller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, March 2000. (Cited on pages 49 and 116.)

[116] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(5):621–645, July 2006. (Cited on pages 49, 63, and 117.)

[117] D. L. Parnas. On the Criteria To Be Used in Decompsing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. (Cited on page 49.)

[118] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, 1974. (Cited on page 50.)

[119] B. Nuseibeh. To Be *and* Not to Be: On Managing Inconsistency in Software Development. In *Proceedings of the Eighth IEEE International Workshop on Software Specification and Design (IWSSD 1996)*, pages 164–169, Schloss Velen, Germany, March 1996. IEEE Computer Society. (Cited on page 50.)

[120] Z. Protić. *Configuration management for models: Generic methods for model comparison and model co-evolution*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, October 2011. (Cited on pages 51 and 53.)

[121] M. F. van Amstel, S. Bosems, I. Kurtev, and L. Ferreira Pires. Performance in Model Transformations: A Comparison Between ATL and QVT. In Visser and Cabot [40], pages 198–212. (Cited on pages 52 and 101.)

[122] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. (Cited on pages 52 and 86.)

[123] C. F. J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. PhD thesis, Eindhoven University

of Technology, Eindhoven, The Netherlands, 2007. (Cited on pages 52 and 78.)

[124] J. Mendling. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*, volume 6 of *Lecture Notes in Business Information Processing*. Springer Verlag, 2008. (Cited on page 52.)

[125] M. Saeki and H. Kaiya. Measuring Model Transformation in Model Driven Development. In J. Eder, S. L. Tomassen, A. L. Opdahl, and G. Sindre, editors, *Proceedings of the CAiSE 2007 Forum at the Nineteenth International Conference on Advanced Information Systems Engineering*, volume 247 of *CEUR Workshop Proceedings*, pages 77–80, Trondheim, Norway, June 2007. CEUR-WS.org. (Cited on page 53.)

[126] A. Toulmé. Presentation of EMF Compare Utility. In *Eclipse Modeling Symposium (co-located with Eclipse Summit Europe 2006)*, Esslingen, Germany, October 2006. (Cited on page 53.)

[127] N. Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the Twenty-seventh International Conference on Software Engineering (ICSE 2005)*, pages 284–292, St. Louis, MO, USA, May 2005. ACM. (Cited on page 53.)

[128] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999. (Cited on pages 53, 132, and 140.)

[129] D. Varró and A. Pataricza. Automated Formal Verification of Model Tranformations. In J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, editors, *Proceedings of the 2003 Workshop on Critical Systems Development in UML (CSDUML 2003), Technical Report TUM-I0323*, pages 63–78, San Francisco, USA, October 2003. Technische Universität München. (Cited on page 54.)

[130] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and Systems Modeling*, 3(2):85–113, May 2004. (Cited on page 54.)

[131] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards Verified Model Transformations. In D. Hearnden, J. G. Süß, B. Baudry, and N. Rapin, editors, *Proceedings of the Third International Workshop on Model Development, Validation and Verification (MoDeV²a 2006)*, pages 78–93, Genova, Italy, October 2006. Le Commissariat à l'Energie Atomique – CEA. (Cited on pages 55 and 165.)

[132] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addsion Wesley Series in Computer Science. Addison-Wesley, first edition, 1979. (Cited on page 59.)

[133] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term Rewriting with Traversal Functions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):152–190, April 2003. (Cited on pages 61 and 63.)

[134] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981. (Cited on pages 62 and 110.)

[135] M. Shepperd. Early life-cycle metrics and software quality models. *Information and Software Technology*, 32(4):311–316, May 1990. (Cited on page 62.)

[136] R. C. Martin. Object Oriented Design Quality Metrics: An Analysis of Dependencies. *ROAD*, 2(3), September–October 1995. (Cited on page 64.)

[137] M. Marchesi. OOA Metrics for the Unified Modeling Langauge. In *Proceedings of the Second Conference on Software Maintenance and Reengineering (CSMR 1998)*, pages 67–73, Florence, Italy, March 1998. IEEE Computer Society. (Cited on page 64.)

[138] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1-2):35–61, April–May 2004. (Cited on page 66.)

[139] J. A. Kreibich. *Using SQLite.* O'Reilly Media, first edition, 2010. (Cited on page 66.)

[140] E. Gansner, E. Koutsofios, and S. North. *Drawing graphs with dot*, February 2002. (Cited on page 67.)

[141] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, July 2002. (Cited on page 67.)

[142] B. J. Arnoldus, J. Bijpost, and M. G. J. van den Brand. REPLEO: a Syntax-Safe Template Engine. In *Proceedings of the sixth International Conference on Generative Programming and Component Engineering (GPCE 2007)*, pages 25–32, Salzburg, Austria, October 2007. ACM. (Cited on page 67.)

[143] R. Likert. A technique for measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932. (Cited on pages 68 and 96.)

[144] A. Field. *Discovering Statistics using SPSS*. Sage, second edition, 2005. (Cited on pages 71 and 97.)

[145] H. Abdi. The Kendall Rank Correlation Coefficient. In N. J. Salkind, editor, *Encyclopedia of Measurements and Statistics*, volume 2, pages 508–510. Sage, 2007. (Cited on pages 73 and 98.)

[146] S. J. Dubner and S. D. Levitt. *Freakonomics*. Penguin UK, revised edition, 2006. (Cited on page 73.)

[147] R. K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage, fourth edition, 2009. (Cited on pages 76 and 101.)

[148] N. Blaikie. *Analyzing Quantitative Data: From Description to Explanation*. Sage, 2003. (Cited on page 77.)

[149] D. L. Clason and T. J. Dormody. Analyzing Data Measured by Individual Likert-Type Items. *Journal of Agricultural Education*, 35(4):31–35, 1994. (Cited on page 77.)

[150] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions On Software Engineering*, 27(7):630–650, July 2001. (Cited on page 77.)

[151] R. Harrison. Quantifying internal attributes of functional programs. *Information and Software Technology*, 35(10):554–560, 1993. (Cited on page 77.)

[152] R. Harrison, L. G. Samaraweera, M. R. Dobie, and P. H. Lewis. Estimating the quality of functional programs: an empirical investigation. *Information and Software Technology*, 37(12):701–707, 1995. (Cited on page 77.)

[153] T. Alves and J. Visser. Metrication of SDF Grammars. Technical Report DI-PURe-05.05.01, Departamento de Informática da Universidade do Minho, Braga, Portugal, May 2005. (Cited on pages 77 and 112.)

[154] L. Kapová, T. Goldschmidt, S. Becker, and J. Henss. Evaluating Maintainability with Code Metrics for Model-to-Model Transformations. In G. Heineman, J. Kofron, and F. Plasil, editors, *Proceedings of the Sixth International Conference on the Quality of Software Architectures (QoSA 2010)*, volume 6093 of *Lecture Notes in Computer Science*, pages 151–166, Prague, Czech Republic, June 2010. Springer Verlag. (Cited on page 78.)

[155] V. R. Basili, L. C. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751 – 761, October 1996. (Cited on pages 78, 85, 101, and 109.)

[156] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Object Technology. Addison-Wesley, 1999. (Cited on page 79.)

[157] M. Wimmer and G. Kramler. Bridging Grammarware and Modelware. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, number 3844 in Lecture Notes in Computer Science, pages 159–168, Montego Bay, Jamaica, October 2005. Springer Verlag. (Cited on page 81.)

[158] A. Kleppe. A Language Description is More than a Metamodel. In *Proceedings of the Fourth International Workshop on Software Language Engineering (SLE 2007)*, Nashville, TN, USA, October 2007. (Cited on page 81.)

[159] ATL Transformations. `http://www.eclipse.org/m2m/atl/atlTransformations/`. (Cited on pages 83, 94, and 95.)

[160] A. Vignaga. Metrics for Measuring ATL Model Transformations. Technical report, MaTE, Department of Computer Science, Universidad de Chile, 2009. (Cited on page 85.)

[161] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976. (Cited on pages 89 and 108.)

[162] J. Cabot and E. Teniente. A metric for measuring the complexity of OCL expressions. In *Workshop on Model Size Metrics (co-located with MODELS 2006)*, Genova, Italy, October 2006. (Cited on page 89.)

[163] OMG. Architecture-Driven Modernization (ADM): Software Metrics Meta-Model (SMM). ptc/ 2009-03-03, Object Management Group, March 2009. (Cited on page 92.)

[164] K. Thoms and S. Efftinge. Xpand. `http://wiki.eclipse.org/Xpand`. (Cited on page 92.)

[165] PicoJava Checker. `http://jastadd.org/jastadd-tutorial-examples/picojava-checker`. (Cited on page 94.)

[166] J. Muñoz, M. Llacer, and B. Bonet. Configuring ATL Transformations in MOSKitt. In M. D. Del Fabro, F. Jouault, and I. Kurtev, editors, *Proceedings*

of the Second International Workshop on Model Transformation with ATL (MtATL 2010), volume 711 of *CEUR Workshop Proceedings*, pages 50–59, Málaga, Spain, June 2010. CEUR-WS.org. (Cited on pages 94 and 95.)

[167] D. Wagelaar. MDE Case Studies. `http://soft.vub.ac.be/soft/research/mdd:casestudies`. (Cited on page 95.)

[168] M. Page-Jones. *The practical guide to structured systems design*. Yourdon Press, second edition, 1980. (Cited on pages 100 and 109.)

[169] S. Black. The Role of Ripple Effect in Software Evolution. In N. H. Madhavji, J. Fernandez-Ramil, and D. E. Perry, editors, *Software Evolution and Feedback: Theory and Practice*, chapter 12, pages 249–268. Wiley, 2006. (Cited on page 100.)

[170] D. M. Oppenheimer, T. Meyvis, and N. Davidenko. Instructional manipulation checks: Detecting satisficing to increase statistical power. *Journal of Experimental Psychology*, 45(4):867–872, July 2009. (Cited on page 102.)

[171] P. H. Nguyen. Quantitative Analysis of Model Transformations. Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2010. Supervised by M.G.J. van den Brand and M.F. van Amstel. `http://alexandria.tue.nl/extra1/afstversl/wsk-i/nguyen2010.pdf`. (Cited on page 105.)

[172] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *Proceedings of the Fifteenth IEEE International Conference on Program Comprehension (ICPC 2007)*, pages 49–58, Banff, AB, Canada, June 2007. IEEE Computer Society. (Cited on page 117.)

[173] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974. (Cited on page 119.)

[174] D. Holten and J. J. van Wijk. Visual Comparison of Hierarchically Organized Data. *Computer Graphics Forum*, 27(3):759–766, May 2008. (Cited on page 121.)

[175] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society, 1996. (Cited on page 123.)

[176] J.-M. Favre. Meta-Model and Model Co-evolution within the 3D Software Space. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA 2003)*, pages 98–109, Amsterdam, The Netherlands, September 2003. (Cited on page 123.)

[177] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2008)*, pages 222–231, Munich, Germany, September 2008. IEEE Computer Society. (Cited on page 123.)

[178] K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In B. Baudry, A. Faivre, S. Ghosh, and A. Pretschner, editors, *Proceedings of the fourth workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa 2007)*, pages 47–56, 2007. (Cited on page 128.)

[179] J. Wang, S.-K. Kim, and D. Carrington. Verifying metamodel coverage of model transformations. In J. Han and M. Staples, editors, *Proceedings of the Australian Software Engineering Conference (ASWEC 2006))*, pages 270–282, Sydney, Australia, April 2006. IEEE Computer Society. (Cited on page 129.)

[180] M. Lawley and J. Steel. Practical Declarative Model Transformation with Tefkat. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 139–150, Motego Bay, Jamaica, October 2005. Springer Verlag. (Cited on page 129.)

[181] J. A. McQuillan and J. F. Power. White-Box Coverage Criteria for Model Transformations. In F. Jouault, editor, *Proceedings of the First International Workshop on Model Transformation with ATL (MtATL 2009)*, Nantes, France, July 2009. (Cited on page 129.)

[182] E. Planas, J. Cabot, and C. Gómez. Two Basic Correctness Properties for ATL Transformations: Executability and Coverage. In I. Kurtev, M. Tisi, and D. Wagelaar, editors, *Proceedings of the Third International Workshop on Model Transformation with ATL (MtATL 2011)*, volume 742 of *CEUR Workshop Proceedings*, pages 1–9, Zurich, Switzerland, July 2011. CEUR-WS.org. (Cited on page 129.)

[183] J. von Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. Constructing and Visualizing Transformation Chains. In I. Schieferdecker and A. Hartman, editors, *Proceedings of the Fourth European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2008)*, volume 5095 of *Lecture Notes in Computer Science*, pages 17–32, Berlin, Germany, June 2008. Springer Verlag. (Cited on page 129.)

[184] J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, September 2006. (Cited on page 130.)

[185] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998. (Cited on pages 132 and 156.)

[186] E. M. Clarke Jr., O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. (Cited on pages 132 and 156.)

[187] J. M. Wing and M. Vaziri-Farahani. Model Checking Software Systems: A Case Study. *SIGSOFT Software Engineering Notes*, 20(4):128–139, October 1995. (Cited on pages 132 and 156.)

[188] I. Kurtev, K. van den Berg, and M. Akşit. UML to XML-Schema Transformation: a Case Study in Managing Alternative Model Transformations in MDA. In *Forum on Specification and Design Languages (FDL'03)*, Frankfurt, Germany, September 2003. (Cited on page 132.)

[189] D. Baum. *NQC Programmer's Guide*, 2003. (Cited on pages 132 and 140.)

[190] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. (Cited on pages 133 and 140.)

[191] A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley, 2008. (Cited on page 136.)

[192] S. Andova, M. G. J. van den Brand, and L. J. P. Engelen. Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models. In F. Durán and V. Rusu, editors, *Proceedings of the Second International Workshop on Algebraic Methods in Model-based Software Engineering (AMMSE 2011)*, pages 65–79, Zurich, Switzerland, June 2011. (Cited on page 136.)

[193] W. H. J. Feijen and A. J. M. van Gasteren. *On a Method of Multiprogramming*. Monographs in Computer Science. Springer Verlag, 1999. (Cited on pages 143 and 145.)

[194] J. C. M. Baeten and C. A. Middelburg. *Process Algebra with Timing*. Monographs in Theoretical Computer Science An EATCS Series. Springer Verlag, 2002. (Cited on page 143.)

[195] R. Williams. Finite State Machines – Implementation Options. In *Real-Time Systems Development*, chapter 6, pages 110–149. Elsevier, 2006. (Cited on page 147.)

[196] G. J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proceedings of the Third International Spin Workshop*, Enschede, The Netherlands, April 1997. (Cited on page 155.)

[197] G. Weißenbacher and W. Herzner. CEDAR: Counter-Example Driven Abstraction Refinement – A Pattern Supporting Formal Verification of Large Systems. In *Proceedings of the Tenth European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, pages 1–11, Irsee, Germany, July 2005. (Cited on page 156.)

[198] J.-R. Abrial, M. K. O. Lee, D. Neilson, P. N. Scharbach, and I. H. Sørensen. The B-Method. In S. Prehn and W. J. Toetenel, editors, *Proceedings of the Fourth International Symposium of VDM Europe*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405, Noordwijkerhout, The Netherlands, October 1991. Springer Verlag. (Cited on page 156.)

[199] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002. (Cited on page 164.)

Appendix A

ACP Axioms

A1	$x + y = y + x$	
A2	$(x + y) + z = x + (y + z)$	
A3	$x + x = x$	
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$	
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	
A6	$x + 0 = x$	
A7	$0 \cdot x = 0$	
A8	$x \cdot 1 = x$	
A9	$1 \cdot x = x$	
A10	$a.x \cdot y = a.(x \cdot y)$	
D1	$\partial_H(1) = 1$	
D2	$\partial_H(0) = 0$	
D3	$\partial_H(a.x) = 0$	if $a \in H$
D4	$\partial_H(a.x) = a.\partial_H(x)$	if $a \notin H$
D5	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	
M	$x \| y = x \lfloor\lfloor y + y \lfloor\lfloor x + x \mid y$	

LM1	$0 \lfloor\!\lfloor x = 0$	
LM2	$1 \lfloor\!\lfloor x = 0$	
LM3	$a.x \lfloor\!\lfloor y = a.(x \| y)$	
LM4	$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	
CM1	$0 \mid x = 0$	
CM2	$(x + y) \mid z = x \mid z + y \mid z$	
CM3	$1 \mid 1 = 1$	
CM4	$a.x \mid 1 = 0$	
CM5	$a.x \mid b.y = c.(x \| y)$	if $\gamma(a, b) = c$
CM6	$a.x \mid b.y = 0$	if $\gamma(a, b)$ undefined
CM7	$a.x \mid b.y = (a \mid b) \cdot (x \| y)$	
CM8	$(x + y) \mid z = x \mid z + y \mid z$	
CM9	$x \mid (y + z) = x \mid y + x \mid z$	
SC1	$x \mid y = y \mid x$	
SC2	$x \| 1 = x$	
SC3	$1 \mid x + 1 = 1$	
SC4	$(x \| y) \| z = x \| (y \| z)$	
SC5	$(x \mid y) \mid z = x \mid (y \mid z)$	
SC6	$(x \lfloor\!\lfloor y) \lfloor\!\lfloor z = x \lfloor\!\lfloor (y \| z)$	
SC7	$(x \mid y) \lfloor\!\lfloor z = x \mid (y \lfloor\!\lfloor z)$	
SC8	$x \lfloor\!\lfloor 0 = x \cdot 0$	

Appendix B

ASF+SDF Survey

General

1. How would you rate your knowledge of ASF+SDF?

Very low	Low	Medium	High	Very high
○	○	○	○	○

2. How familiar are you with the model transformation you are analyzing?

I never heard about it	I heard about it	I saw it before	I looked at it in detail	I designed it
○	○	○	○	○

3. Please indicate the time you started the questionnaire and the time you finished it.
   Starting time: _____
   End time: _____

Analysis

4. What is the amount of effort required for the comprehension of the transformation?

Very low	Low	Medium	High	Very high
○	○	○	○	○

5. To what extent is the structure of the transformation designed, such that parts belonging to the same functionality are coupled tightly and parts that do not belong to the same functionality are coupled loosely?

Very poor	Poor	Medium	Good	Very good
○	○	○	○	○

6. To what degree is the programming style in the transformation uniform?

Very low	Low	Medium	High	Very high
○	○	○	○	○

7. Could the transformation be written using fewer elements (but adhering to the same specification as the given transformation)?
Think of elements as equations, variables, functions, signatures, conditions, etc.

Yes, definitely	Yes, maybe	Neutral	No, not really	No, not at all
○	○	○	○	○

8. To what extent do you think does the transformation provide the functionality it is supposed to provide?

Very low	Low	Medium	High	Very high
○	○	○	○	○

9. To what degree is the transformation (or parts of it) reusable to create other transformation?

Very poor	Poor	Medium	Good	Very good
○	○	○	○	○

10. What is the expected difficulty of performing an average modification to the transformation?

Very low	Low	Medium	High	Very high
○	○	○	○	○

11. What is your expectation about the effort needed by a software engineer to understand the transformation well enough to be able to find errors in it?

Very low	Low	Medium	High	Very high
○	○	○	○	○

12. How would you rate the conciseness of the transformation?

Very low	Low	Medium	High	Very high
○	○	○	○	○

13. To what degree does the transformation contain conflicting elements? Think of things like conflicting variable names, equations, etc.

Very low	Low	Medium	High	Very high
○	○	○	○	○

14. How useful is the transformation as a starting point for creating another (functionally related) transformation?

Not useful at all	Not useful	Neutral	Useful	Very useful
○	○	○	○	○

15. How much effort do you need on average to track the signature to which an equation is related?
Do not just try one.

Huge effort	Lots of effort	Medium effort	Little effort	Hardly any effort
○	○	○	○	○

16. How would you rate the modifiability of the transformation?

Very low	Low	Medium	High	Very high
○	○	○	○	○

17. Can you give an indication of how well the transformation adheres to its specification?
The question here is not to give this indication this, but if you could indicate it if required.

Certainly not	Probably not	Maybe	Probably	Certainly
○	○	○	○	○

18. How much effort do you need to understand the purpose of an average function?
Do not just try one.

Huge effort	Lots of effort	Medium effort	Little effort	Hardly any effort
○	○	○	○	○

19. How much effort is needed to adapt the transformation to support a change in one of the metamodels?

Huge effort	Lots of effort	Medium effort	Little effort	Hardly any effort
○	○	○	○	○

20. How well are the elements of the transformation structured in suitable modules (or "a modular manner")?

Very poor	Poor	Medium	Good	Very good
○	○	○	○	○

21. How would you rate the completeness of the transformation?

Very low	Low	Medium	High	Very high
○	○	○	○	○

22. How many elements of the source language can be transformed by the transformation?

None	Few	Average	Most	All
○	○	○	○	○

23. How would you rate the balance of the modules in terms of size and functionality?

Very poor	Poor	Medium	Good	Very good
○	○	○	○	○

24. To what degree is the transformation free of redundant or spare elements? Think of elements as variables, equations, productions, etc.

Very low	Low	Medium	High	Very high
○	○	○	○	○

25. How much effort is needed to adapt the transformation in order to meet changing requirements?

Huge effort	Lots of effort	Medium effort	Little effort	Hardly any effort
○	○	○	○	○

26. How would you rate the reusability of the transformation?

Very low	Low	Medium	High	Very high
○	○	○	○	○

27. How would you rate the consistency of the transformation?

Very low	Low	Medium	High	Very high
○	○	○	○	○

28. How do you rate the understandability of the transformation?

Very low	Low	Medium	High	Very high
○	○	○	○	○

Appendix C

ATL Survey

Description

The goal of our research is to make the quality of ATL model transformations measurable. To investigate the influences on the quality of ATL model transformations, we have to know what their perceived quality is. Therefore we would like you to participate in our experiment. We would like you, as an ATL expert, to evaluate the quality of (a number of) ATL model transformation(s) manually. The strength of the results of this experiment will benefit from the amount of model transformations that are evaluated. Filling out this survey will take approximately 10 to 15 minutes of your time.

Instructions

1. Download one of the seven ATL model transformations and its description. (Just pick one randomly.)
2. Evaluate the transformation.
3. Fill out the questionnaire provided below.

Survey

Background questions

1. Please enter your name.

   _____

2. Please enter your e-mail address.
   We might use this to contact you with regard to your answers to the open questions.

   _____

3. How would you rate your knowledge of ATL?

Very low	Low	Medium	High	Very high
◯	◯	◯	◯	◯

4. What is the number of the transformation you are evaluating?

1	2	3	4	5	6	7
◯	◯	◯	◯	◯	◯	◯

5. Please enter the name of the transformation you are evaluating.

   _____

6. How familiar are you with the model transformation you are analyzing?

I never heard about it	I heard about it	I saw it before	I looked at it in detail	I designed it
◯	◯	◯	◯	◯

Evaluation questions

7. What is the amount of effort required for the comprehension of the transformation?

very little			Average			very much
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

8. To what degree is the programming style in the transformation uniform?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

9. Could the transformation be written using fewer elements (but adhering to the same specification as the given transformation)?

No, not at all			Neutral			Yes, definitely
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

10. To what extent do you think does the transformation provide the functionality it is supposed to provide?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

11. To what degree is the transformation (or parts of it) reusable to create other transformation?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

12. What is the expected difficulty of performing an average modification to the transformation?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

13. What is your expectation about the effort needed by a software engineer to understand the transformation well enough to be able to find errors in it?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

14. How would you rate the conciseness of the transformation?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

15. To what degree does the transformation contain conflicting elements? Think of things like conflicting variable names, etc.

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

16. How useful is the transformation as a starting point for creating another (functionally related) transformation?

Not useful at all			Neutral			Very useful
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

17. How would you rate the modifiability of the transformation?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

18. Can you give an indication of how well the transformation adheres to its specification?
    The question here is not to give this indication this, but if you could indicate it if required.

    Very low — Average — Very high
    ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7

19. How much effort do you need to understand the purpose of an average rule or helper?
    Do not just try one.

    Hardly any effort — Medium effort — Huge effort
    ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7

20. How much effort is needed to adapt the transformation to support a change in one of the metamodels?

    Hardly any effort — Medium effort — Huge effort
    ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7

21. How would you rate the completeness of the transformation?

    Very low — Average — Very high
    ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7

22. How many elements of the source metamodel can be transformed by the transformation?

    none — Average — all
    ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7

23. To what degree is the transformation free of redundant or spare elements? Think of elements as variables, rules, etc.

    Very low — Average — Very high
    ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7

24. How much effort is needed to adapt the transformation in order to meet changing requirements?

    very little — Average — very much
    ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7

25. How would you rate the reusability of the transformation?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

26. How would you rate the consistency of the transformation?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

27. How would you rate the understandability of the transformation?

Very low			Average			Very high
◯ 1	◯ 2	◯ 3	◯ 4	◯ 5	◯ 6	◯ 7

Open questions

28. What affects understandability of a model transformation in your opinion (either positively or negatively)?

    _____

29. What affects modifiability of a model transformation in your opinion (either positively or negatively)?

    _____

30. What affects completeness of a model transformation in your opinion (either positively or negatively)?

    _____

31. What affects consistency of a model transformation in your opinion (either positively or negatively)?

    _____

32. What affects conciseness of a model transformation in your opinion (either positively or negatively)?

    _____

33. What affects reusability of a model transformation in your opinion (either positively or negatively)?

    _____

34. If you have any other remarks, please put them in the textbox below.

    _____

Summary

Assessing and Improving the Quality of Model Transformations

Software is pervading our society more and more and is becoming increasingly complex. At the same time, software quality demands remain at the same, high level. Model-driven engineering (MDE) is a software engineering paradigm that aims at dealing with this increasing software complexity and improving productivity and quality. Models play a pivotal role in MDE. The purpose of using models is to raise the level of abstraction at which software is developed to the level where concepts of the domain in which the software has to be applied, i.e., the target domain, can be expressed effectively. For that purpose, domain-specific languages (DSLs) are employed. A DSL is a language with a narrow focus, i.e., it is aimed at providing abstractions specific to the target domain. This makes that the applicability of models developed using DSLs is typically restricted to describing concepts existing in that target domain. Reuse of models such that they can be applied for different purposes, e.g., analysis and code generation, is one of the challenges that should be solved by applying MDE. Therefore, model transformations are typically applied to transform domain-specific models to other (equivalent) models suitable for different purposes. A model transformation is a mapping from a set of source models to a set of target models defined as a set of transformation rules.

MDE is gradually being adopted by industry. Since MDE is becoming more and more important, model transformations are becoming more prominent as well. Model transformations are in many ways similar to traditional software artifacts. Therefore, they need to adhere to similar quality standards as well. The central research question discoursed in this thesis is therefore as follows. *How can the quality of model transformations be assessed and improved, in particular with respect to development and maintenance?*

Recall that model transformations facilitate reuse of models in a software development process. We have developed a model transformation that enables reuse of analysis models for code generation. The semantic domains of the source and target language of this model transformation are so far apart that straightforward transformation is impossible, i.e., a semantic gap has to be bridged. To deal with model transformations that have to bridge a semantic gap, the semantics of the source and target language, as well as possible additional requirements should be well understood. When bridging a semantic gap is not straightforward, we recommend to address a simplified version of the source metamodel first. Finally, the requirements on the transformation may, if possible, be relaxed to enable automated model transformation.

Model transformations that need to transform between models in different semantic domains are expected to be more complex than those that merely transform syntax. The complexity of a model transformation has consequences for its quality. Quality, in general, is a subjective concept. Therefore, quality can be defined in different ways. We defined it in the context of model transformation. A model transformation can either be considered as a transformation definition or as the process of transforming a source model to a target model. Accordingly, model transformation quality can be defined in two different ways. The quality of the definition is referred to as its internal quality. The quality of the process of transforming a source model to a target model is referred to as its external quality. There are also two ways to assess the quality of a model transformation (both internal and external). It can be assessed directly, i.e., by performing measurements on the transformation definition, or indirectly, i.e., by performing measurements in the environment of the model transformation. We mainly focused on direct assessment of internal quality. However, we also addressed external quality and indirect assessment.

Given this definition of quality in the context of model transformations, techniques can be developed to assess it. Software metrics have been proposed for measuring various kinds of software artifacts. However, hardly any research has been performed on applying metrics for assessing the quality of model transformations. For four model transformation languages with different characteristics, viz., for ASF+SDF, ATL, Xtend, and QVTO, we defined sets of metrics for measuring model transformations developed with these languages. While these metric sets can be used to indicate bad smells in the code of model transformations, they cannot be used for assessing quality yet. A relation has to be established between the metric sets and attributes of model transformation quality. For two of the aforementioned metric sets, viz., the ones for ASF+SDF and for ATL, we conducted an empirical study aimed at establishing such a relation. From these empirical studies we learned what metrics serve as predictors for different quality attributes of model transformations.

Metrics can be used to quickly acquire insights into the characteristics of a model transformation. These insights enable increasing the overall quality of model transformations and thereby also their maintainability. To support maintenance, and also development in a traditional software engineering process, visualization techniques are often employed. For model transformations this appears as a feasible approach as well. Currently, however, there are few visualization techniques available tailored towards analyzing model transformations. One of the most time-consuming processes during software maintenance is acquiring understanding of the software. We expect that this holds for model transformations as well. Therefore, we presented four complementary visualization techniques for facilitating model transformation comprehension. The first two techniques are aimed at visualizing the dependencies between the components of a model transformation. The other two techniques are aimed at visualizing the coverage of the source and target metamodels by a model transformation.

The development of the metric sets, and in particular the empirical studies, have led to insights considering the development of model transformations. Also, the proposed visualization techniques are aimed at facilitating the development of model transformations. We applied the insights acquired from the development of the metric sets as well as the visualization techniques during the development of a chain of model transformations that bridges a number of semantic gaps. We chose to solve this transformational problem not with one model transformation, but with a number of smaller model transformations. This should lead to smaller transformations, which are more understandable. The language on which the model transformations are defined, was subject to evolution. In particular the coverage visualization proved to be beneficial for facilitating the co-evolution of the model transformations.

Summarizing, we defined quality in the context of model transformations and addressed the necessity for a methodology to assess it. Therefore, we defined metric sets and performed empirical studies to validate whether they serve as predictors for model transformation quality. We also proposed a number of visualizations to increase model transformation comprehension. The acquired insights from developing the metric sets and the empirical studies, as well as the visualization tools, proved to be beneficial for developing model transformations.

Samenvatting

Assessing and Improving the Quality of Model Transformations

Software dringt meer en meer door in onze samenleving en wordt steeds complexer. Tegelijkertijd blijven de eisen aan de kwaliteit van software op eenzelfde, hoog niveau. Modelgedreven softwareontwikkeling is een softwareontwikkelingparadigma dat is gericht op het omgaan met de toenemende complexiteit van software en op het verhogen van de kwaliteit ervan. Modellen spelen een sleutelrol binnen dit paradigma. Het doel van het gebruik van modellen is het verhogen van het abstractieniveau waarop software wordt ontwikkeld tot het niveau waar concepten van het domein waarin de software toegepast dient te worden, het doeldomein, effectief uitgedrukt kunnen worden. Om dit doel te bereiken worden domeinspecifieke talen gebruikt. Een domeinspecifieke taal is een taal met een beperkte focus, het is bedoeld om abstracties te verschaffen die specifiek zijn voor het doeldomein. Dit maakt dat de toepassing van modellen ontwikkeld met behulp van een domeinspecifieke taal zich meestal beperkt tot het beschrijven van concepten in het doeldomein. Hergebruik van modellen zodat deze kunnen worden gebruikt voor andere doeleinden, bijvoorbeeld analyse of codegeneratie, is een van de uitdagingen die opgelost dient te worden door het toepassen van modelgedreven softwareontwikkeling. Daarom worden typisch modeltransformaties toegepast om domeinspecifieke modellen te transformeren naar andere (equivalente) modellen die geschikt zijn voor andere doeleinden. Een modeltransformatie is een afbeelding van een verzameling bronmodellen op een verzameling doelmodellen gedefinieerd als een verzameling van transformatieregels.

Modelgedreven softwareontwikkeling wordt geleidelijk opgepakt door de industrie. Omdat modelgedreven softwareontwikkeling steeds belangrijker worden, krijgen ook modeltransformaties een prominentere rol. Modeltransformaties zijn in vele opzichten vergelijkbaar met traditionele softwareartefacten. Daarom dienen ze

ook aan dezelfde kwaliteitseisen te voldoen. De centrale onderzoeksvraag die in dit proefschrift is verhandeld is daarom als volgt. *Hoe kan de kwaliteit van modeltransformaties worden vastgesteld en verhoogd, met name met betrekking tot ontwikkeling en onderhoud?*

Zoals reeds vermeld, faciliteren modeltransformaties hergebruik van modellen in een softwareontwikkelproces. We hebben een modeltransformatie ontwikkeld die hergebruik van analysemodellen voor codegeneratie bewerkstelligt. De semantische domeinen van de bron- en doeltaal van deze modeltransformaties liggen dusdanig ver uit elkaar dat een rechtstreekse transformatie onmogelijk is, met andere woorden er moet een semantisch gat overbrugd worden. Om om te gaan met modeltransformaties die een semantisch gat moeten overbruggen, dienen zowel de semantiek van de bron- en doeltaal van de transformatie als eventuele andere vereisten aan de transformatie grondig begrepen te zijn. Als het overbruggen van een semantisch gat lastig blijkt te zijn, raden we aan te beginnen met een vereenvoudigde versie van de brontaal. Als dit niet afdoende is, kan er voor gekozen worden om, indien mogelijk, de eisen aan de transformatie te verzwakken om zo toch tot een automatische model transformatie te komen.

Het is te verwachten dat modeltransformaties die een semantisch gat moeten overbruggen complexer zijn dan modeltransformaties die slechts syntaxis transformeren. De complexiteit van een modeltransformatie heeft consequenties voor de kwaliteit ervan. Kwaliteit in het algemeen is een subjectief concept. Daarom kan het op verschillende manieren gedefinieerd worden. Wij hebben het gedefinieerd in de context van modeltransformaties. Een modeltransformatie kan beschouwd worden als een transformatiedefinitie of als het proces volgens welke een bronmodel in een doelmodel getransformeerd wordt. Overeenkomstig kan de kwaliteit van modeltransformaties op twee verschillende manieren gedefinieerd worden. We verwijzen naar de kwaliteit van de definitie als de interne kwaliteit van een modeltransformatie. We verwijzen naar de kwaliteit van het proces volgens welke een bronmodel in een doelmodel getransformeerd wordt als de externe kwaliteit van een modeltransformatie. Er zijn ook twee manieren om de kwaliteit van een modeltransformatie vast te stellen (zowel intern als extern). Het kan direct vastgesteld worden, door te meten aan de transformatiedefinitie, of indirect, door te meten in de omgeving van een modeltransformatie. We hebben ons met name gericht op het direct vaststellen van interne kwaliteit, hoewel we ook externe kwaliteit en indirecte vaststelling van kwaliteit hebben besproken.

Gegeven deze definitie van kwaliteit in de context van modeltransformaties kunnen technieken worden ontwikkeld om deze kwaliteit vast te stellen. Softwaremetrieken zijn voorgesteld voor het meten aan vele verschillende soorten softwareartefacten. Echter, er is nauwelijks onderzoek gedaan naar het toepassen van metrieken voor het vaststellen van de kwaliteit van modeltransformaties. Voor vier model-

transformatietalen met verschillende karakteristieken, namelijk voor ASF+SDF, ATL, Xtend en QVTO, hebben we verzamelingen metrieken gedefinieerd om te meten aan modeltransformaties die ontwikkeld zijn gebruikmakend van deze talen. Hoewel deze verzamelingen metrieken gebruikt kunnen worden om zogenaamde 'bad smells' in de broncode van modeltransformaties te detecteren kunnen ze nog niet direct gebruikt worden voor het vaststellen van kwaliteit. Daarvoor dient een relatie gelegd te worden tussen de metrieken en diverse attributen van modeltransformatiekwaliteit. Voor twee van de voorgenoemde verzamelingen van metrieken, namelijk die voor ASF+SDF en voor ATL, hebben we een empirisch onderzoek uitgevoerd met als doel deze relatie vast te stellen. Van deze empirische onderzoeken hebben we geleerd welke metrieken gebruikt kunnen worden als voorspellers voor verschillende kwaliteitsattributen voor modeltransformaties.

Metrieken kunnen gebruikt worden om snel inzicht te verkrijgen in de karakteristieken van een modeltransformatie. Deze inzichten zorgen ervoor dat de algehele kwaliteit van modeltransformaties en daarmee ook hun onderhoudbaarheid verhoogd kan worden. Om onderhoud en ook ontwikkeling in een traditioneel softwareontwikkelingsproces te ondersteunen worden vaak visualisatietechnieken gebruikt. Dit lijkt ook voor modeltransformaties een valide aanpak. Op het moment zijn er echter weinig visualisatietechnieken beschikbaar die zijn gericht op het analyseren van modeltransformaties. Een van de meest tijdrovende processen in het onderhoudstraject van software is het begrijpen van de software. We verwachten dat dit ook geldt voor modeltransformaties. Daarom hebben we vier complementaire visualisatietechnieken gepresenteerd voor het faciliteren van het verkrijgen van begrip van een modeltransformatie. De eerste twee technieken zijn gericht op het visualiseren van afhankelijkheden tussen de componenten van een modeltransformatie. De andere twee technieken zijn gericht op het analyseren van de 'coverage' van het bron- en doelmetamodel door een modeltransformatie.

Het ontwikkelen van de metriekenverzamelingen en met name de empirische onderzoeken hebben geleidt tot inzichten aangaande de ontwikkeling van modeltransformaties. Ook zijn de visualisatietechnieken gericht op het faciliteren van de ontwikkeling van modeltransformaties. We hebben de inzichten verkregen van het ontwikkelen van de metriekenverzamelingen, evenals de visualisatietechnieken toegepast tijdens de ontwikkeling van een keten van modeltransformaties die een aantal semantisch gaten moeten overbruggen. We hebben er voor gekozen om dit transformatieprobleem niet op te lossen met een modeltransformatie, maar met een aantal kleinere modeltransformaties. Dit zou moeten leiden tot kleinere modeltransformaties, welke beter begrijpbaar zijn. De taal waarop deze modeltransformaties zijn gedefinieerd was aan evolutie onderhevig. Met name de visualisatie van metamodelcoverage heeft bewezen erg nuttig te zijn voor het faciliteren van het proces van co-evolutie van de modeltransformaties.

Samenvattend, we hebben kwaliteit in de context van modeltransformaties gedefinieerd en we hebben de noodzaak voor een methodologie om deze vast te stellen besproken. Daarom hebben we verzamelingen metrieken gedefinieerd en empirische onderzoeken uitgevoerd om te valideren of de metrieken bruikbaar zijn als voorspellers van de kwaliteit van modeltransformaties. We hebben ook een aantal visualisatietechnieken voorgesteld om het begrip van modeltransformaties te vergroten. De verkregen inzichten van het ontwikkelen van de metriekenverzamelingen, evenals de visualisatietechnieken hebben bewezen nuttig te zijn voor het ontwikkelen van modeltransformaties.

Curriculum Vitae

Personal Information

Marinus Franciscus (Marcel) van Amstel
Date of birth: September 28, 1981
Place of birth: Tilburg, The Netherlands

Education

Computer Science and Engineering (Master of Science) *2000–2006*
Eindhoven University of Technology, Eindhoven
Bachelor degree obtained in 2005
Master degree obtained in 2006

Certificate Technological Management *2004–2005*
Eindhoven University of Technology, Eindhoven
Certificate obtained in 2005

Atheneum *1993–2000*
Koning Willem II College, Tilburg
Diploma obtained in 2000

Professional Experience

Eindhoven University of Technology *2006–2011*
PhD. candidate

LaQuSo (Laboratory for Quality Software) *2010–2010*
Junior researcher

IPA Dissertation Series

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environ-*

ments. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of

Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science,

Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electri-

cal Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code*

Generation with Templates. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model coevolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19