Towards Correct Modelling and Model Transformation in DPF

Xiaoliang Wang

Dissertation for the degree of philosophiae doctor (PhD)
at the University of Bergen

2016

Dissertation date: June 14th.

University of Bergen, Norway
Submitted March 14th, 2016

To my daughter Xiangyi and my wife Weiping

Contents

Acknowledgements

I should express my sincere acknowledgements to those nice and brilliant people who have helped me go through the journey.

Firstly, I would like to express my sincere gratitude to my supervisor Yngve Lamo for his continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study. I would also like to thank my co-supervisor Adrian Rutle for his tremendous help in the improvement of my English writing. These discussions about research and life not only inspired my to find solutions in different directions, but also improved my ability to express my thoughts clearly. Moreover, I also want to pay my appreciate to my co-supervisor Uwe Wolter for his support from University of Bergen and valuable feedback from research; for teaching me mathematics and different strategies to understand knowledge; for sharing his marvellous experiences and stories.

I am grateful to my wife Weiping and my daughter Xiangyi, for their unconditional love and accompany. They deserve the credit for the accomplishment of this thesis. In addition, I also thank my parents and sisters. It is their patience and love that encourage me to reach this place. I cannot walk so far without their support.

I would like to thank my opponents Zhiming Liu, Arnaud Gotlieb and Magne Haveraaen, for the time they have spent reviewing this work. I am also grateful to all my fellow researchers and anonymous reviewers who pointed out flaws and suggested possible improvements in my research.

I also want to say thanks to Department of Computing, Mathematics and Physics in Bergen University College. It has given me a nice working environment. The nice colleagues also give great help to me when necessary. I would also like to say thanks to Lars Kristensen, Carsten Helgesen, Kristin Hetland, Volker Stolz, Sven-Olai Høyland, Helga Jonsdottir, Bjarte Kileng. Moreover, I want to thank my friends in office: Florian Mantz, Ajith Kumar, Yi Wang, Helge Erdal, Erik Eikeland, Bin Wu and Fernando

Macias. Thank you for hanging out and having lunch discussion during the last several years.

Finally, I want to thank my friends in Bergen: Tao Yan, Yunpeng Ding, Yifan Zhou, Huaiyang Zhu, Ning Yang, Yu Hong and Zuojun Xiong. It was a wonderful time to live at Haukeland and hang out with you guys. Also special thanks to Xiaoxi Xu and Yunsheng Yang who have helped me a lot during the first several years staying in Bergen.

Scientific Environment

The research presented in this thesis has been accomplished at the Department of Computing, Mathematics and Physics at Bergen University College, in cooperation with the Department of Informatics at University of Bergen.

HØGSKOLEN
I BERGEN
BERGEN UNIVERSITY COLLEGE

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

Abstract

Model-driven engineering (MDE) is a model-centric software development methodology. It promotes models as first-class entities in software development. Models are used to represent software along software development process and to finally generate software automatically by model transformations. Thus, the quality of software is highly dependent on models and model transformations. This thesis devotes to construct correct models and model transformations in Diagram Predicate Framework (DPF), which provides a formal diagrammatic approach to (meta)modelling and model transformation based on category theory.

The thesis presents the *DPF Model Editor*, a graphical (meta)modelling editor for DPF which supports diagrammatic (meta)modelling. In addition, we propose bounded verification approaches of models and model transformations respectively by using Alloy. Alloy consists of a modelling language and the Alloy Analyzer to examine Alloy specifications. The verification approaches proposed in the thesis translate models and model transformations into Alloy specifications, which are passed to the Alloy Analyzer to verify whether the models and model transformations satisfy some desired properties.

Because of the inherent limitation of Alloy, the verification approaches also encounter a scalability problem: it may take quite long time or become intractable to verify larger models or complex model transformations. To tackle the problem, the thesis also presents several techniques to optimize the approaches. The first technique splits models into submodels and reduces the verification of the models into the verification of some submodels. The other two techniques are proposed for the verification of model transformations: one technique uses a modelling approach to simplify model transformations; while the other one optimizes the translation of model transformation rules. Experimental results show that these techniques alleviate the scalability problem.

List of Publications

This thesis is based on a sequence of publications and organized into two parts. The first part introduces the background information and presents the state-of-the-art of verification in MDE. Then we summarize the publications to give an overview of the contribution of the thesis. At the end of the part, we conclude the thesis and discuss our future work. In the second part, the publications are list.

Paper A
: Yngve Lamo, Xiaoliang Wang, Florian Mantz, Wendy MacCaull, and Adrian Rutle. DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment. In *Computer and Information Science*, volume 429 of *SCI*, pages 37–52. Springer, 2012

Paper B
: Xiaoliang Wang, Adrian Rutle, and Yngve Lamo. Towards User-Friendly and Efficient Analysis with Alloy. volume 1514 of *CEUR Workshop Proceedings*, pages 28–37, 2015

Paper C
: Xiaoliang Wang, Fabian Büttner, and Yngve Lamo. Verification of Graph-based Model Transformations Using Alloy. *ECEASST*, 67, 2014

Paper D
: Xiaoliang Wang, Adrian Rutle, and Yngve Lamo. Scalable Verification of Model Transformations. In *Proceedings of the 11th Workshop on Model-Driven Engineering, Verification and Validation co-located with 17th International Conference on Model Driven Engineering Languages and Systems*, volume 1235 of *CEUR Workshop Proceedings*, pages 29–38. CEUR-WS.org, 2014

Paper E
: Xiaoliang Wang and Adrian Rutle. Model Checking Healthcare Workflows Using Alloy. In *MMHS*, volume 37 of *Procedia Computer Science*, pages 481–488. Elsevier, 2014

Paper A is a joint work among several authors. My contribution to this paper is introducing and implementing the `Signature Editor`, and developing the example presented in the paper. Paper B is co-authored with Yngve Lamo and Adrian Rutle. I am the main author of the papers, being supervised by them. My contribution to the paper is introducing the transformation between DPF models and Alloy specifications; implementing and integrating the verification into the DPF modelling tool; initializing and validating the splitting technique; designing the example, performing the experiment and analysing the result. Paper C is co-authored with Yngve Lame and Fabian Büttner. I am the main author of the paper, being supervised by Yngve Lamo. Fabian Büttner suggested using Alloy to perform verification. My contribution to the paper is introducing and implementing the transformation from model transformation systems to Alloy specifications; initializing the idea of verifying property by checking the two conditions; designing the example and performing the experiment. Paper D is co-authored with Yngve Lamo and Adrian Rutle. I am the main author of the papers, being supervised by them. Yngve and Adrian proposed the annotation technique while I initialized the splitting technique. My contribution of the paper also included implementing the two techniques; performing the experiments and analysing the result. Paper E is co-authored with Adrian Rutle. I am the main author of the paper, being supervised by him. Adrian proposed to verify workflow models by using the verification approach in Paper C. My contribution to the paper is introducing and implementing the transformation from workflow models to Alloy specifications; designing the workflow example and performing the experiments to verify properties of the example.

Part I

Overview

CHAPTER 1

Introduction

In this chapter, we will present the background information that is needed for the thesis. That is to give a brief introduction to Model-Driven Engineering (MDE), a software development methodology, and its main concepts: models and model transformation.

1.1 Model-Driven Engineering (MDE)

Software researchers and developers have made their efforts continuously to construct reliable software faster and more efficiently. They promoted the development of faster, cheaper and easier programming languages from machine code to assembly language, then to the 3rd generation languages[1], e.g., Java [6], C++ [7] etc. In addition, they also upgraded the underlying computing environment, e.g., from earlier CPU, to operating systems, then to application frameworks (e.g., J2EE [8],.NET [9], and CORBA [10]), to decrease programming complexities. These techniques have increased the productivity of programming by raising the abstraction level of programming languages and platforms.

However, these abstractions mainly occurred in the solution domain, i.e., the computing techniques which are used to construct a software, rather than in the problem domain, i.e., the application fields where the software will be used (e.g., web and mobile applications, financial services). Therefore, the present mainstream development methodologies, e.g., waterfall [11], Rapid Application Development (RAD) [12], and Agile software development [13], are all *code-centric* where programmers represent concepts in the problem domain as elements in the solution domain.

[1] Machine code and assembly language are called the 1st and the 2nd generation languages respectively

Software development using these methodologies is associated with several problems. Firstly, software developers spend considerable time on mapping the concepts from the problem domain to the elements in the solution domain [14]. Thus, they cannot focus on the requirements of the problem domain. This may cause that they cannot fully and correctly understand the requirements of the problem domain. Secondly, it may cause misunderstandings among users, designers and developers because users and designers may not be familiar with languages in the solution domain. As a consequence, it may lead to the fact that the software constructed does not satisfy the desired requirements of the problem domain. Lastly, along with the growing complexity of software and platforms [15], the developers need to spend long time on techniques, e.g., to migrate software to a new version or a different platform, or to learn new Application Programming Interfaces (API)s or new features to program properly. This hinders the productivity of software development [14].

Model-Driven Engineering (MDE) [14, 16], also referred as model-driven development (MDD) or model-driven software development (MDSD) in the literature [17], is an endeavor to tackle these problems by separating the problem domain and the solution domain during software development. The methodology is *model-centric*, i.e., models are the first-class entities in software development. Software designers use models to describe the structure, behavior and requirements of the problem domain. Afterwards, software can be (partially) derived from the models by automatic execution of model transformations which map concepts in the problem domain to elements in the solution domain.

MDE has been promoted in last decades [16]. Many industrial standards implementing MDE have emerged during its development, such as model-driven architecture (MDA). MDA is initiated by the Object Management Group (OMG) [18] since 2001 [19, 20, 21, 22]. It aims to provide a set of guidelines for the structuring of models [23]. A main MDA standard is the Unified Modelling Language (UML) [24], a general-purpose modelling language. It intends to provide a standard language to visualize the design of systems. In addition, two other standards, Meta-Object Facility (MOF) [25] and XML Metadata Interchanges (XMI) [26], are used to specify type systems and store models which can be expressed in MOF. Eclipse Modelling Framework (EMF) [27] is an existing implementation of the MDA standards. It is an Eclipse [27]-based modelling framework and provides code generation facility for building tools from structural models which are specified based on MOF. Moreover, OMG provides also a model transformation standard Query/View/Transformation (QVT) [28].

MDE offers many advantages over code-centric methodologies [14]. On one hand, software designers can focus on the problem domain and easily express their design intentions in domain specific languages. This also facilitates efficient communication among users, designers and developers, thus software designers can correctly and easily identify the requirements of the problem domain. On the other hand, the automatic model transformations relieve software programmers from tedious coding and construct software automatically with higher productivity and higher quality. In addition, model transformations make it easier to maintain or deploy software. For example, when the requirements of the problem domain change, software developers can adjust models, and then regenerate software by using model transformations. Similarly, when software shall be deployed onto a new environment, software developers can construct a corresponding model transformation which generates a new version of software which is compatible to the new environment.

Additional benefits with MDE are their reliability and quality of software, e.g., software fulfills the requirements of the problem domain, and whether software is free of bugs. This is normally ensured by verification of programs using either some informal approach, e.g., testing, or formal techniques, e.g., abstract static analysis, model checking [29], etc. However, testing usually involves manual construction of test cases. Even though some researchers are working on automatic generation of test cases [30], the testing results can still not ensure the absence of bugs. Formal verifications can avoid this problem, but usually require manual construction of a high-level model of source programs [31][2]. Nevertheless, such verifications have scalability problems because it involves the complexity of the problem domain which is caused by the concepts and relationships between them, and the complexity of the solution domain which is caused by the structures, programming language and related techniques [33, 34]. This restricts which systems that can be efficiently verified [35].

In MDE, since software is generated from models by model transformations, the reliability and quality of software can be ensured by the verification of models and model transformations. Such verification offers two advantages over the verification of software. Firstly, because models are specified in the problem domain, the verification of models avoids the complexity of the solution domain. Therefore, the complexity of verification of models is significantly reduced in contrast to verification of code. Secondly, the verification of models can be performed before or without implementation. The software designers can find design mistakes early in the modelling phase. This will help in building better software at a lower cost. In this thesis, we will study verification of models and model transformations. Before delving into the main topic, we firstly present the background information of the thesis; then we will introduce models and model transformations in MDE.

[2] Some works, e.g., [32], applied verification techniques directly on code.

1.2 Model

The term *model* can be interpreted distinctively within different contexts. The general meaning of model, as in dictionary [36], is "a representation of something, either as a physical object which is usually smaller than the real object, or as a simple description of the object which might be used in calculations". In formal method, a system is usually specified as a specification using formal logics, e.g., FOL (First Order Logic – also known as predicate logic). In mathematical languages, a specification is a logical formula with a set of variables in a logical language. Within this context, a model of the specification means an interpretation of the variables where the formula is evaluated to *true*. In software engineering, a model denotes "an abstraction of a (real or language-based) system allowing predictions or inferences to be made" [37]. In this thesis, we use the term model with the same meaning as in software engineering. Note that, the term model here corresponds to the term specification in formal method.

In software engineering, systems contain a collection of elements which are related and interact with each other [38]. The interactions among these elements result in changes of the systems, e.g., adding or deleting of some elements, or the change of relations among these elements. Given a system, its state is the snapshot of the system at a given moment of time, i.e., all the elements which are contained at that time and the relations among them. It represents the system before or after an interaction.

Example 1 (Human Creation System) *The Lord creates Adam first and then create Eve to accompany Adam.*

Let us take the human creation system as an example. The system contains the elements: the Lord, Adam and Eve, and the relations: Adam is the husband of Eve and Eve is the wife of Adam. In addition, the system evolves; at the beginning, it has only the Lord; then Adam and Eve are added successively as a result of creation.

```java
class Person{
    public Person wife, husband;
    public static void main(String[] ps){
        Person Adam=new Person();
        Person Eve=new Person();
        Adam.wife=Eve;
        Eve.husband = Adam;
    }
}
```

Listing 1.1: A program in Java

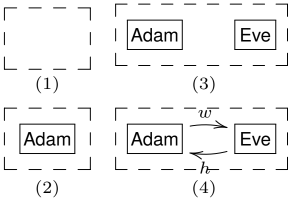


Figure 1.1: State changes

Models are abstractions of systems. According to [39], it means that, depending on the usage of models, only the relevant elements and relations are projected into models while the other irrelevant ones are just omitted. For example, to describe the human creation system, we only care about whom are created but dismiss who create them and how they are created. The system can be represented as the Java program in Listing 1.1 or the 4 models in Figure 1.1. The Java program constructs Adam and Eve, and then they become a couple. The 4 models in dashed lines depict the 4 corresponding states of the system. Elements are depicted as nodes, e.g., `Adam`, while relations are depicted as arrows between the nodes.

Software systems are usually complex [33]; they contain hardware, software, people, facilities and processes, which collaborate with each other [38]. It is difficult or impossible to project all the relevant information into one model [40]. In addition, software development is a process which consists of a sequence of phases, e.g., requirement analysis, design etc. Each phase has its own objective. For example, models in the requirement analysis phase identify the structure, behavior and requirements from the problem domain; models in the design phase represent the architecture, including the structural and functional features of the software to be built. Thus, various models are used to represent different aspects of software systems throughout the development process.

According to the relationship of models and systems, two different kinds of models, *token models* and *typed models*, are distinguished [37]. Elements and relations in token models capture singular aspects of the elements and the relations in systems. It means that all the relevant elements and relations of systems are represented one-to-one as elements in token models. Token models can be used to represent the states of systems. For example, each figure in Figure 1.1 is a token model for the human creation system since each person contained by the system at a time has a corresponding node in the model. In comparison, type models capture universal aspects of a system's elements and relations by classification. It means that the elements and relations of systems are classified into concepts and relationships, respectively, of type models; these concepts and relationships represent elements and relations which are classified as equal with respect to certain properties. For example, in the Java program, Adam and Eve are classified as `class Person` while the relations between them as the fields `wife` and `husband`. Since type models represent systems in a many-to-one way, they are more concise compared to the one-to-one representation in token models. Moreover, with this concise form, model designers can focus on general properties of concepts rather than individual objects. In MDE, most models are typed models [37]. Hereafter, the models mentioned in this thesis are typed models.

In software engineering, modelling, i.e., to design a model representing a system, can be textual, diagrammatic or hybrid. Textual modelling is to design models with text, e.g., the human creation system can be modelled as a program in Java. There are some textual modelling languages, e.g., Extensible Markup Language (XML) [41], Alloy [42], which are oriented to design models. Some other textual modelling approaches add annotations in a program which is written in a specific programming language, e.g., Java Modelling Language (JML) [43] in Java, Spec# [44] and.NET contracts [45] in C#. In contrast, diagrammatic modelling approaches represents models with visualization as graphs or graph-based structures, called *diagrammatic models*. Diagrammatic modelling has already been widespread used in software engineering for decades. Flowcharts in the 70s were used to describe behavioral aspects of software systems; Petri nets in the 80s were used to represent discrete distributed systems; Entity-Relation (ER) diagrams [46] in 80s gained popularity as the conceptual representation of data structures; In the 90s, UML diagrams became the de facto standard to represent structural and behavioral aspects of software systems. There are fundamental practical differences between textual modelling and diagrammatic modelling [47]. But both approaches have their advantages and limitations. Petre [48] pointed out that diagrammatic modelling won over textual modelling because it provided richer information, intuitive representation of complex structure, direct mapping to domain elements, accessibility and comprehensibility, fitness to human visual system and a higher level of abstraction. But she also emphasized that textual modelling had advantages when considering clarity, the quality of annotation and recognition. Thus, some researchers proposed to use both approaches [49, 50]. Since most models specified in text can be represented as an abstract syntax tree (AST) which can be viewed as a diagrammatic model, in this thesis, we focus on diagrammatic models without loss of generality.

Several diagrammatic modelling languages have appeared during the last decades, e.g., Business Process Modelling Notation (BPMN) [51] for process modelling, Architecture description language (ADL) [52] for system architecture modelling, and the ones mentioned earlier. Within these languages, UML became the de-facto standard and state-of-the-art language in MDE. UML [53] is a general-purpose modelling language which consists of 8 different diagrams, e.g., *Class Diagrams*, *Activity Diagrams*, *Object Diagram*[3] etc. Each diagram is oriented to describe an aspect of a software system. A class diagram describes a system by representing the concepts and relationships among these concepts which are involved in the system; while object diagrams are used to represent the states of a system. Since software systems could be inherently complex, modelling them involves in most cases description of two main aspects: *Structure* and *Constraints*. In the following paragraphs, we will present an example in UML to explain these two aspects.

1.2.1 Structure

Example 2 (A Civil Status System) *A civil status system describes the marital relations between persons. The system should satisfy the following requirements:*

[3] Note that Object Diagrams are excluded since UML 2.4

1. *A person has at most one wife or husband*
2. *If a person A has another person B as his wife, then B should have A as her husband*
3. *A person cannot have him/herself as wife or husband*

Recall that models contain concepts and relationships which denote elements and relations of corresponding systems. For instance, from the description as in the Example 2, we can identify one concept, *Person*, which denotes all the persons in the system. In addition, two relationships, *wife* and *husband*, are used to denote the wife and the husband relations between two persons in the system.

Given a model, its concepts and relationships can be described as a graph-based structure. In UML, class diagrams can be used to describe such structures. A class diagram is depicted as a graph where nodes represent classes and edges represent associations between these classes. The classes and associations represent concepts and relationships in a model. For example, we specify a model to describe the civil status system in the Example 2. Its structure is depicted as the class diagram in Figure 1.2. The node **Person** denotes persons. There is only one reflexive edge which connects **Person** to itself. Each end of the edge are labelled with *wife* and *husband*. The edges can be read as "a person may have another person as wife" and "a person may have another person as husband". The edge, as well as the two labels, which is called bidirectional association in UML, denotes relations between persons. Notice the two $0..1$ on the two edges. They are the constraints which we will discuss in the next section.

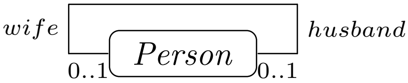


Figure 1.2: A civil status system in UML

1.2.2 Constraints

In addition to elements and relations, a system has some requirements which restrict its elements and relations. For example, the civil status system in Example 2 has three requirements. The structure in Figure 1.2 only denotes the elements and relations of the civil status system, but not such requirements. Usually, models contain constraints which are used to specify these requirements. In UML, some simple requirements, such as cardinality restrictions on relationships, can be specified as structural constraints, e.g. multiplicity constraints directly on the structure. For example, the requirement 1 can be specified as the multiplicity constraints $0..1$ on both ends of the edge in Figure 1.2. However, the expressiveness of structural constraints are quite limited, some requirements, e.g., the requirement 2 and 3, cannot be expressed with structural constraints. Thus, additional constraint languages are needed to specify these requirements as propositions on structures.

One popular additional constraint language is the Object Constraint Language (OCL) [54]. It was firstly initiated by IBM, then gained popularity in industry and became the standard constraint language used with UML. OCL is a typed specification language which expresses query or specifies invariants over objects in a model [54]. Following the terminology in [55], the constraints specified in additional constraint languages are hereafter called additional constraints. In the civil status system, the requirement 2 and 3 can be expressed as the following invariants on the class **Person**. The invariant on Line 2 states that, if a person has a wife (husband), the person is the husband (wife) of his (her) wife (husband); while the invariant on Line 3 states that the wife or the husband of a person is not the person herself/himself.

```
1 context Person
2 inv: self.wife <> null implies self.wife.husband=self and
3      self.husband <> null implies self.husband.wife=self
4 inv: self.wife <> self and self.husband <> self
```

Listing 1.2: Additional constraints in OCL

1.2.3 Diagram Predicate Framework (DPF)

The traditional modelling approaches, using diagrammatic modelling languages to specify structures while textual languages to define constraints, are adopted by software developers. However, there are two main problems with this solution [55]. The first problem is caused by mixing diagrammatic and textual modelling approaches. This mixture makes it challenging to update and synchronize structures and constraints. For instance, if a minor change occurs in a structure, e.g. change a name of an element or remove an element, the expressions in the corresponding constraints which refer to the element will cause a syntax error. Another problem is about the abstraction level of the adopted constraint language. OCL is a general constraint language; it is not oriented to a specific domain. To express constraints in a domain, the software designer still need to customize the language. Modelling can become complex and error-prone in this way. In addition, it is complex and difficult to reason about models at such a low level of abstraction.

To solve these problems, Diagram Predicate Framework (DPF) proposes a formal diagrammatic approach of (meta)modelling and model transformation based on category theory [56]. This framework is initialized by a joint

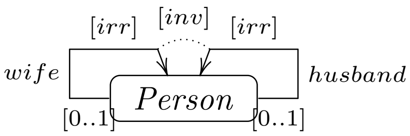


Figure 1.3: A civil status system in UML

research project between Bergen University College and University of Bergen. It is an extension of the Generalised Sketches formalism developed by Diskin et al. in [57, 58, 59]. Several researchers have further made their contribution to enrich the features of the framework. Adrian Rutle formalized the theoretical foundation of the modelling framework [55]; Alessandro Rossini focused on model versioning and deep modelling framework [60]; while Florain Mantz studied model migration [61]. Please refer to `dpf.hib.no` for more background information.

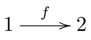
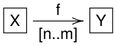
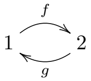
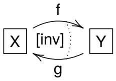
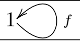
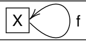
p	$\alpha^{\Sigma}(p)$	Proposed Visualization	Semantic Interpretation
`multi(n, m)`			$\forall x \in X : n \leq \lvert f(x) \rvert \leq m \wedge 0 \leq n \leq m \wedge m \geq 1$
`inverse`			$\forall x \in X, \forall y \in Y : y \in f(x)$ iff $x \in g(y)$
`irreflexive`			$\forall x \in X : x \notin f(x)$

Table 1.1: A sample signature Σ

With DPF, the structures of models are depicted as directed graphs while constraints are formulated diagrammatically on the directed graphs based on *predicate*s. For example, the civil status system can be specified as the DPF model in Figure 1.3. The structure of the model is depicted as a direct graph. The graph is similar to the one in Figure 1.2 except that, both relationships, wife and husband, are depicted as two directed arrows, instead of bidirectional associations in UML. In addition, instead of using the textual OCL invariants, five diagrammatic constraints over the graph (depicted within [] on edges and dashed lines between edges) are used to specify the requirements 1, 2 and 3. These diagrammatic constraints over the graph are formulated based on predicates `multi(n,m)`, `inverse` and `irreflexive` in Table 1.1. Each predicate has a name p, an arity $\alpha^{\Sigma}(p)$, a proposed visualization and a semantic interpretation. The arity of a predicate specifies on which kind of graphs a constraint can be formulated based on the predicate. A constraint over a structure which is formulated based on a predicate implies a graph morphism from the arity of the predicate to the structure. For example, the constraint between the edges **wife**

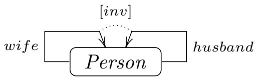
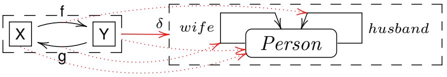
Constraint based on `inverse`	$\delta : \alpha^{\Sigma}(inverse) \to S$
	

Table 1.2: Diagrammatic constraint based on predicate

and **husband** are formulated based on the `inverse` predicate, as shown in Table 1.2. The red dashed arrows indicate the implicit graph morphism $\boldsymbol{\delta : \alpha^{\Sigma}(p) \to S}$.

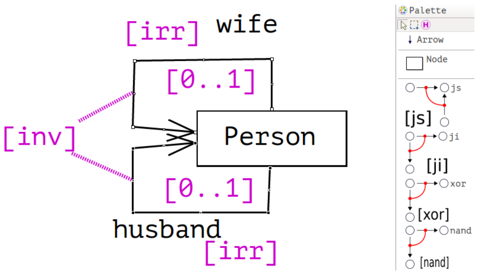
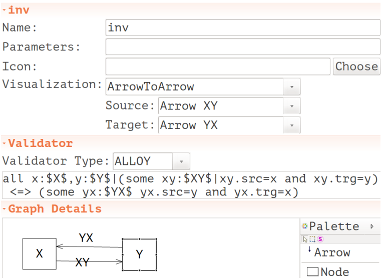


Figure 1.4: A model in the `DPF Model Editor`


- inv
Name: inv
Parameters:
Icon: Choose
Visualization: ArrowToArrow
Source: Arrow XY
Target: Arrow YX
- Validator
Validator Type: ALLOY

```
all x:$X$,y:$Y$|(some xy:$XY$|xy.src=x and xy.trg=y)
 <=> (some yx:$YX$ yx.src=y and yx.trg=x)
```

- Graph Details

Figure 1.5: `inverse` defined in the `Signature Editor`

In order to support modelling in DPF, we have implemented a DPF workbench using Eclipse modelling technologies [62]. The workbench consists of the `DPF Model Editor` for creating models. For example, the model in Figure 1.3 can be created in the editor as shown in Figure 1.4. Using the workbench, model designers can construct the structure of the model using the provided concepts, e.g., `Arrow` and `Node` in the figure; constraints are formulated on the structure by clicking on an applicable predicate; when a subgraph is selected, a list of applicable predicates will appear, i.e., the predicates from the arity of which there exists a graph morphism to the selected subgraph. Several predefined predicates, including the predicates listed in Table 1.1, are shipped with the editor. In addition, we also implemented the `Signature Editor`, a tool to specify customised predicates. Model designers can employ the tool to define their own predicates. For example, the predicate `inverse` can be specified in the tool as shown in Figure 1.5. The syntax of the predicates is specified graphically; while the semantics can be specified in different languages, e.g. Java, OCL or Alloy. Then these predicates, along with the predefined predicates, can be loaded into the `DPF Model Editor` to formulate constraints. This work is presented in Paper A.

1.3 Instance

A model $\mathfrak{S} = (S, C^{\mathfrak{S}})$ which consists of a structure S and constraints $C^{\mathfrak{S}}$ defines its instances. For diagrammatic models, each instance is a graph or graph-based structure *well-typed* by the structures of models; moreover, it also satisfies all the constraints of the models. Formally, for graph-based structures, when we say that a structure I is well-typed by another structure S, denoted as $I : S$, it means that there is a graph morphism $\iota : I \rightarrow S$. While we say that a structure I satisfies a constraint c, denoted as $I \vDash c$, it means that I satisfies c according to the semantic of c. In addition, we say that a structure I *conforms* to a model $\mathfrak{S}$, denoted as $I \vDash \mathfrak{S}$, if the structure is an instance of the model. In the following example, we illustrate instances which are depicted as *UML Object Diagram* and DPF instance respectively.

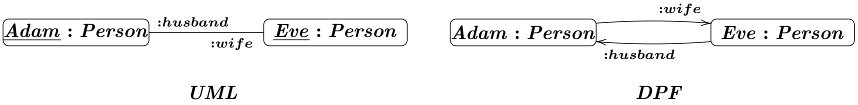


Figure 1.6: An instance of the civil status models in UML and DPF

Example 3 (Instance) *Figure 1.6 shows two instances of the models in Figure 1.2 and 1.3. The two instances are depicted as a UML object diagram and DPF instance respectively. The UML object diagram contains two objects* **Adam** *and* **Eve** *of type* **Person**, *and one link between the two objects. The link represents the relations between the two persons: Adam's wife is Eve and Eve's husband is Adam. The instance is represented similarly in DPF. The only difference is that relations are represented as directed edges.*

Problem Domain	MDE	Formal Method	Mathematics
System	Model	Specification	Formula
State	Instance	Interpretation	Model

Table 1.3: Terms correspondences among different contexts

Note that we use a model to represent a system which contains elements and their relations. In each instance, a node represents an element in the system while an edge represents that the two elements have relations. Thus, an instance establishes a one-to-one mapping of system elements. From this consideration, an instance can be viewed as a token model of the system; it can be used to represent a state of the system. Recall also that the term model in MDE corresponds to the term specification in formal logics. Thus, an instance of a model corresponds to an interpretation of a specification, i.e., a model of a formula in mathematical language. The correspondences among these contexts are shown in Table 1.3.

1.4 Metamodel

When developers build a software system, they use a programming language, e.g., Java, to write the code which is compliant with the syntax and the semantics of the language. Similarly, when designers construct a model, they also need a modelling language to design the model which is compliant with the syntax and the semantics of the language. For instance, UML class diagrams and UML object diagrams are constructed according to UML [63]. Following the "everything is a model", vision of MDE [64], a modelling language can be described by a *metamodel* at a higher level of abstraction. In other word, "a metamodel is a model of a modelling language" [65]. In addition to being a model, metamodel has its distinguished features. It captures the essential features of a language by describing its abstract syntax, concrete syntax and semantics [65]. The abstract syntax defines modelling concepts, their attributes and their relationships, as well as rules to specify valid models [24]; the concrete syntax provides a notation which is used to visualize models; the semantics interprets the meaning of the concepts and the relationships in the language. If we consider only the abstract syntax, a metamodel defines the constructors to specify a valid model. It means that models specified in a modelling language should conform to the metamodel of the language. From this view, "a model is an instance of a metamodel" [24] and each model has a metamodel. The following figure presents a simplified metamodel for UML class diagram, which is adopted from [55].

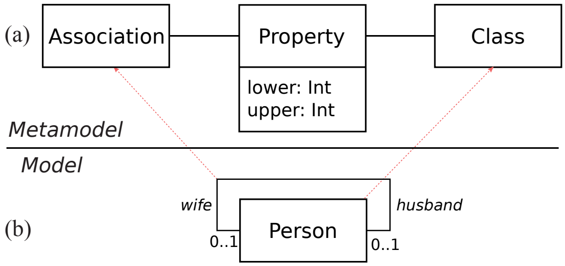


Figure 1.7: A simplified metamodel for UML class diagram

Example 4 (A simplified metamodel for UML Class Diagram) *Figure 1.7 shows a metamodel for UML Class diagram. The metamodel contains three concepts: Class, Association and Property. The concepts are used to create elements*

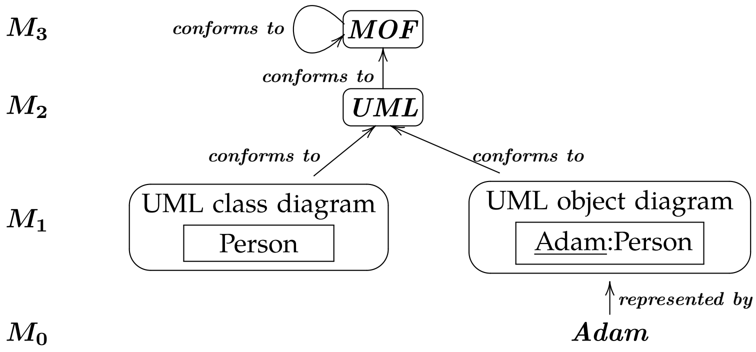
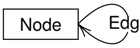


Figure 1.8: OMG's 4-layered hierarchy

in a model. Each element in a model is typed by a concept in the metamodel. The red dashed arrows are used to indicate the typing relation between the civil status model and its metamodel.

Metamodel, being a model, in turn, has its own metamodel. This pattern will repeat until a model has itself as metamodel, called *reflexive model*. Thus, for diagrammatic models, there is a modelling hierarchy where each model at a layer has the model at the layer above as its metamodel and is the metamodel of the model at the layer below. OMG envisions a 4-layered hierarchy. At the top layer, M_3 is the reflexive model MOF, which is also the metamodel of UML. At the layer blow M_2, it contains the models specified in MOF. The prominent model at this layer is UML. At the layer M_1 is the models specified in the language defined in M_2, e.g., UML. At the bottom layer M_0 are the real world objects. In Figure 1.8, we illustrate the idea of the OMG modelling hierarchy.

This modelling hierarchy has several problems [66, 67]. Firstly, it is necessary to recognize and support two classification: *linguistic*, i.e., modelling from language perspective, and *ontological*, i.e., modelling from conceptual perspective, in a modelling hierarchy [37, 68, 69]. However, this modelling hierarchy emphasize only the linguistic classification. For example, in Figure 1.9, elements on the layer M_2 are just instances of language elements on the layer M_1. Secondly, in order to specify model conceptually, a type-instance relation has to be introduced in the metamodel layer M_2. This violates the strict metamodelling doctrine. As a consequence, the multilayer hierarchy collapse into a single layer [70]. DPF tackles the issues by introducing a multi-layer modelling hierarchy, which is illustrated in Paper A. At the top of the hierarchy is the reflexive model . Models at each layer, except the top layer, conforms to a model at the layer above. With this hierarchy, in DPF, the two classifications are formalized [55]. In addition, there is no type-instance introduced in layers. Furthermore,

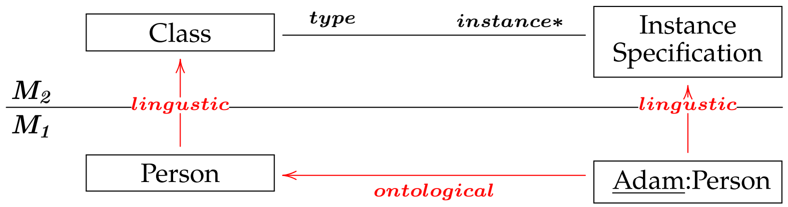


Figure 1.9: Linguistic and ontological conformance; adopted from [55]

in theory, it is possible to construct modelling hierarchies with infinite layers. This relieves the limitation that modelling hierarchies are restricted to 4 layers as in OMG. Since the topic of modelling hierarchy is beyond the scope of the thesis, please refer to [55, 65] for further information.

1.5 Model Transformation

In addition to models, the first-class entity in MDE, model transformations are also equally important. It is the heart and soul of MDE [71]. As we stated earlier, model transformations can be used to translate models into code automatically. This increases the productivity and quality of software development. Moreover, they have more applications in MDE [71]. For example, they can be used to

1. refine models along software development processes
2. optimize the structure of models while ensuring their behavior features unchanged
3. migrate software from one language or platform to another
4. integrate several models which represent different aspects of a software system into one model

Model transformations are the generation of target models from source models [21, 72]. If we view everything as a model, model transformations appear in computer science before MDE. Data in a format, e.g., arrays, can be transformed into another form, e.g., lists. In a compiling process, a lexical analyzer transforms source code in a programming language into abstract syntax trees, according to a grammar. Even compliers can also be viewed as a transformation from a higher view, since they translate source code in a higher level programming language to a lower level programming language, e.g., assembly language or machine code. These transformations are *text-to-model* or *text-to-text*. Since MDE is a model-centric methodology, transformations in MDE are mainly *model-to-text* or *model-to-model* transformation. Code generation from a model is a typical model-to-text transformation. In this thesis, we will mainly consider model-to-model transformations.

Model transformations are executed automatically by a *transformation engine*. The engine performs transformations according to a set of transformation rules which describe how one or more constructs in a source language can be transformed into one or more constructs in a target language [21, 72]. The transformation rules are specified in a *transformation language* at the metamodel layer. We depict the overview of model transformation in Figure 1.10. In the thesis, the source/target metamodels and the transformation rules are called *model transformation system*. In Figure 1.2, we presented the civil status system as a UML class diagram with bidirectional association. We will describe model-to-model transformations to translate this model into a model in DPF.

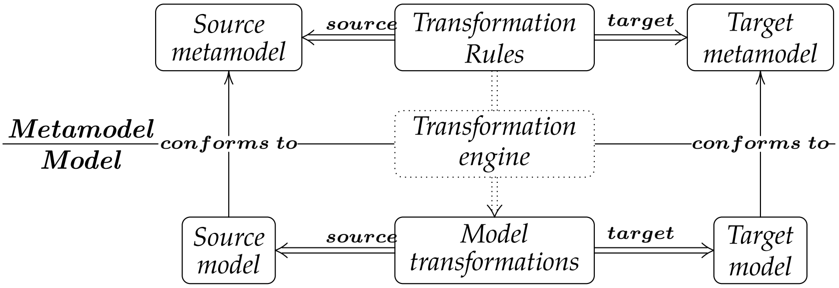


Figure 1.10: Model transformation overview

Example 5 (Transformation of UML Class Diagram to DPF model) *In Figure 1.11, we shows two transformation rules: Association-to-EReference and Class-to-Class, which are used to transform UML class diagrams to DPF models. The first rule transforms each* **Class** *in UML to a* **Node** *in DPF; while the second rule transforms each (binary)* **Association** *in UML to a pair of* **Edges**. *Both rules are denoted by blue dashed rectangle. Note that we do not present transformation rules in a specific transformation language as shown in the sequel. Here, we just present conceptually which concepts/relationships in the source metamodel are transformed into which concepts/relationships in the target metamodel. The model elements which are present in the rules, such as* **Class** *and* **Node**, *exist in the source and target metamodels, respectively. We use the reflexive model in DPF as the target metamodel.*

Given a source UML class diagram (e.g., the one in the bottom left of Figure 1.11), for each model elements of type **Class**, *e.g. the* **Person** *in red dashed*

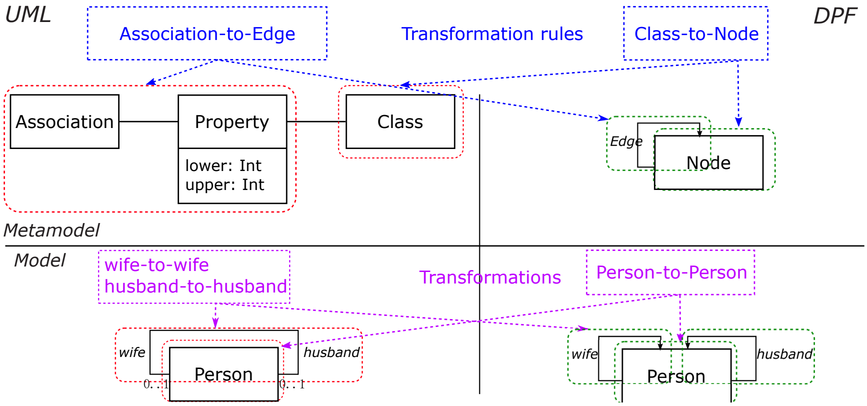


Figure 1.11: Transformation of a UML class diagram to an DPF model

rectangle, a transformation engine executes a transformation by creating a corresponding model element of type **Node***, e.g. the* **Person** *in green dashed rectangle in the target model. A similar transformation is executed by using the second rule which translates a bidirectional association in UML into two edges in DPF. Notice that, after the transformations, the target model is not consistent with the source model, since the requirements 1-3 which are specified as constraints disappear in the target model in DPF. To make the target model consistent with the source model, additional constraints should be added to explicitly specify the requirements.*

1.5.1 Classification of model transformation

According to the features listed in [73], model transformation can also be classified into different categories. Based on whether the source metamodel and the target metamodel are same, model transformations are *homogeneous* (same metamodel) and *heterogeneous* (different metamodels). Moreover, model transformations are *out-place* if the target model is created separately from the source model, or *in-place* if the target model is derived by updating the source model. This feature concerns how a transformation is performed by a transformation engine. Furthermore, model transformations could be *bidirectional* if the transformations can be performed from the source model to the target model and the opposite direction (usually for model synchronization [74]), or *unidirectional* if they can be performed only in one direction. Bidirectional transformations can be achieved by defining two separate complementary unidirectional rules, one for each direction [75]. These classifications are orthogonal, e.g., a transformation specified in a declarative approach can be executed in either in-place or out-place way. In this thesis, we will consider homogeneous, in-place and unidirectional model transformation.

Another classification is based on the languages which are used to specify transformation rules [73]. In this classification, model transformations are either imperative/operational, e.g., QVT Operational Mappings, or declarative, e.g., graph-based transformation. Imperative languages specify explicit control flows about how a transformation should be executed; while declarative languages focus on what should be changed by the transformation [76]. In comparison, declarative model transformations have several advantages over imperative ones [55]. First, they are formally specified. Second, they support bidirectional transformation definition. Third, they share a simpler semantic model and hide procedure information from transformation definition. Thus, the order of execution, traversal of source models, as well as generation of target models are implicit; the semantic preservation between models can defined declaratively. However, operational approaches have advantages in execution, e.g., increase efficiency through incrementally updating models and control over the order of execution. In this thesis, we focus on declarative approaches, especially on graph-based transformation approaches.

1.5.2 Graph-based Transformation

The graph-based transformation approach is inspired by the theoretical work of graph transformations on different types of graphs [77]. In this approach, (meta)models are specified as graphs; the graph which represents a model is typed by the graph which represents the metamodel of the model. Model transformation rules are defined as *typed graph productions* on the model layer. Model transformations are executed based on these typed graph productions. The graph-based transformation approach is declarative and formal, and allows for composition; even though it has some scalability problems, lacks tool support and is incompatible to other approaches [72, 75]. Some tools, e.g., AGG, AToM^3, VIATRA2, GReAT [78, 79, 80] adopt this approach. Hereafter, model transformations implicitly mean graph-based transformations unless explicitly stated otherwise. In the following paragraphs, we will illustrate typed graph productions and discuss how they define model transformations.

Typed graph productions generally describe how to transform a typed graph by deleting and adding some elements. Formally, a typed graph production is specified as $\boldsymbol{p} : \boldsymbol{L} \leftarrow \boldsymbol{K} \rightarrow \boldsymbol{R}$. $\boldsymbol{L}$, $\boldsymbol{K}$ and $\boldsymbol{R}$ are graphs well-typed by a graph; $\boldsymbol{l} : \boldsymbol{K} \rightarrow \boldsymbol{L}$ and $\boldsymbol{r} : \boldsymbol{K} \rightarrow \boldsymbol{R}$ are two typed graph morphisms. Note that these two morphisms are required to be injective. $\boldsymbol{K}$ represents the unchanged elements; $\boldsymbol{L} \backslash \boldsymbol{K}$ represents the deleted elements; $\boldsymbol{R} \backslash \boldsymbol{K}$ represents the added elements. A model transformation rule can be specified as a typed graph production where $\boldsymbol{L}$, $\boldsymbol{K}$ and $\boldsymbol{R}$ are graphs well-typed by the graph which represents a metamodel. For example, the two rules in Example 5 can be specified as the two typed graph production in

Figure 1.12. The rule `Class-to-Node` adds a **Node** for each **Class**. The rule `Association-to-Edge` adds **Edge**s for each **Association**. Notice that we use **:** to denote the typing information. In addition, in the figure, we depict $\boldsymbol{L}$, $\boldsymbol{K}$ and $\boldsymbol{R}$ using colors for simplicity. For example, we use black to denote $\boldsymbol{K}$, green to denote $\boldsymbol{R} \backslash \boldsymbol{K}$ and red to denote $\boldsymbol{L} \backslash \boldsymbol{K}$. Thus, $\boldsymbol{L}$ ($\boldsymbol{R}$) contains all the elements in red (green) and black. In these two rules, there are no elements deleted hence no element is in red; it means $\boldsymbol{L} = \boldsymbol{K}$.

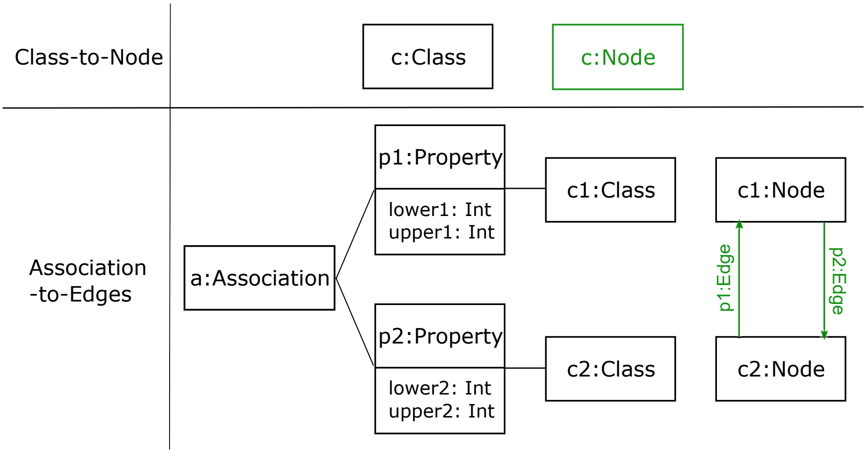


Figure 1.12: Graph productions

With graph productions, one can specify adding and deleting of graph elements, but cannot specify merging and splitting of graph elements. In [81], Lamo et al. proposed to specify model transformation rules based on integration models and co-span. In this approach, transformation rules are specified as co-spans $\boldsymbol{r} : \boldsymbol{L} \rightarrow \boldsymbol{I} \leftarrow \boldsymbol{R}$ where $\boldsymbol{I}$ is the integration model of $\boldsymbol{L}$ and $\boldsymbol{R}$. $\boldsymbol{r}$ contains two non injective graph morphisms. In this way, they can specify adding, deleting, merging and splitting of graph elements. In this thesis, we focus on graph productions.

Typed graph productions can be used to transform a source typed graph to a target typed graph. Such a transformation can be executed by applying a typed graph production to a typed source graph based on different formal theoretical mechanisms, e.g., double pushout (DPO) approach [77]. In the sequel, we will use the DPO approach to show how a model transformation is executed by applying a typed graph production.

Given a typed graph production $\boldsymbol{p}$ and a typed graph $\boldsymbol{G}$, a *direct transformation* $\boldsymbol{G} \Rightarrow \boldsymbol{H}$ is an application of the production via a match $\boldsymbol{m} : \boldsymbol{L} \rightarrow \boldsymbol{G}$, i.e., a typed graph morphism from $\boldsymbol{L}$ to $\boldsymbol{G}$. Such a transformation can be formally described as two pushout operations, called double pushout, shown as diagrams $(\mathbf{1})(\mathbf{2})$ in Figure 1.13. The transformation first deletes

$$\begin{tikzcd}
N \arrow[dr, "p"'] & L \arrow[l, "nac"'] \arrow[d, "m"] \arrow[l, phantom, "\neq", shift right=8] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\
 & G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

Figure 1.13: Double Pushout (DPO)

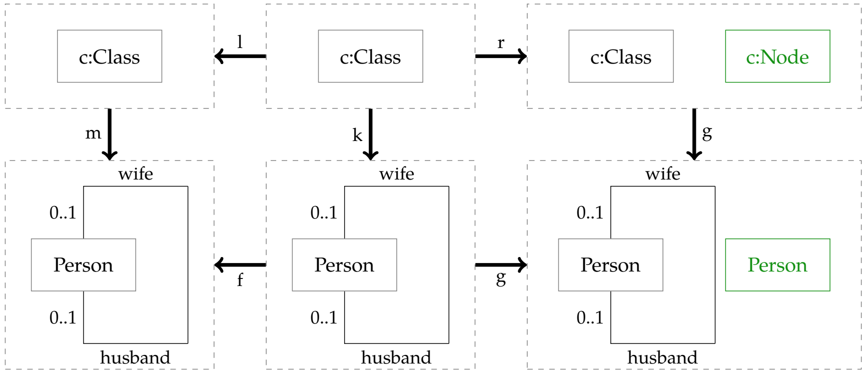


Figure 1.14: Direct model transformation using DPO

elements $m(L \backslash K)$ from G, thus derives a graph D. Then the graph H can be generated by adding the elements $n(R \backslash K)$ to D. This mechanism can be used to specify how a *direct model transformation* is executed, i.e., to transform a model by applying a model transformation rule. For example, a direct model transformation is executed by applying the rule `Class-to-Node` on the civil status model in Figure 1.2. The transformation is shown in Figure 1.14, in which no element is deleted and only one **Person:Node** is added, depicted in green.

Given a typed graph G, a typed production can be applied to transform the graph based on DPO approach if there exists a match $m : L \rightarrow G$ and the *gluing condition* [77] is satisfied. The gluing condition states that the *identification points*, i.e., elements identified by m (i.e., e_1 and e_2 in L whose images are the same $m(e_1) = m(e_2)$), and the *dangling points*, i.e., the nodes ns in L whose image $m(n)$ is the source or target of some deleted edges, are not changed.

The gluing condition is mandatory for DPO; it means that the condition must be satisfied when using DPO approach. In contrast, *application conditions* can be used to restrict the application of graph productions intentionally. For example, the `Class-to-Node` can be applied to the target graph in Figure 1.14 and generate another **Node**. This is what we want to avoid since the transformation aims to generate exactly one **Node** for each **Class**. Here, we use a *negative application condition* (NAC) on L to avoid the situation. A NAC on L is a graph morphism $nac : L \rightarrow N$ where N is a graph. A graph production can be applied to a graph if the NAC on L is not satisfied, i.e., there exists no inject graph morphism from $p : N \rightarrow G$ such that $nac; p = m$, as shown in Figure 1.13. For the two rules in Figure 1.12, N equals to R. Another kind of application condition is positive application condition (PAC). PAC is different from NAC in that a graph production can be applied to a graph if the PAC is satisfied. Note that, PAC/NAC can be specified on the right side R of graph productions too.

Model transformation from a source model S_1 to a target model S_n is a sequence of direct model transformation $S_1 \xrightarrow{r_1} \ldots \xrightarrow{r_m} S_n$. Each step $S_i \xrightarrow{r_j} S_{i+1}$ is an application of certain graph production r_j on S_i. The models S_i for $1 < i < n$ which are generated during the transformation are called *intermediate models*.

Single Pushout (SPO) approach [82] is another well-known approach to perform transformations. The difference between the two approaches is that DPO performs a transformation with two pushouts while SPO uses one pushout. Moreover, DPO is more strict since it requires the gluing condition when a graph production is applied to perform a transformation. It implies that, given a match of a graph production, the production may not be used to transform a graph. In comparison, SPO requires not the gluing condition and every graph production can be used to transform a graph if there exists a match of the production. However, a transformation using DPO can be invertible, which is not the case when using SPO. In addition, there exist other approaches which perform transformations. For example, the double-pullback approach is similar to the DPO approach except that the (1) and (2) in Figure 1.14 are pullbacks but not necessarily pushouts [83]; in the sesqui-pushout approach, the production morphisms l and r may be not injective and (1) is a pullback but not necessarily a pushout [84]. In this thesis, we will focus on transformations using DPO.

Until now, model transformation can transform a source model to a target model by deleting or adding nodes and edges. However, it is necessary to translate constraints from a source model into constraints in a target model to maintain the consistence between the two models. For example, in Example 5, we only created two direct edges in the DPF model for the bidirectional association in the UML model. However, in the UML model, there are constraints which specify requirement 1-3. In order to make the generated DPF model contains the same information as the original UML model, several corresponding constraints should be added, i.e., two multiplicity constraints and two irreflexive constraints on the two edges **wife** and **husband**, and an inverse constraint between the two edges. However, the previously mentioned model transformation approaches cannot specify this kind of transformation. In DPF, a *constraint-aware model transformation* [85] proposed a solution to this dilemma. With this approach, the diagrammatic constraints in the source models can be transformed into suitable diagrammatic constraints in the target models. In this thesis, we will not consider this kind of transformation. Therefore, we just mention this approach here. For interested readers, please refer to [55] for further details.

In addition to being used to specify model transformation rules, graph productions can also be used to specify *graph constraints* , which are properties of graphs. These graph properties can formulate whether a graph $\boldsymbol{G}$ contains (or not) a certain subgraph $\boldsymbol{G'}$, or whether a graph $\boldsymbol{G}$ contains a subgraph $\boldsymbol{G_1}$ providing it contains (or not) a subgraph $\boldsymbol{G_2}$. A graph constraint $\boldsymbol{P \leftarrow L \rightarrow R}$ $(\boldsymbol{N \leftarrow L \rightarrow R})$ consists of three components, which are the PAC $\boldsymbol{N}$ (NAC $\boldsymbol{P}$), the left-hand side $\boldsymbol{L}$ and the right-hand side $\boldsymbol{R}$, and two injective graph morphism. The three components are graphs well-typed by the structure of a model. Its semantics, i.e., whether a given graph satisfies the constraint, is depicted in the Figure 1.15. Given a graph constraint $\boldsymbol{gc}$, a graph $\boldsymbol{G}$ satisfies $\boldsymbol{gc}$ if, for each match $\boldsymbol{m : L \rightarrow G}$ which satisfying the application condition NAC/PAC, there is a match $\boldsymbol{n : R \rightarrow G}$ where $\boldsymbol{m = r;n}$. When we say that a match $\boldsymbol{m : L \rightarrow G}$ satisfies a PAC $\boldsymbol{pac : L \rightarrow P}$ (a NAC $\boldsymbol{nac : L \rightarrow N}$), we mean that there is a (no) morphism $\boldsymbol{p : P \rightarrow G}$ $(\boldsymbol{p : N \rightarrow G})$ such that $\boldsymbol{pac;p = m}$ $(\boldsymbol{nac;p = m})$. Rensink [86] generalized graph constraints as *nested condition* on simple graphs; Pennemann [87] lifted the application of nested condition to weak adhesive HLR categories. In addition, it is proven that the nested conditions on graph are equivalent to *graph formula* in First-Order Logic (FOL) [88]. In DPF workbench, we also implemented an editor to specify graph constraints, which will be discussed in Paper B.

$$\begin{tikzcd}
\boldsymbol{N} \arrow[dr, "p"'] \arrow[r, phantom, "\neq"', near end] & \boldsymbol{L} \arrow[l, "nac"'] \arrow[r, "r"] \arrow[d, "m"] & \boldsymbol{R} \arrow[dl, "n"] \\
& \boldsymbol{G} &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\boldsymbol{P} \arrow[dr, "p"'] \arrow[r, phantom, "="', near end] & \boldsymbol{L} \arrow[l, "pac"'] \arrow[r, "r"] \arrow[d, "m"] & \boldsymbol{R} \arrow[dl, "n"] \\
& \boldsymbol{G} &
\end{tikzcd}$$

(b)

Figure 1.15: The semantics of graph constraints

In the end, we will discuss the relation of models and model transformations. Recall that a model specifies the structural information of a software system; an instance of a model can be viewed as a state of a software system. Since a model transformation translates a source model to a target model, which are the instances of the corresponding metamodels, a model transformation can thus be viewed as a transition between the states of the software system which the metamodels represent. From this perspective, a metamodel and a set of model transformation rules define a transition system in which, the states are the models while the transitions between the states are the model transformations which are executed based on the transformation rules. The transition system can be viewed as a semantic behavior of the metamodel. Regarding to this, the model transformation rules specify or "model" a dynamic behavior of the software system which are represented as the metamodel. There exist some approaches, e.g., Petri Net, Activity Diagram and Sequence Diagram in UML, and Business Process Model and Notation (BPMN) which are oriented to behavior modelling. In a general context, the specifications defined with those approaches are also called models. In this thesis, we distinguish two types of models: structural models, or static models referred in [89], which are used to identify the concepts and their relationships in a software system, and behavior models, or dynamic models, which are used to specify the dynamic behavior of a software system. We will focus on structural models. Hereafter, models are used to denote structural models unless special considerations are mentioned.

CHAPTER 2

Verification in MDE

Models and model transformations are of great importance in MDE; models are the first-class entities in MDE while model transformations are the heart and soul of the methodology. In addition, software can be derived from models by model transformations. Therefore, the reliability and quality of software is highly dependent on the correctness of models and their corresponding model transformations, i.e., they fulfill some desired properties. In other word, it is a significant factor of the success of MDE to ensure that models and model transformations are correct. Thus, it is necessary and important to verify the correctness of models and model transformations in MDE. This is also the main topic of the thesis. In this chapter, we will firstly introduce the concepts related to verification in MDE. Then we will review the state-of-the-art of verification in MDE to illustrate verification techniques and properties that can be verified.

2.1 Introduction

The meaning of the term *verification* varies in different contexts. In [36], it means "to prove that something exists or is true, or to make certain that something is correct". In software engineering, verification means "confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled" [90]. Informally, verification is about "are we building the product correctly" [91]. Another similar concept for software quality assurance is *validation*. It means "confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled" [90]. Informally, validation is about "are we building the correct product" [91]. By comparison, in other words, verification ensures that software has been built according to some specifications, while validation ensures that software actually meets the needs of users, and that specifications are correct in the first place. From the technical view, verification involves static methods for verifying design while validation involves dynamic methods for checking and testing the real product [92]. In this thesis, we will not distinguish the two terms and use verification uniformly.

2.1.1 Verification Techniques

There exist different verification techniques in software engineering, informal or formal. Testing is an informal verification technique to find bugs in software by running test cases against some desired results, called *oracle*s. If a test case cannot produce the same result as its oracle, a bug is found. Test cases are usually designed manually. In the last few decades, researchers have been promoting automatic test case generation [93, 94, 95]. In order to test software more thoroughly, the test cases should have high degree of *code coverage* [96]. It means that the more parts of software are tested, the less chance that software contain bugs. The technique is widely used in software development to ensure the quality of software. However, "Program testing can be used to show the presence of bugs, but never to show their absence" [97]. In addition, since testing is performed by running software, this technique can only be applied after (part of) the software is implemented.

Similar to testing, runtime verification is also performed after a system is implemented. It checks whether a system satisfies a given property by monitoring of executions of the system with respect to the property [98]. The properties to be verified are usually specified in LTL or its variants [99, 100]. This technique is usually applied to verify systems of dynamic nature, e.g., service oriented systems, adaptive and self-healing systems etc, where other verification techniques cannot be applied [101]. Typically, runtime verification is performed by a monitor which answers whether an execution of a system satisfies the property. Monitors can be generated automatically from formal specifications. However, how to generate efficient monitors from specifications is a main challenge to apply this technique [98, 101].

In contrast, formal verification techniques, e.g., *model checking* [102] and *deductive verification* [103], can be applied before implementation and without running the software. These techniques can be used to verify systems by analyzing their mathematical representations. They can detect errors which cannot be found by tests, thus guarantee a higher level of quality. Model checking is an automatic technique to verify reactive systems which have finite state spaces [104]. The properties to be verified are usually specified in some temporal logics, e.g., linear temporal logic (LTL) or computational tree logic (CTL) [105]. A model checking algorithm exhaustively explores the state space of a system to check whether the system satisfies a property. The technique gains success in hardware verification and has also been applied to verify programs. However, the technique has the well-known *state explosion* problem: the state space of systems grows exponentially along with the size of systems. To handle this problem, several techniques are used to reduce the state space of systems. However, the problem remains still an obstacle to the use of the verification technique [102, 106].

Another formal verification technique, deductive verification, can be applied to verify systems which may have infinite state spaces by using logical reasoning [103, 107]. Using this technique, a system and the property to be verified are encoded as logical formulae in some formal logic, e.g., First Order Logic (FOL), Higher Order Logic (HOL). Thus, a verification problem, whether a system satisfies a property, is transformed to a logical reasoning problem: whether the formula representing the property can be derived from the formula representing the system by deduction procedures of the underlying logic [108]. The logical reasoning problem can be solved by using Constraint Satisfaction Problem (CSP) solvers, Boolean Satisfiability Problem (SAT) solvers (e.g. SAT4J [109], MiniSAT [110], zChaff [111]), satisfiability modulo theories (SMT) solvers (e.g., Yices [112], Z3 [113]) or *theorem provers* (e.g., HOL4 [114], ACL2 [115] Isabelle [116] and Coq [117]). Since most of the formalisms and languages are undecidable, e.g., OCL and FOL, these tools trade off between automation and expressiveness of the underlying logics [106]. For example, propositional logic (PL) is decidable, thus a logical reasoning problem in PL can be solved automatically by SAT solvers. However, the formulae in this logic is limited in expressiveness. In comparison, FOL is more expressive but is undecidable. Thus, SMT solvers can automatically solve some, but not all, logical reasoning problems in the logic; SMT solvers may produce the "UNKNOWN" result when the solver cannot find a proof for a formula. With theorem provers, it is possible to solve a logical reasoning problem in HOL, which is more expressive than FOL. But the approach usually requires manual interaction of the experts who acquire the knowledge of the formalism [104].

In the last decades, due to the advances in SAT-solvers and other satisfiability solvers, e.g., CSP solvers, the bounded verification approach [118] has become more promising. This approach reduces verification problems in a more expressive logic, usually undecidable, into logic reasoning problems which can be solved by SAT/CSP solvers automatically. However, such verification approaches usually impose a finite bound over the domains of system variables. Thus, the verification result can only hold within this bound, not for all the cases [106]. In addition, the verification approaches have the scalability problem: it becomes intractable or takes quite long time when large systems are verified.

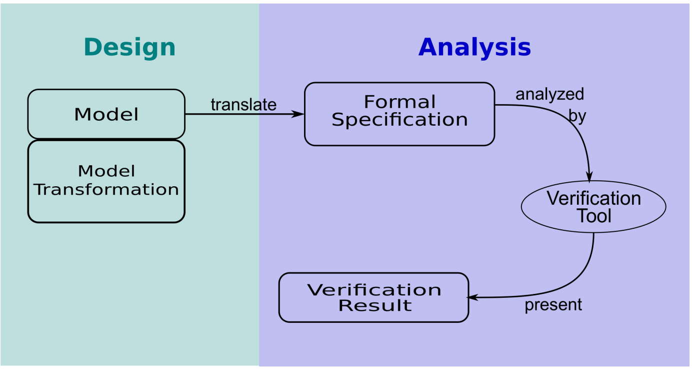


Figure 2.1: Verification of models and model transformations

The verification of models and model transformations can be performed by using the above-mentioned techniques and approaches. In general, models and model transformations are specified in the design domains, i.e., the syntax and semantics of the modelling languages, and the formalization underlying the languages. Meanwhile, verification techniques and approaches work in the analysis domains, i.e., the formalizations and logics that they use, and the tools and prototypes which implement them. The two domains are usually oriented to different users. For instance, UML and OCL are oriented for model designers. In contrast, model checkers, theorem provers and constraint solver have their own specification languages which are oriented to experts. Because of this difference, verification of models and model transformations is usually performed in two phases, as shown in Figure 2.1. In the first phase, models and model transformations constructed in the design domain and the properties to be verified are translated or encoded to corresponding specifications in the analysis domain. Then, in the second phase, different techniques or tools in the analysis domain can be used to verify the models and the model transformations by analysing the derived specifications.

In the following two sections, we will present an overview of verification approaches of models and model transformations separately. In each section, we firstly discuss the properties which are of importance. Then the existing verification approaches will be presented according to the verification techniques used in the analysis domain.

2.2 Verification of Models

The verification of models aims to check whether they are correct with respect to some requirements. For example, one may be concerned about whether some constraints in a model contradict each other; the contradiction among constraints will result in inconsistency in the model. Moreover, one may wonder whether some requirements have been already included or specified as constraints in a model. In the context of verification, we use the term *property* to denote such requirements. Cabot et al. [119] proposed several properties of models which are interesting to model designers. Most of the studies which will be discussed verify such properties. In general, the properties can be divided into the following two kinds [89].

Satisfiability Given a model and a proposition, whether some instance of the model satisfies the proposition. Examples of satisfiability are:

- *strong satisfiability* A model has at least one instance such that, for every type t in the model, there exists at least one element in the instance which is typed by t. Formally, the property is expressed as: $\exists I | I \vDash M \wedge (\forall t \in M, \exists e \in I | e : t)$, where M, I, t and e represent a model, an instance, a type in a model and an element in an instance, respectively. In addition, $e : t$ denotes that e is typed by t. The formulae below use the same notation.

- *weak satisfiability* A model has at least one non-empty instance. Formally, this is expressed as: $\exists I | I \vDash M \wedge (\exists t \in M, \exists e \in I | e : t)$.

- *liveliness of a type* t A model has at least one instance in which at least one element is typed by t. Formally, the property can be expressed as: $\exists I | I \vDash M \wedge (\exists e \in I | e : t)$.

Validity Given a model and a proposition, whether every instance of the model satisfies the proposition. Examples of validity are:

- *lack of constraint subsumption* Given a model M with a set of constraints $\{c_1, \dots, c_n\}$, a constraint c_i subsumes another constraint c_j where $i \neq j$ if, for every instance I of the model M', if I satisfies c_i then I satisfies c_j too, denoted as $c_i \Rightarrow c_j$. M' is the same as M except that M' excludes the constraint c_j. Formally, the property can be expressed as: $\forall I \vDash M' | I \vDash c_j$

- *lack of constraint redundancy* Two constraints c_1 and c_2 are redundant if c_1 subsumes c_2 and vice versa, denoted as $c_1 \Leftrightarrow c_2$.

In [119], constraints are considered redundant only if they subsume each other. Recall that constraints are used to describe requirements in the problem domain. From the modelling prospective, if c_1 subsumes c_2, the requirement specified by c_2 has already been described by c_1 implicitly. Thus, c_2 is redundant if both c_1 and c_2 exist. In the thesis, we consider a constraint redundant if another one subsumes it.

Note that constraints specify requirements too, but they are used in modelling context. Recall that model designers use constraints to define which structures can be considered as instances of the model under design. From this context, constraints are used to answer the question: *given a model and a structure, whether the structure is an instance of the model?* That is whether the structure satisfies the requirements specified by the constraints. In comparison, in verification context, properties are used to answer the question: *given a model, whether some or all instances of the model satisfy some proposition?*

Model designers are interested in satisfiability and validity of models. However, since the underlying formalisms of constraints/properties languages, e.g., OCL, may be undecidable, it is not possible to find an automatic procedure to verify arbitrary properties of models. Therefore, most verification approaches of models use different strategies which choose between automation and the expressiveness of underlying formalisms. In this section, we will give an overview of the literatures on the verification approaches of structural models. Most of the approaches verify structural models specified as UML class diagrams with attached OCL invariants (The two together are called UML class models hereafter). The approaches are categorized according to the strategies which they use.

2.2.1 Decidable Verification

Some formalisms or logics are decidable. For example, Description Logics (DL)s are logics to represent a domain of interest as concepts, which denote classes of objects, and roles, which denote relations between objects. DLs are used to formalize ontological models in databases and Semantic Web. Most DLs are decidable fragments of FOL. If verification problems can be formalized in these logics, they can be solved automatically [120].

Caoli et al. [121, 122, 123] proposed a verification approach of UML class diagrams. In their studies, they focused only on multiplicity constraints. They showed that UML class diagrams with such constraints can be formalized as knowledge bases in DLs. The knowledge bases are translated into linear inequalities, which can be resolved by some CSP solvers. They could verify finite properties of UML class models, e.g., checking whether a class is forced to have either zero or infinitely many objects.

Queralt et al. [124, 125] focused also on a subset of OCL in their verification approach of UML class models. The difference is that, they identified a decidable subset of OCL, called OCL-Lite, rather than just the multiplicity constraints as Caoli et al. focused on. They showed that their approaches ensured termination and completeness for verifying UML class diagrams with OCL-Lite constraints. Such UML class models could be encoded into DL knowledge bases, which are then analyzed by the DL reasoner, Pellet [126]. This approach can check properties like satisfiability and lack of constraint redundancy.

In two other studies, Queralt et al. [127, 128] proposed another approach to analyze database schemas, which were specified as UML class models, to verify properties like the liveliness of types and satisfiability. The authors firstly determined whether a model had any infinite instance. If not, a property could be checked by using a reasoning procedure which tries to construct an instance satisfying the property. The studies are implemented in the standalone tool AuRUS. This is more usable than the previously mentioned approaches in [121, 122, 123, 124, 125] where no tool or prototype were implemented. Moreover, Rull et al. [129] extended AuRUS by providing users a hint about how to change models to fix a problem when a model does not satisfy a property.

2.2.2 Bounded Verification

All the verification approaches mentioned in 2.2.1 can be performed automatically. However, the properties and the models which can be verified are limited. In comparison, the verification approaches which use bounded verification techniques can verify arbitrary models and properties automatically by using CSP solvers or SAT solvers. Usually, such approaches set limitation or bound on the structures of the models.

Constraint programming [130, 131] is a declarative programming paradigm in which the problem to be solved is described as a constraint satisfaction problem (CSP) and a solution to the problem can be given by a general constraint solver. A CSP represents a set of variables where each variable is associated with a finite domain. In addition, CSPs contain constraints, i.e., relations among the variables. A solution of a CSP is an interpretation which assigns a value to each variable and, at the same time, satisfies all constraints. A constraint solver finds a solution by exploring the search space, i.e., all the possible interpretation of the variables. Since each variable is associated with a finite domain, the search space is finite.

Cabot et al. [119] presented a bounded verification approach of UML class models by using constraint programming. In this work, the authors provided a transformation which translated a UML class model and the property to be verified into a CSP. Then the constraint solver called ECL^{i}PS^e [132] is used to find a solution to the CSP. If a solution is found, it means the UML class model satisfies the property. The verification approach is bounded, i.e., for each class or association in a class diagram, a number restricts how many objects or links an instance may have. Note that the numbers for various classes or associations may be different. In this way, the bound restricts the search space in which the constraint solver finds a solution to a CSP. However, the drawback of the approach is its incompleteness. If a solution is found, the verification result, i.e., the UML class model satisfies the property, is valid for all the cases. In other word, no matter how large the search space is, the UML class model always satisfies the property. Otherwise, it can only guarantee that the UML class model does not satisfy the property within the bound, because it is not certain whether a solution can be found within a larger search space.

The approach is implemented as a standalone Java application UMLtoCSP [133]. The tool loads a class diagram in XMI format which is created in the modelling tool ArgoUML [134] and OCL invariants in a separate file. It also allows users to set bounds for the classes and associations in the class diagram and choose the properties to be verified. If the solver finds a solution for the corresponding CSP within a bound, an image of an instance will be presented. Otherwise, it only shows that the property is unsatisfiable but presents no further information. The tool is outdated; it has not been updated since 2009. In addition, since the tool is not integrated into any modelling framework, the compatibility with the latest version of ArgoUML is not maintained; a class diagram created by the current version of ArgoUML cannot be loaded into the tool.

Two studies extended [119] from different perspectives. One of the extensions is the application of the verification approach on EMF models by González et al. [135]. This study is similar to [119] except that it verifies EMF models rather than UML class models. Therefore, constraints can be embedded into EMF models rather than being specified in a separate file. The work is implemented as a plugin EMFtoCSP in Eclipse [27] which is similar to UMLtoCSP [133]. If a model satisfies a property, the tool shows a real instance of the model rather than an image. As [119], the verification approach is bounded and provides no feedback when the model does not satisfy a property. The other extension is a slicing technique [136] which splits a model into several submodels based on the constraints and the property to be verified (mainly strong satisfiability and week satisfiability). The structure of a model can be split into several parts according to dependencies between classes. In this way, the technique reduces the verification of a whole model into the verification of its submodels. The authors present experimental results to show that the technique can make verification more efficient. However, the authors did not present a formal proof of the techniques. Moreover, the constraints are mainly multiplicity constraints and the properties to be verified are only satisfiability. In this thesis, a similar but formal technique will be presented to split models into submodels. The technique can handle arbitrary constraints other than multiplicity constraints and more properties, and is presented in Paper B.

There are also many researchers who proposed approaches to verify models based on the Boolean satisfiability problem (SAT) solvers due to the recent advances in SAT-solvers [137]. SATs are a subset of CSPs where all variables are boolean. SAT solvers check whether a boolean formula in proposition logic is satisfiable. That is, whether an assignment to the variables of the formula makes the formula true. SAT solvers are generally more efficient than CSP solvers [138]. In the following paragraphs, we will list some works which verify models by using SAT solvers.

Anastakasis et al. [139] presented an approach which was implemented in the tool UML2Alloy to check the satisfiability of UML class models. With their approach, UML class models and the property to be verified are translated into an Alloy specification. Then the Alloy Analyzer examines the specification by translating it into a SAT problem which is solved by a SAT solver. Since there exist differences between the formalisms of UML/OCL and Alloy, only a subset of UML/OCL can be translated into Alloy specification. In addition, the verification approach using Alloy is bounded, in similar manner as the approaches UMLtoCSP and EMFtoCSP; users must set bounds, which is called *scope* in Alloy, for the search space. Shah et al. [140] further extended [139] by translating instances of Alloy specifications into UML object diagrams. With this extension, it is possible to translate back and forth between UML and Alloy. However, there is no useful information provided when a model does not satisfy a property. For example, when a model is inconsistent, it should provide information about which elements cause such inconsistency.

Another bounded verification approach which uses SAT solvers is proposed by Kuhlmann et al. [141]. The approach is integrated into the USE framework [142]. USE was originally used to examine UML class models by generating arbitrary instances of the models (in USE, these generated instances are called snapshots). Instances of models are generated manually at first, then automatically by using a scripting language [142]. Later, Kuhlmann extended the tool by providing a technique to translate UML class models into formulae in FOL with relation features. These formulae are solved by the SAT-based constraint solver Kodkod [143]. Thus, USE has the feature of verifying satisfiability and constraint dependency of UML class diagrams. The verification in USE is bounded, and the bounds for classes and associations are configured in a separated file.

Both of the above mentioned approaches verify UML class models by using SAT solver. They are similar in many aspects: both perform bounded verification approach and present an instance if a model satisfies a property but provide no feedback otherwise. But the translations from the UML class models to SAT are different. UML2Alloy uses a formal model transformation which translate UML class models into Alloy specifications. The specifications are then translated into Kodkod structures by the Alloy Analyzer. While USE translates elements in the original UML class models into Kodkod structures directly. Moreover, the translation in USE can handle more concepts of UML class models than the one in UML2Alloy, e.g., association classes, OCL sequences and bags. In addition, UML2Alloy loads models in XMI file while in USE models are defined as specifications in plain text. The previous two groups of works verify models by indirect use of SAT solvers. In contrast, Soeken et al. [144] proposed a verification approach which uses SAT solver directly; UML class models and the property to be verified are translated into bit vectors. Then the vectors are translated into boolean expressions and passed to a SAT solver.

2.2.3 Interactive Verification

The previously mentioned verification approaches of models are all automatic, but they are also incomplete in the sense that they cannot verify arbitrary models and properties and some verification results are valid only within a bound. In comparison, the verification approaches using theorem provers are able to verify more expressive models and properties. In addition, verification results are also valid for all the cases. But this kind of approaches are interactive and require manual interference.

Egea et al. [145] proposed a formalization of metamodels, models (which are instances of the metamodels) and conformance relations between metamodels and models. With this formalization, metamodels and models are translated into specifications in membership equational logic (MEL), which can be analyzed by the validation tool ITP/OCL [146] to verify if a model is well-formed with regard to a metamodel.

HOL-OCL [147] is an interactive proof environment for verification of UML class models. It is integrated into a MDE toolchain [148] which supports a formal model-driven software engineering process. Models in the framework can be specified in different metamodels, e.g., UML, Dresden OCL2 or SecureUML [149]. With this verification approach, these models are encoded into theories in HOL-OCL, which are then analyzed in the interactive theorem prover Isabelle [116] to verify properties, e.g., the satisfiability of class invariants and no contradiction between postconditions of methods and class invariants. The verification process requires the knowledge and expertise of HOL and the Isabelle prover.

Rahim [150] analysed models by using the theorem prover Prototype Verification System (PVS). PVS is based on HOL and has its own specification language. The author proposed a set of rules which were specified in the Epsilon Transformation Language. The rules are used to transform a UML class model into a PVS specification. Then the specification is examined by PVS to verify the model. Since the focus of the work is the transformation from UML class diagram and OCL constraints into a PVS specification, no verification result is given.

Clavel et al. [151] proposed an approach to verify the unsatisfiablity of OCL invariants within a class diagram, i.e., there are no object diagrams that satisfy the OCL invariants. This property is the negation of consistency. In this work, they present a mapping from subsets of OCL to FOL. Then the derived FOL expression can be passed to an automated theorem prover or a SMT solver to verify the unsatisfiablity of these OCL invariants. The verification approach is unbounded, but can only handle subsets of

OCL. Furthermore, since this is a preliminary work, no tool or prototype of the approach is implemented.

Beckert et al. [152] formalized UML class models in dynamic logic, a multi-modal extension of FOL for reasoning about properties of programs. The formalization is implemented in Java and integrated into the KeY framework [153], which targets to provide a software development environment, including design, implementation, formal specification and verification. The verification is performed by an internal semi-automatic theorem prover, which requires manual interaction.

2.2.4 Set based Approaches

There also exist some studies which focus on mapping models into specifications in set based formalisms. Even though verification is not covered or not emphasized in their work, their formalization may lead to verification potentials in the future. Roe et al. [154] and Kim et al. [155] initialized works to translate UML class models into Object-Z (an extension of Z [156] to construct specifications in an object-oriented way) specifications. However, these studies either provide no tools, e.g., in [155], or need special expertise or knowledge of existing tools for Z and Object-Z [154]. Moreover, Marcano et al. [157] proposed a verification approach of UML class models by using B [156]. In this approach, a class model is translated to a formal specification in B. Then the specification is analyzed by Atelier-B (a tool which aims to develop quality-ensured software by rigorous mathematical reasoning) to verify consistency of UML diagrams and detect contradiction between invariants. However, when an error is found in the model, it requires special knowledge to understand B specifications to find the problem. Szlenk [158] proposed a mathematical formalization of the semantics of UML class diagrams using notions from sets and partial functions. Based on this, they presented an approach to verify consistence of UML class diagrams. However, the approach did not consider any constraints.

2.2.5 Summary

Feature	2.2.1	2.2.2	2.2.3
Logic	DL	FOL	HOL
Decidable	✓	×	×
Automatic	✓	✓	×
Bounded	×	✓	×
Analyzer	DL reasoner	constraint solver	theorem provers

Table 2.1: Features of verification approaches

From the overview, we can see that most of the works verify models by using deductive verification techniques. They formalize the models to be verified into specifications in some logics. Then the specifications are analyzed by logical reasoning tools to verify properties. But these verification approaches have distinctive features since they use different verification techniques. Table 2.1 summarizes the features of the aforementioned verification approaches in each category[1]. It shows that the approaches trade off between automation, expressiveness of the underlying logics and completeness of verification results. Some automatic approaches can verify properties specified in less expressive logic, e.g., DL, and can ensure that verification results are valid for all cases. Other automatic approaches can verify some properties specified in a more expressive logic, e.g., FOL, but verification results may not be valid, since they are bounded. The other approaches can verify properties specified in very expressive logic, e.g., HOL, where verification results are valid for all cases. But this is at the expense of losing automation.

As stated before, the modelling and the verification are proceeded in different domains. From the perspective of model designers, the automatic verification approaches are preferable over the ones which require manual intervention since the former requires little or no knowledge in the analysis domain. In addition, the approaches which are shipped with tools gain more favor than the ones without tools. Most of the verification approaches are supported by tools. However, since most of the tools are oriented to verification purposes, they are not integrated into modelling environments; (EMFtoCSP is integrated into Eclipse but the tool is not well maintained; the latest version does not work: the generated CSP specifications cannot be read by ECL^{i}PS^e.) We have identified three challenges related to existing verification approaches.

1. In order to verify models, model designers have to commute between modelling tools and verification tools to verify a model. This is not convenient to verify models specified in the modelling domain. In this thesis, we will present a bounded verification approach by using Alloy which is integrated into the DPF workbench [1]. The work is presented in Paper B.

2. If a model satisfies a property, most of the verification approaches present a result, e.g., an instance of an UML model. Otherwise, they will present no more information than claiming that the model does not satisfy the property. In our verification approach, when a model does not satisfy a property, the problematic part of the model will be highlighted.

[1] The works in subsection 2.2.4 only provided encoding approaches but no verification approach. Therefore, they are not list in the Table 2.1

3. Automatic verification approaches usually have scalability problems. When a larger model is verified, the approaches become intractable or take quite long time. To solve the problem, we present a technique to split models into submodels such that the verification of models can be reduced to the verification of submodels. This technique is also present in Paper B.

2.3 Verification of model transformations

The verification of model transformations aims to check whether the model transformations or the generated target models satisfy some properties. For example, given a model transformation system, one may be interested in whether model transformations from every source model will terminate, whether the generated target model is well-formed with regard to the target metamodel, etc. There are also many other properties which are studied in the literature. In [159, 160], the authors reviewed and classified the properties to be verified of model transformation into two categories: *language-related* and *transformation-related*. Language-related properties concern model transformations from a *computational* perspective [159]. From this perspective, a transformation can be viewed as a computation which is executed by a transformation engine according to transformation rules. Typical properties in this category are:

1. *termination* guarantees that the execution of model transformations will terminate

2. *determinism* guarantees that model transformations will generate a unique target model. Termination and determinism together are called confluence.

3. *typing* ensures that transformation specifications are well-formed with regards to their transformation languages

4. *preservation of execution semantics* ensures that the execution of transformations is performed according to the semantics of the corresponding transformation languages. For example, some languages may not allow that an element in the source model to be matched twice.

The properties in the transformation-related category examine model transformations from a *modelling* perspective [159]. From this perspective, model transformations are viewed as transitions between different models. These properties concern the semantics of:

1. the source and target metamodels; an example property is *conformance* of target models. It means that model transformations always generate target models which are instances of the target metamodels

2. *model syntax relations* which ensure that certain structures of the source models will be transformed into other structures of the target models. These relations connect the patterns of the source models with the patterns of the target models

3. *model semantic relations* which connect the meaning of the source models to the meaning of the target models. An example property is *bisimulation* where both models "are able to simulate each other from an observation point of view" [159]

In the thesis, we focus on conformance of target models. Given a metamodel and a model, it is trivial to examine whether the model is an instance of the metamodel by checking the model against the structure and constraints of the metamodel. However, it is not trivial to verify the conformance of target models from model transformations, since this involves transitions of models according to model transformation rules [161]. We propose two solutions to verify conformance. One solution is to check a *direct* condition in which each direct model transformation from an instance of the source metamodel can generate an instance of the target metamodel. Another solution is to check a *sequential* condition in which each model transformation from an instance of the source metamodel can generate an instance of the target metamodel after the application of a number of transformation rules. The sequential condition is weaker than the direct condition since it does not require that the intermediate models in a model transformation conform to the target metamodel. The study is presented in Paper C.

Different approaches have been proposed in the literature to verify such properties. In this section, we will review the literature and categorize the approaches according to the techniques they use.

2.3.1 Manual Mathematical Proof

Some studies have tried to verify properties of model transformations by constructing manual mathematical proof, rather than using verification techniques, e.g., testing, model checking, theorem proving, as shown in the sequel. Such verification requires related theory background and mathematical knowledge. But the studies usually can guarantee certain properties for all transformations [159].

Some researchers have proposed some criteria and proved that if model transformations fulfill the criteria, the model transformations satisfy the desired properties. Bruggink [162] proposed some criteria to verify termination of graph transformation systems. The author observed the fact that, transformations included *creation chains*, i.e., chains of edges where each edge involved in the creation of the next edge. The existence of infinite creation chains was the source of infinite rule applications. They also presented an algorithm to prove the absence of infinite creation chains by recording the length of creation chains. If the length was bounded, there was no infinite creation chains, thus proved termination. Based on the same idea, Küster [163] also established a set of criteria to verify termination and confluence of model transformation systems with control conditions.

Varró et al. [164] also proposed criteria to verify termination of graph transformation systems. But these criteria were based on Petri net using algebraic techniques. A simple Petri net was derived to simulate a graph transformation system; the Petri net abstract from the structure of instance models and only count the number of elements of a certain type. The graph transformation system was proved terminating if the Petri net run out of tokens after limited number of steps.

Heckel et al. [165] proposed some criteria to verify whether a graph transformation system is confluent. They analyzed graph transformation rules and generated critical pairs which were two parallel dependent transformations, i.e., the intersection of their matches did not consist of common gluing points. Such transformations may result in violation of confluence. They proved that a graph transformation system was confluent if it was terminating and every critical pair was strictly confluent. The critical analysis is implemented and integrated into the modelling tool AGG [166].

There are also some researchers who have proposed approaches to construct model transformation systems. They proved that a model transformation system satisfies some desired properties if it is constructed with their approaches. Ehrig et al. [167] introduced a mechanism, called layered graph grammar, which group rules into different layers according to deletion and nondeletion layer conditions. The application of rules are ordered by the layers. In this way, the transformation steps which create elements are separated from the ones which delete elements. The authors showed that a layered graph grammar with injective matches terminates. Barroca et al. [168] proposed a transformation language DSLTrans to specify model transformation systems. The language also uses layered transformation rules which guarantees confluence and termination of model transformations by construction. Different from layered transformation rules, Lamo et al. [81] proposed to specify model transformation rules based on integration models and co-span. In this approach, transformations are performed by rule amalgamation [169] instead of applying rules one by one. They showed that the approach guarantees confluence and termination.

The two studies above proposed approaches to construct general model transformation systems. Some other similar approaches are oriented to special intentions, e.g., refactoring, refinement, etc. Baar et al. [170] proposed a mechanism to construct graph transformation rules for refactoring of UML class models. This mechanism is proved to preserve semantics before and after refactoring, i.e., the semantics of models before refactoring coincides with the semantics of models after refactoring. Hermann et al. [171] proposed a model synchronization framework generated from TGG with bidirectional update propagation operations. The operations are proved to preserve consistency and are invertible to each other. Padberg et al. [172] proposed a formal rule-based refinement technique of algebraic Petri nets. The technique preserves safety properties combined with the introduced place preserving morphism. Massoni et al. [173] proposed an approach to develop model refactorings that preserve semantics by construction for UML class diagrams. A set of basic, semantic-preserving transformation laws were used to compose more complicated model refactorings. The laws were verified by translating them and the class diagrams into Alloy to reason about the soundness.

In summary, the studies in this category mainly concern termination and confluence. They examine model transformations from the computational perspective. Since termination and confluence of graph transformations are proved undecidable [174, 175], the works trade off between the expressiveness of the transformation language and the desired properties [159]; some studies allow powerful transformation languages but can only promise that the properties are satisfied if their criteria are fulfilled; The others promise that the desired properties are always satisfied but restrict the expressiveness of transformation languages.

2.3.2 Testing

Testing, as an informal verification approach in software engineering, is to "exercise software with test cases to find failures or demonstrate correct execution" [176]. The traditional testing techniques can be applied to verify model transformations but face three challenges: 1. automatic generation of test models (the source models which are transformed and tested), 2. the specification of test oracles which is used to check whether transformations produce desired target models, 3. the comparison of the test models and test oracles [177].

Numerous studies tried to generate test models with guaranteed metamodel coverage, i.e. each source metaclass should be instantiated at least once in at least one test model and, properties of metaclasses (e.g., meta-attributes) should take several representative values [178]. For example, Fleurey et al. [179] used equivalence partitioning to achieve coverage. They manually identified equivalence classes for a source metamodel. Then a tool is used to automatically generate a test model for each class. Strüermer [180] proposed a similar approach but used a classification method to identify equivalence classes. However, the equivalence partitioning technique may produce numerous test models; many of these models are unrelated since model transformations may only affect a part of the metamodel. To solve the problem, some researchers also computed effective metamodel [179, 181], i.e., the fragment of the source metamodel actually affected by the transformations; other researchers [179, 182] proposed using mutation-testing techniques to generate test models semi-automatically; some test models are provided manually by testers and then a tool generated comprehensible test models using mutation testing techniques. Different from these techniques, Sen et al. [183] proposed an approach to generate test models using the Alloy Analyzer.

As for generation of test oracles, four methods can be used for testing transformations as summarized by Mottu [184]: reference transformation, inverse transformation, expected output models and constraints. Most works in the literature use the last method, where the outputs of transformations are checked against constraints, e.g., post-conditions of transformations or invariants of target meta models. Guerra [185] proposed a visual contract language to specify correctness requirements for model transformations; Baudry [177] specified constraints in modified OCL; Kolovos et al. [186] proposed a special language, the Epsilon Comparison Language (ECL), which can be used to specify constraints between source and target models. In addition, graph patterns can be used to specify constraints among models where target models are checked whether they match these patterns. Moreover, Orejas et al. [187] used the graph patterns approach to specify constraints.

There exist also testing tools which implement the above mentioned approaches. The tools generally can construct test cases which consist of test models and test oracles. A testing engine is also included to perform transformations and check the produced models against the test oracles. Moreover, optionally, a testing analyzer can be used to examine the test result. For example, Lin et al. [188] proposed a testing framework which used C-SAW model transformation engine to run test specifications. The specifications contain transformation rules, test models and test oracle. The framework has a test analyzer to highlight the test result in the target models. Giner et al. [189] performed testing similarly, except they used the EPSILON tool which generates test models from the HUTN description and uses the EVL script to verify target models. The tool has no test analyzer. The two works do not consider metamodel coverage when generating test cases. In comparison, Darabos et al. [190] provided metamodel coverage support by using mutations.

Even though the mentioned works make progress in testing of model transformations, verification with testing is an informal approach which is not complete; it tries to obtain verification confidences through test case coverage. As pointed out in [161], verification with testing is sensitive to the implementation of model transformations and the previous works are most specific to certain model transformation languages. Moreover, the properties are mainly static constraints on the target metamodel.

2.3.3 Model Checking

Model checking is an automatic technique to verify finite state systems. It searches exhaustively through the state space of a given system to determine if the system holds some behavior properties which are usually expressed in temporal logics. It was firstly used successfully in verifying hardware systems, e.g. complex sequential circuit, and is also used to verify software [102].

In [191], Heckel initialized the application of model checking to verification of graph-based model transformations. The idea is to interpret a graph transformation system as a transition system in which states are graphs and transitions are given by applications of transformation rules. Thus, properties of graph transformations can be verified by using the model checking technique.

In [192, 193, 194], Rensink presented the tool GRaphs for Object-Oriented VErification (GROOVE). It attempts to construct the state space of graph transformation systems using an abstraction technique. GROOVE uses simple edge-labelled graphs [2] to denote states while the transitions between states are generated by the application of graph transformation rules with negative application condition by using the single-pushout approach. Thus, the existing model checking algorithms can be applied to verify linear temporal properties of model transformations by automatically analysing the generated state space [196]. However, the tool encounters the state explosion problem; the state space constructed from the tool is usually large, even though some abstraction techniques are applied [197]. Another tool, Henshin [198], uses the same approach to generate state spaces but focuses on in-place model transformations. It encounters the same problem.

Lúcio et al. [199] presented a symbolic model checker to verify model transformations specified in the DSLTran language. The model checker constructs the state space from model transformation rules. Since model transformations in DSLTran are confluent and terminating by construction, each state in the state space corresponds to a target model by applying all the possible transformation rules in a given layer. In addition, each transition corresponds to transformations between two adjacent layers. The properties to be verified are the relation between source and target models, e.g., providing a certain structure appears in the source model, whether another structure is presented in the target model. They are also specified as transformation rules in the DSLTran language. Such a property is satisfied if the property holds in every path of the state space. If the property is not satisfied, a counterexample can be presented to assist the designer to fix the problem.

These studies tried to construct model checker for graph-based model transformations. In contrast, there are also some studies which verify model

[2] The tool is extended to support attributed graphs by Kastenberg [195]

transformations by using existing model checkers. Schidt et al. [200, 201] presented a tool CheckVML which translated graph transformation systems into Promela models. Such models can be analyzed by the model checker SPIN [202]. Troya et al. [203] proposed a formalization for the semantic of ATLAS Transformation Language (ATL) [204] in rewriting logic. ATL is a domain-specific language for model-to-model transformation specification based on OCL formalism. The language provides a mixture of declarative and imperative constructs. Based on this, they used the model checking tool Maude [205] to simulate and analyse transformations. The approach can verify such properties as whether an interesting target model can be derived, or whether every source model can be transformed. Gracia et al. [206] presented an approach to verify model transformations which are specified as transformation algorithms in $^{+}$CAL [207] to manipulate models. $^{+}$CAL is an algorithm language to write high-level descriptions of algorithms. Such algorithms can be translated into formal specifications which can be analyzed by model checkers. In this work, the models are specified using Essential MOF (EMOF) and OCL. To verify such transformations, EMOF and OCL are formalized as a specification in $^{+}$CAL. Thus, the specification along with the transformation algorithms in $^{+}$CAL can be fed into the model checker TLC [208] to verify two properties:

1. transformation can produce valid target models for every valid source model;
2. the generated target models satisfy certain constraints.

Boronat et al. [209] proposed an approach for verifying endogenous model transformations. The approach is implemented in MOMENT2 in which model transformation systems can be defined in EMF and verified in Maude [205]. In the approach, models specified in MOF/OCL and model transformation rules specified in QVT are formalized into a specification in rewriting logic. The specification can be analyzed by a LTL model checker in Maude to verify different properties, e.g., safety and liveness properties.

In summary, the verification approaches in this category either construct a special model checker or use an existing model checker to verify model transformations. But the model checking technique has an inherent problem: the state explosion problem, which hinder the application of the technique. In addition, since model checking is used to analyze finite state systems, it is difficult to use the technique to verify the model transformation systems which have infinite state spaces.

2.3.4 Theorem Proving

“Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system." [210]. There are bunches of studies which verify model transformations by using theorem proving [161]. Generally, the model transformations specified in a language or other forms are translated into the specifications used in theorem provers. Then properties to be verified are checked by finding a proof with the theorem provers.

Asztalos et al. [211] proposed a first-order logic based formalization of model transformations with control flows. This formalization uses assertions in Assertion Description Language (ADL) to describe the constraints of source models, model transformation rules, the pre- and post-conditions of model transformations and properties to be verified. These assertions are added onto the control flow graph of model transformations. In addition, the authors also proposed deduction rules which can be used to derive new assertions from initial assertions. The deduction rules are general and not dependent on model transformations. Thus, they can verify whether model transformations satisfy some properties by checking whether the assertions for the properties can be derived from the initial assertions for the model transformations by using the deduction rules. The verification approach is integrated in the Visual Modelling and Transformation System (VMTS) [212] which is an n-level metamodelling and model transformation specification framework. The initial assertions can be derived automatically from model transformation specifications within the framework. The verification part is implemented in the logic programming tool SWI-Prolog [213]; the initial assertions and the deduction rules are specified as a program; the verification of properties can be executed as queries of the program. The approach can be used to verify properties like termination and confluence.

Calegari et al. [214] also proposed a verification approach of model transformations by using the type theory, Calculus of Inductive Construction (CIC) [215]. But the approach aims to verify transformations which were specified in ATL [204]. With this approach, ATL transformations are formalized as CIC specifications which are then analyzed by the theorem prover Coq [215] to verify whether the generated target models always satisfy postconditions if the source models satisfy preconditions.

Lano et al. [216] presented UML Reactive System Development Support (UML-RSDS), a subset of UML to specify model transformations. In this work, UML class diagrams are used to specify model transformation rules. The control flows of model transformations, i.e., the conditions and the order of the model transformation execution, are specified as an UML activity diagram. Moreover, the pre- and post-conditions of each transformation rule can be specified as OCL expressions. In addition, they also presented toolsets to verify and analyse such transformations. Depending on the properties to be verified, different verification techniques are integrated into the toolsets. For example, to verify syntactic correctness and language-level semantic correctness of rules, the model transformation specifications are translated automatically into B [217] specifications which are verified by internal consistency proof in B. The tool can also be used to verify other properties, e.g., confluence, by syntactic analysis of transformation rules.

Cabot et al. [218] proposed an approach for verifying declarative model-to-model transformations, which is transformed to the verification of models. Given a declarative description of transformations, e.g., in Triple Graph Grammars (TGG) [219] and QVT, a set of OCL invariants can be automatically generated. The invariants state under what conditions source models and target models can represent transformations according to the transformation rules. The invariants, as well as the source metamodel and the target metamodel, are treated as static UML/OCL class diagrams, called transformation models [220]. Thus, the existing verification tool for models, e.g., UMLtoCSP [133] or HOL-OCL [147], can be used to analyse the transformation models to verify some properties of the transformations, e.g. whether all valid source models can be transformed. These properties can be encoded as consistency properties of the transformations models.

Some researchers proposed verification approaches for model-to-code transformations which are formalized as model-to-model transformations. Stenzel et al. [221] presented a framework by using the interactive theorem prover KIV [222] to verify Java code generation. In this framework, Java code generation is specified as QVT transformations from some source models, e.g., UML models or Ecore models, to the Java annotated abstract syntax tree (JAST). Then the QVT transformations and the semantics of JAST are formalized as a formal calculus in KIV. The calculus is fed into KIV to check whether the generated Java code is type correct and satisfies some semantic properties. Based on the same idea, Giese et al. [223] also proposed verification of model-to-code transformations. The difference is that, the transformations are formalized as TGG in Fujaba tool suite [224]. In addition, they use the theorem prover Isabelle/HOL [116].

In summary, the verification of model transformations by using theorem proving can check various properties, from termination to the conformance of target models. However, usually, the process to find a proof is semi-automatic; it requires manual assistance. This implies that the one who verify model transformations should acquire required mathematical knowledge, which is not usually met by software engineers. Furthermore, none of the studies in the referred literature generate counterexamples when the properties to be verified are false. In this case, it is difficult to know what causes such failure.

2.3.5 Automatic Reasoning

Due to the advancement in automatic reasoning techniques, e.g., SAT solvers and SMT solvers, some studies applied these techniques to verify model transformations automatically.

Inaba et al. [225] proposed to verify the conformance of target models by using Mona [226], a decision procedure in Monadic Second-Order Logic. In this study, they focused on model transformation rules specified in Core UnCAL, a subset of the graph transformation language UnQL [227]. The verification problem in the subset of the language can be reduced to the validity of monadic second-order logic formula over trees, which is decidable and can be solved by Mona. However, the expressiveness of the model transformation language is restricted, e.g., only the typing of graphs can be described.

Büttner et al. [228] proposed a verification approach of ATL model transformations by using SMT solvers. They contributed a formalization for a subset of ATL. This enables to encode ATL transformations into first-order logic formulae. Then the formulae were passed to SMT solvers to verify whether the model transformation can always generate instances of target metamodels from instances of source metamodels. If an invalid instance of target metamodels is generated, a counterexample can be presented to assist the designer to fix the problem. However, it requires expert knowledge to understand the counterexamples. Even though the approach is oriented to a subset of ATL, the approach is incomplete, i.e., it cannot verify all properties. SMT solvers may generate an "UNKONWN" result to indicate that the properties cannot be verified.

In [229], Büttner et al. proposed an algorithm which translated model transformation rules specified in a subset of ATL into a transformation model [220]. The transformation model merged all the related information, e.g., the source/target metamodels and OCL invariants of transformations. Then the existing verification approach of models can be applied to verify whether the target models satisfied some desired properties. For example, UML2Alloy [139] can be used to translate the transformation model into an Alloy specification which is examined by the Alloy Analyzer. This approach is applied to verify an industrial case in [230]. Büttner et al. [231] also proposed a similar verification approach which was aimed to analyze refinement. These approaches translated model transformations into intermediate transformation models and then to Alloy specifications. In contrast, Baresi et al. [232] translated model transformation rules specified using AGG [166] into Alloy specifications directly and verify properties like whether a specific target model can be generated after a finite model transformation steps by using the Alloy Analyzer. The studies provided no general translation between model transformation rules and Alloy specification. Based on the same idea, Anastakasis [233] proposed a verification approach which simulated model transformations as Alloy specification directly. Then the Alloy Analyzer can be used to verify whether each transformation can generate target models which conform to the target metamodel. If not, a counterexample will be given by the Alloy Analyzer. By contrast, the counterexample is easier to understand than the one given by SMT solvers. The author illustrated the approach with a running example, but provided no systematic translation from model transformation to Alloy specifications. The verification approaches using Alloy are bounded. Users need to set a bound for the model transformations. The bound may restrict how many elements that are included in the source models, or how many model transformation steps are examined. As applying bounded verification approaches on models, the approaches are also incomplete. It means that the verification result may be valid within the bound.

By using automatic reasoning techniques, researchers can formally verify model transformation automatically. This is different from the approaches with theorem provers, where the verification is usually performed interactively. However, since properties of model transformations are generally undecidable [88], most of the studies deployed two strategies to verify the properties: they either aimed to verify model transformations in (a subset of) a specific language, or used bounded verification approaches to verify general model transformations. In addition, the approaches also have scalability problem: it takes longer time or becomes intractable when larger model transformation systems are verified.

2.3.6 Summary

Feature	Mathematical Proof	Testing	Model Checking	Theorem proving	Automatic Reasoning
Formal	✓	×	✓	✓	✓
Automatic	×	×	✓	×	✓
Problem	Require expert	Informal, Expensive	The state explosion problem	Require expert	Incompleteness
Property	LR	TR	TR	LR& TR	TR

LR: Language related

TR: Transformation related

Table 2.2: Features of verification approaches

Various verification techniques can be used to verify model transformations. Because of inherent difference of the underlying techniques, the verification approaches have their pros and cons. Table 2.2 shows the features of the aforementioned verification approaches. Mathematical manual proof can guarantee verification result for all the transformations. But this requires expertise knowledge of mathematics. Testing is popular in industry, but it is expensive and informal; it requires construction of test cases and cannot find all the bugs in the model transformations. The formal verification techniques can examine model transformations, but they have limitations. Model checking can be performed automatically. But the state explosion problem hinders its application; it may take long time or become intractable to verify a model transformation system. With theorem provers, complex properties can be verified, but it usually needs interaction with the users. This requires special knowledge of the provers and the underlying formalisms. Automatic reasoning is incomplete in that either some properties cannot be verified or the verification result is valid within a bound.

In this thesis, we choose automatic reasoning with Alloy for two reasons. Firstly, it inherits merits of formal verification. Model transformations are formalized as Alloy specifications. Secondly, it promises termination for all properties to be verified since verification by the Alloy Analyzer is bounded. In addition, it gives designers quick feedback when they construct model transformation rules. For example, when a property is not satisfied, the counterexample may warn designers and hint how to fix the problem. In Paper C, we present the bounded verification approach of model transformations. We propose a transformation from graph productions to Alloy specifications Then we use the Alloy Analyzer to verify whether the target models generated always conform to the target metamodel. This is similar to the approaches [232, 233]. But we presented a systematic translation from model transformations to Alloy specifications, which is not given by the two studies. In addition, to solve scalability problems, we also proposed some techniques which are present in Paper D. Moreover, in Paper E, we applied the verification approach to analyse workflow in healthcare domain. We demonstrate the application by verifying properties of the workflow modelling language DERF [234] and general properties of workflow models, e.g., termination and absences of deadlocks.

CHAPTER 3

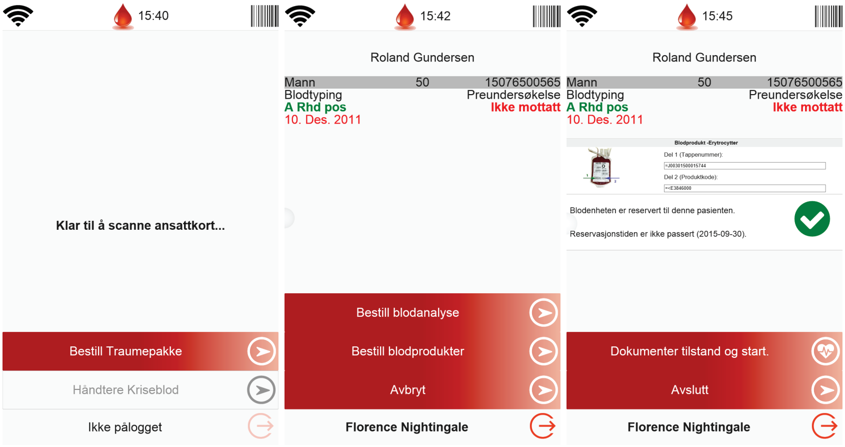
Contribution

In this chapter, we will illustrate the contributions of the thesis. The main contributions consist of three parts: 1. Enhanced tool support for diagrammatic (meta)modelling. 2. Verification of structural models. 3. Verification of model transformations. In the next subsections, we will detail each of these parts and use a running example from the health care domain to demonstrate them. This example describes a blood transfusion workflow used in a joint project between Bergen University Hospital and Bergen University. In this project, the hospital decided to develop an app to improve patient security in blood transfusion workflow. We specified the workflow as a workflow model as a high level design of this app by using DPF framework. The app is later implemented based on the model. Moreover, we verified the model by using the verification approaches in the thesis. This ensured that the model satisfied some desired properties in the requirement.

3.1 The Running Example: Blood Transfusion

We will first introduce the running example which is about blood transfusion workflow in the healthcare domain. This example will be used throughout this chapter to illustrate the contributions of the thesis. The blood transfusion workflow presented in this section is described in the guidelines used at Haukeland University Hospital in Bergen, Norway[1].

[1] The PhD candidate joined a project at the Haukeland University Hospital in which he developed a blood transformation app to assist nurses during blood transfusion.



15:40

Klar til å scanne ansattkort...

15:42

Roland Gundersen

Mann 50 15076500565
Blodtyping Preundersøkelse
A Rhd pos Ikke mottatt
10. Des. 2011

15:45

Roland Gundersen

Mann 50 15076500565
Blodtyping Preundersøkelse
A Rhd pos Ikke mottatt
10. Des. 2011

Blodprodukt -Erytrocytter

Del 1 (Tappenummer): =J00301500015744

Del 2 (Produktkode): =<E3846000

Blodenheten er reservert til denne pasienten.

Reservasjonstiden er ikke passert (2015-09-30).

(a) (b) (c)

Figure 3.1: Screenshots of blood transfusion application

Blood transfusion is a common medical procedure in which patients receive blood products to replace lost blood, e.g., during a surgery or injury. The complete blood transfusion workflow includes many tasks to ensure the quality of blood products and the safety of the recipients. For example, blood is collected from different blood donors; blood donations are screened against infections, e.g., HIV, prior to use; blood collections can be processed into different components, e.g., red blood cells, white blood cells or plasma, for more effective usage; blood products are stored in a blood bank which stores and preserves blood in hospitals; the blood of recipients must be typed and screened to ensure compatibility before transfusion [235].

Since blood transfusion is such a highly complex and safety-critical procedure, it is necessary to have computer-based applications that assist health personnel to perform the task. The Haukeland University Hospital requires therefore an application running on handheld devices to assist nurses in performing blood transfusion tasks. Some screenshots of the application are shown in Figure 3.1. Figure 3.1a shows the initial state of the application; Figure 3.1b shows that a nurse have logged into the application; Figure 3.1c shows that a blood transfusion is successfully performed.

We will now introduce a typical blood transfusion scenario in order to clarify the tasks which the application should support. For the sake of simplification, the application only considers a part of the blood transfusion procedure. That begins with obtaining blood products from the blood bank with a completed blood transfusion at the end.

Assume Dr. Danielsen will perform a surgery on Mr. Gundersen (patient) in a few days and blood transfusion is needed during the operation.

Dr. Danielsen asks Miss Olsen (nurse) to order blood from the blood bank. Firstly, Miss Olsen will log into the application by scanning her employee card. Then she identifies the patient information by scanning the bar code on the wristband of Mr. Gundersen. Afterwards, Miss Olsen should send two items to the blood bank: two blood samples of Mr. Gundersen and a blood order. The blood bank uses the samples to identify the blood type of the patient, if it is unknown, and to perform some pre-transfusion screening for infection test. The blood samples must be labelled with the patient information before being sent out. The blood order should contain information about how many units of different blood products should be ordered, which departments should pay for the ordering and when the blood products will be used. The blood order must be authorised by Dr. Danielsen before being sent out.

When the two items are sent out, the nurse waits for the blood products. We assume that the blood products arrive on time. During the surgery and before the transfusion of each blood product, Miss Olsen should check whether the product is prepared for Mr. Gundersen by scanning and comparing the bar codes on the wristband and the blood product. If the blood product is for Mr. Gundersen, then the blood transfusion can be performed. Otherwise, the blood transfusion procedure must be interrupted and Miss Olsen should contact the blood bank. During blood transfusion, the conditions of Mr. Gundersen, e.g., blood pressure, pulse, etc., can be recorded. If some reactions happen, the nurse must stop the transfusion and send relevant information to the blood bank.

3.2 Diagrammatic (Meta)modelling

With respect to tool support for diagrammatic modelling, the contribution of this thesis could be divided into three tasks: improving the storage format and the metamodel of DPF and adding the `Signature Editor`. The following subsections describe these tasks.

3.2.1 Storage Format

The DPF Workbench [236] had already have support for diagrammatic (meta) modelling and conformance checking of instances against their models. However, DPF specifications were stored as a binary format which contained structure, visualisation, constraints, and the predicates used to define these constraints. Mixing all this information in one single file leads to certain challenges, among them:

- the metamodel could not be updated, thus the smallest change in the metamodel would require creating the models from scratch.

- the signature could not be customised, thus only a few hard coded predicates could be supported.
- the concrete syntax of the specifications could not be customised, thus only a hard coded visualisation was supported.

To solve these challenges, we re-implemented most of the workbench and modularised the storage format by separating visualisation information and the signature from the DPF specifications. The following example illustrates these changes.

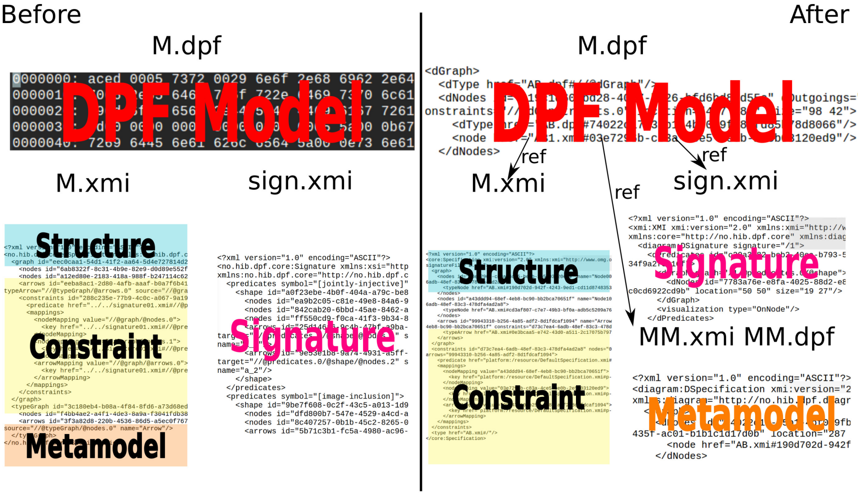


Figure 3.2: Comparison of the old and new storage formats

Example 6 (Modularisation of DPF specifications) *We create a metamodel MM and its instance, the model M, using both old and new DPF Workbench. A comparison of the storage formats of the two (meta)models is shown in Figure 3.2. In the old version of the workbench, the model M is stored in a binary file* `M.dpf`*. It contains all the information, including structure, constraints, visualisation and the metamodel. In addition, the workbench was generating two XMI files:* `M.xmi` *and* `sign.xmi`*. While* `M.xmi` *was used as a potential metamodel for creating instances of the model M,* `sign.xmi` *was only used for showing the predicates of the hard-coded signature; it was not possible to customise the* `sign.xmi` *file.*

In comparison, the model M in the new DPF Workbench is stored in different files. The visualisation information of the model M is now stored in `M.dpf` *while the structure and constraints are stored in the file* `M.xmi`*. In addition, the information of the metamodel MM is not stored in* `M.dpf` *and* `M.xmi` *anymore. Instead, we refer to the elements of the metamodel by using remote links. Thus,*

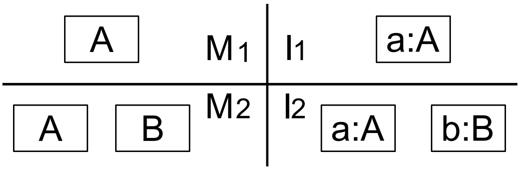


Figure 3.3: Model evolution support

when the metamodel MM is changed, the changes can be detected when the model M is opened next time. For instance, we have a model M_1 which contains a node **`A`**, *as shown in Figure 3.3. After creating an instance I_1 of the model $M1$, we add another element* **`B`** *in M_1 (we call the new model M_2). If we use the old version of the* `DPF Model Editor`*, we have to create I_1 from the scratch; the element* **`a:A`** *have to be created again. In contrast, after modularisation of DPF specifications, we can continue constructing the instance I, e.g., adding a new element* **`b:B`** *typed by* **`B`**, *without creating* **`a:A`**. *Furthermore, the signature stored in* `sign.xmi` *is now possible to be customised using the* `Signature Editor` *(see below).*

3.2.2 Signature Editor

Paper A presents the `DPF Model Editor`, which supports diagrammatic (meta)modelling in the DPF framework. The tool is implemented in Java and as a plugin for Eclipse. With this tool, model designers can construct models in a fully diagrammatic way: model structure is specified as a graph while constraints are formulated using pre-defined predicates from a diagrammatic signature. In the DPF Workbench, a default signature is provided, which consists of predicates that are common for object oriented modelling. In most cases, this set of predicates is not sufficient for expressing all requirements of the problem domain. Therefore, we developed the `Signature Editor` as a tool for extending this default signature and a support for domain specific constraints.

Before explaining the `Signature Editor`, we present the following example, which illustrates how the DPF Workbench is used to define a metamodelling hierarchy for the blood transfusion workflow from Section 3.1. Note that the details of how the DPF Workbench is used to create a metamodelling hierarchy are explained in Paper A. The general idea of the process is that, the `DPF Model Editor` generates an editor from a model and a customised signature, which in turn can be used to create instances of the model, as shown in Figure 3.4.

Example 7 (Metamodel for Blood Transfusion Workflow in DPF) *The blood transfusion workflow can be specified as a diagrammatic workflow model by using the* `DPF Model Editor`*, which is presented in Paper A. This is accomplished*

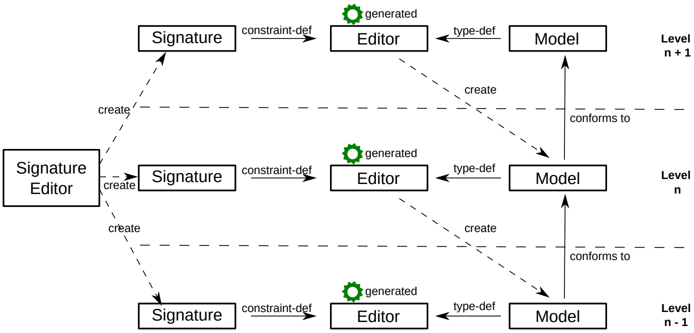


Figure 3.4: Metamodelling hierarchy in DPF

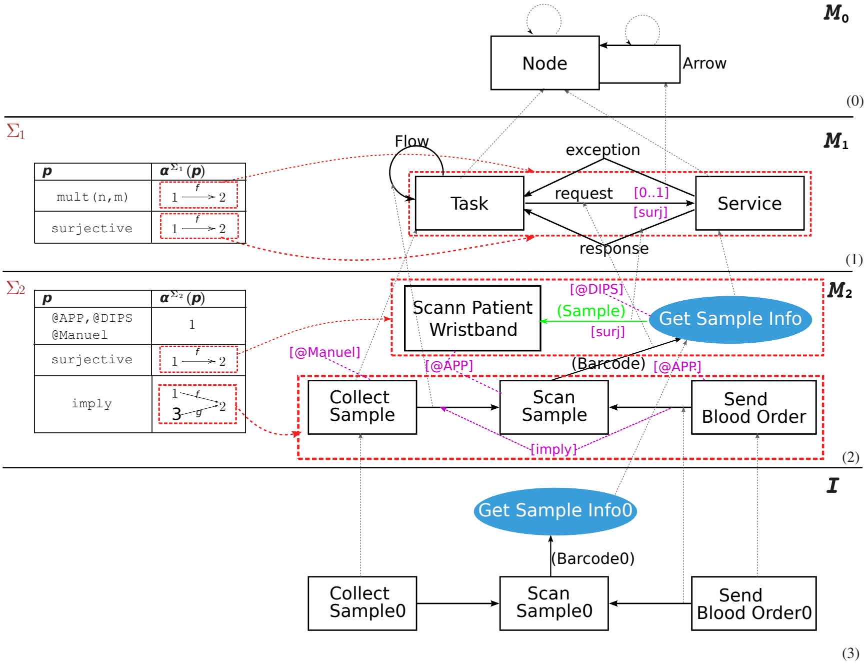


Figure 3.5: An excerpt of the blood transfusion workflow hierarchy

by constructing a modelling hierarchy which consists of 3 layers as shown in Figure 3.5. In order to specify the workflow model, we define a modelling language represented by the metamodel M_1 *at level* **1**. *Note that the definition of the metamodel is done only once and could be reused for the definition of other workflow models, such as admission of patients, assignment of doctors to patients, etc.*

The level **0** *shows the default metamodel* M_0 *of DPF Workbench. Recall that the DPF Workbench comes with a default top level metamodel* M_0 *(consisting*

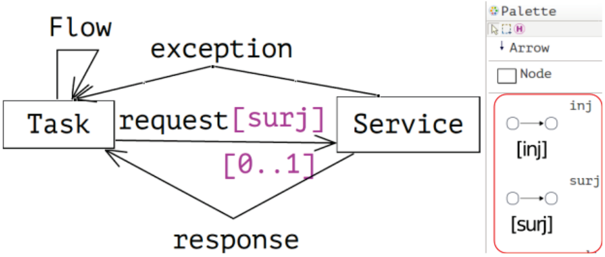


Figure 3.6: Workflow metamodel M_1 in the `DPF Model Editor`

*of **Node** and **Arrow**) and a default signature Σ_1 (consisting of predicates which are common for object oriented modelling). We use this metamodel to construct the workflow metamodel M_1 in the `DPF Model Editor` as shown in Figure 3.6. In M_1, we introduce the concepts **Task** and **Service**; relationships **Flow**, **request**, **response** and **exception**. The concepts are depicted as nodes while the relationships are depicted as arrows between the nodes. **Task** represents ordinary tasks, e.g., scanning the wristband of a patient. **Service** represents special tasks that require communication with other systems, e.g., scanning the bar code on the wristband of Mr. Gundersen requires communication with a Electronic Health Record (EHR) system called DIPS which stores information about patients. The **request** represents the data which is sent to a service; the **response** represents the data which is replied from a service while the **exception** represents the errors that may happen during executing the service.*

*In addition, we require that each **Service** must be requested by one **Task**, and that each **Task** can request at most one **Service**. We formulate these requirements as two constraints [surj] and [0..1] on the edge **request** by using the predicates from the default signature. The mapping between the arity of a predicate to the corresponding constraint is indicted by red dotted lines in Figure 3.5.*

In the `DPF Model Editor`, the typing relation between M_1 and M_0 is guaranteed by construction of model elements. This typing is depicted as dotted gray lines in Figure 3.5.

It should be mentioned that, when some model elements are selected, the applicable predicates will be shown in the `Palatte` area of the editor, highlighted in a red frame. The applicability of predicates is determined by the possibility of creating a graph homomorphism between the arity of the predicates to the selected model elements. Furthermore, the constraints are visualised with colour to distinguish with other structure elements[2].

Next we show how the metamodel in Example 7 is used to define a particular workflow model, in this case, the blood transfusion from Section 3.1. We will also demonstrate the use of the `Signature Editor`.

[2] In Paper A, the applicable predicates are shown in the toolbar while the constraints are in black.

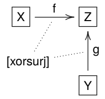
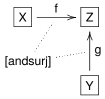
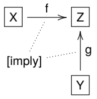
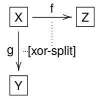
p	$\alpha^{\Sigma}(p)$	**Proposed Visualization**	**Semantic Interpretation**
@APP, @DIPS, @Manuel, @UniLab, @Printer	$\mathbf{1}$	$\boxed{\text{X}}$	For annotation use
xorsuj	$\begin{tikzcd} \mathbf{1} \arrow[r, "f"] & \mathbf{2} \\ & \mathbf{3} \arrow[u, "g"'] \end{tikzcd}$		$\forall z \in Z : e1 = \exists x \in X : f(x) = z, e2 = \exists x \in X : f(x) = z$ $(e1 \vee e1) \wedge \neg(e1 \wedge e2)$
andsuj	$\begin{tikzcd} \mathbf{1} \arrow[r, "f"] & \mathbf{2} \\ & \mathbf{3} \arrow[u, "g"'] \end{tikzcd}$		$\forall z \in Z : e1 = \exists x \in X : f(x) = z, e2 = \exists x \in X : f(x) = z$ $(e1 \wedge e1)$
imply	$\begin{tikzcd} \mathbf{1} \arrow[r, "f"] & \mathbf{2} \\ & \mathbf{3} \arrow[u, "g"'] \end{tikzcd}$		$\forall z \in Z : e1 = \exists x \in X : f(x) = z, e2 = \exists x \in X : f(x) = z$ $\neg e1 \vee e2$
xor-split	$\begin{tikzcd} \mathbf{1} \arrow[r, "f"] \arrow[d, "g"] & \mathbf{2} \\ \mathbf{3} & \end{tikzcd}$		$\forall x \in X : e1 = \exists z \in Z : f(x) = z, e2 = \exists y \in Y : f(x) = y$ $(e1 \vee e1) \wedge \neg(e1 \wedge e2)$

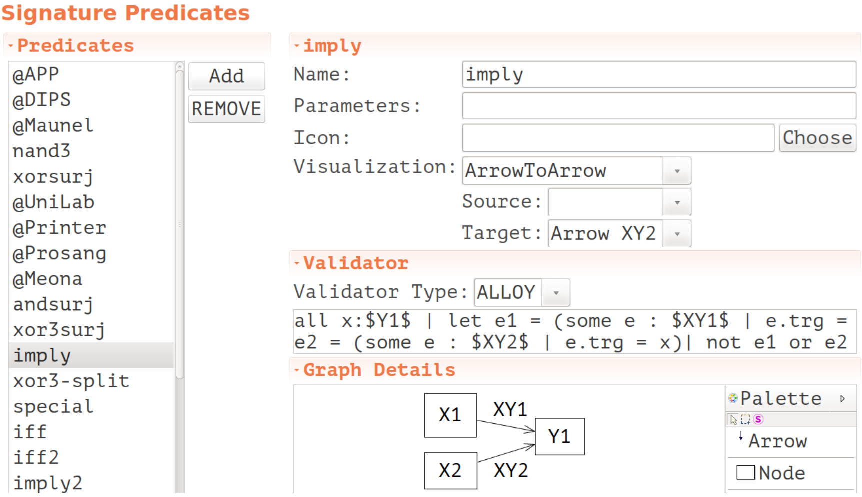
Table 3.1: The excerpt of signature $\mathbf{\Sigma_2}$

Example 8 (Modelling of Blood Transfusion Workflow in DPF) *With the workflow metamodel M_1 in hand, we are ready to specify the blood transfusion workflow as a workflow model. As in the previous example, the metamodel M_1 is used by the* `DPF Model Editor` *to specify the workflow model M_2. Note that for the sake of clarity, in Figure 3.5 we only show a part of the actual workflow model; the complete model is explained in Section 3.2.3.*

The predicates in the default signature are not enough for specifying all the requirements in the blood transfusion workflow. For example, the following requirements should be considered when the workflow is modelled.

1. *After a nurse logins the system, she can order blood or order sample, but not both*
2. *If a blood order is sent out exactly before sending out blood samples, then blood samples should be collected before sending out blood samples*
3. *If blood order is sent out after sending out sample order, it means that the nurse chooses to order sample after logining the system*
4. *If blood order is sent out after sending out samples, it means that the nurse chooses to order sample after logining the system*

The requirement 2 cannot be described by existing predicates in the default signature. Thus, we need new predicates to specify the blood transfusion workflow. Some of these predicates are shown in the user-defined signature $\mathbf{\Sigma_2}$ in Table 3.1.



Signature Predicates

Predicates

@APP
@DIPS
@Maunel
nand3
xorsurj
@UniLab
@Printer
@Prosang
@Meona
andsurj
xor3surj
imply
xor3-split
special
iff
iff2
imply2

imply

Name: imply
Parameters:
Icon:
Visualization: ArrowToArrow
Source:
Target: Arrow XY2

Validator

Validator Type: ALLOY

```
all x:$Y1$ | let e1 = (some e : $XY1$ | e.trg =
e2 = (some e : $XY2$ | e.trg = x)| not e1 or e2
```

Graph Details

Figure 3.7: Workflow signature Σ_2 in the `Signature Editor`

For example, the requirement 2 can be specified as a constraint by using the predicate imply*. We use the* `Signature Editor` *to define the predicates, e.g., the predicate* imply *as shown in Figure 3.7. We specify the name, the arity $\alpha^{\Sigma}(p)$, and semantics of predicates. The arity is shown in the* `Graph Details` *area while the semantics is shown in the* `Validator` *area. The semantics can be specified in Java, Alloy or OCL. Here, we specify the semantics in Alloy and use it for verification purpose in the sequel.*

Notice that the predicates whose names start with @ are used for annotation purposes. Such a predicate denotes the system responsible to perform a task. For example, as shown in Figure 3.5, we use [@APP] on **Scan Sample** and **Send Blood Order**; these tasks are performed on application. Furthermore, the service **Get Sample Info** is annotated with [@DIPS]; this means that the service is performed on the EHR system DIPS.

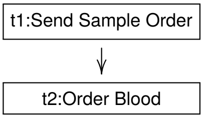
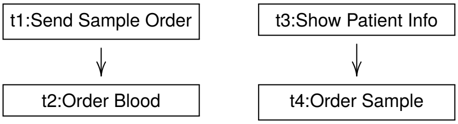
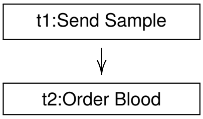
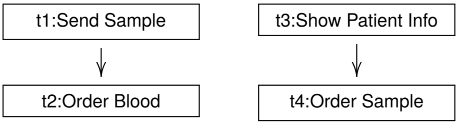
Note that due to the modularisation of the storage format explained in Section 3.2.1, it is possible to visualise elements in M_2 differently, e.g., services are visualised as blue ellipses and tasks are visualised as rectangles. This is accomplished by assigning elements in M_1 with concrete syntax configurations. We have implemented this configuration mechanism as an Eclipse plugin. The plugin defines some interfaces that users can implement to draw their own nodes and arrows. However, the interfaces are highly dependent on the underlying implementation framework, Graphical Editing Framework (GEF) [237].

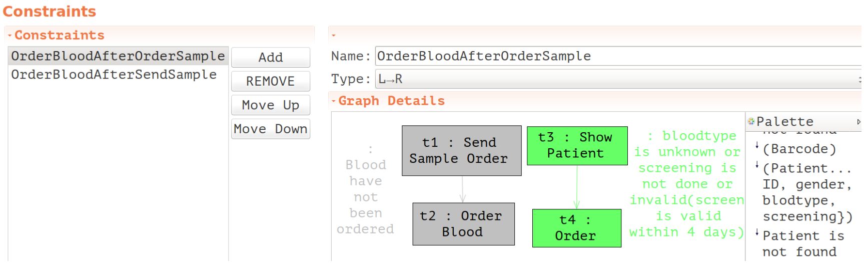
Graph Constraints

In addition to constraints based on DPF predicates, graph constraints may be used to define dependencies among constraints and/or the structures of a model [55]. Given a model, a graph constraint $\boldsymbol{N} \xleftarrow{n} \boldsymbol{L} \xrightarrow{u} \boldsymbol{R}$ consists of three graphs: left $\boldsymbol{L}$, right $\boldsymbol{R}$ and application condition $\boldsymbol{N}$ (PAC or NAC); and two injective graph homomorphisms $\boldsymbol{n}$ and $\boldsymbol{u}$ (see [55, 77]). The components $\boldsymbol{L}$, $\boldsymbol{R}$ and $\boldsymbol{N}$ are graphs typed by the underlying graph of the model.

Example 9 (Graph Constraints) *For example, the requirements 3 and 4 can be specified as two graph constraints respectively, as shown in Table 3.2. The components $\boldsymbol{L}$, $\boldsymbol{R}$ and $\boldsymbol{N}$ are graphs typed by the underlying graph $\boldsymbol{S}$, denoted by $\boldsymbol{L}:\boldsymbol{S}$, $\boldsymbol{R}:\boldsymbol{S}$ and $\boldsymbol{N}:\boldsymbol{S}$ in the table.*

Table 3.2: Graph constraints of the blood transfusion model M_2

$\boldsymbol{N}:\boldsymbol{S}$	$\boldsymbol{L}:\boldsymbol{S}$	$\boldsymbol{R}:\boldsymbol{S}$
OrderBloodAfterOrderSample		
		 t1:Send Sample Order t2:Order Blood t3:Show Patient Info t4:Order Sample
OrderBloodAfterSendSample		
		 t1:Send Sample t2:Order Blood t3:Show Patient Info t4:Order Sample



Constraints

Constraints

OrderBloodAfterOrderSample
OrderBloodAfterSendSample

Add
REMOVE
Move Up
Move Down

Name: OrderBloodAfterOrderSample

Type: L→R

Graph Details

: Blood have not been ordered
t1 : Send Sample Order
t2 : Order Blood
t3 : Show Patient
t4 : Order
: bloodtype is unknown or screening is not done or invalid(screen is valid within 4 days)
Palette
not found
(Barcode)
(Patient... ID, gender, blodtype, screening})
Patient is not found

Figure 3.8: Graph constraints of M_2 in the `Universal Constraint Editor`

In the DPF workbench, we have designed an editor to specify graph constraints (see Figure 3.8). The Constraints field lists all the graph constraints for a model while the right part of the editor shows the details for the selected graph constraint. Since we assume that both n and u are injective, inspired by Henshin [238], the graph constraints are specified with the following colour coding: red elements belong to N minus L; green elements belong to R minus L; and gray elements belong to L. In other words, N is the sum of gray and red elements; R is the sum of gray and green elements; and L is the gray elements.

Remark 1 *Note that in DPF, graph constraints are generalised as universal constraints [239] such that L and R are DPF specifications instead of graphs. However, in this thesis, we only consider classical graph constraints and leave the case with universal constraints to future work.*

3.2.3 The Running Example: Revisited

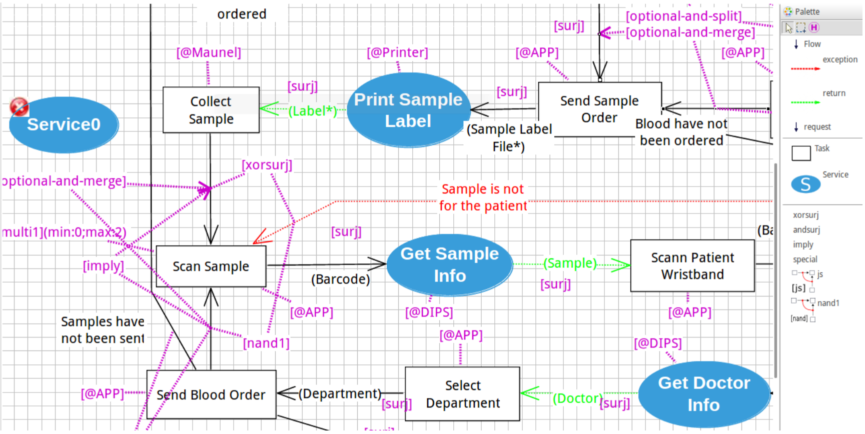


Figure 3.9: Blood transfusion workflow M_2 in the `DPF Model Editor`

In the previous sections we illustrated how the DPF Workbench was used to define the metamodel M_1. Moreover, we showed how the `Signature Editor` was used to extend the default DPF signature. In this section we will revisit the Blood Transfusion Workflow form Section 3.1 and demonstrate how the DPF Workbench is used to model the workflow.

As Figure 3.4 depicts, the metamodel M_1 and the signature Σ_2 are used to generate a model editor (see Figure 3.9), which in turn is used to construct the model M_2. In the figure, we only show part of M_2 while the complete model M_2 is shown in Figure 3.10. In the editor, the concepts and relationships in M_1 are presented as types in the `Palette` area. These types are used to create elements in M_2 which are typed by elements in M_1. The typing relation between M_1 and M_2 are guaranteed by creation. For example, as shown in Figure 3.9, we create **Scan Sample**, **Send Blood Order** and **Collect Sample** that are typed by **Task**, and arrows between them which are typed by **Flow**. We also create **Get Sample Info** typed by **Service** representing a service; the arrow **Scan Sample** → **Get Sample Info** typed by **request** representing the request of the service by sending the barcode of sample; the arrow **Get Sample Info** → **Scann Patient Wristband** typed by **response** representing receiving the information of a sample.

In addition to structural information, there are many constraints which are used to specify the requirements of the blood transfusion workflow. For simplicity, we request that a workflow run does not contain two starting tasks, i.e., the tasks that have no preceding tasks. Therefore, we have the multiplicity constraint [multi] (min:0;max:1) on **Init** as shown in Figure 3.10. Moreover, on every arrow $S \xrightarrow{E} T$, there is a constraint [0..1] (not shown in Figure 3.10). It states that, on the next level, for each s typed by S, there is at most one outgoing arrow e typed by E. These constraints are used to specify that, after each task (or service), at most one task (or service) of the consecutive type is running (or requested) in the next step. For example, the constraint [0..1] on the arrow **Scan Nurse Card** → **Get Nurse Info** states that, after **Scan Nurse Card**, at most one service **Get Nurse Info** is requested. Furthermore, in the blood transfusion workflow, the sequence of sending sample and sending blood order does not matter; nurse can send sample first and send blood order afterwards, or vice versa. In order to enable both of these flows, we create flows **Send Sample** → **Order Blood** and **Send Blood Order** → **Order Sample**. Since we require that a nurse can only send sample and blood order once, we specify the constraints [xorsurj] and [xor3surj] on the incoming flows of **Order Sample** and **Order Blood** to avoid the two tasks are performed more than twice.

The conformance between M_1 and M_2 is guaranteed by first checking that the elements in M_1 are correctly typed by the elements in M_2 and then checking whether all constraints in M_2 are respected by M_1 (formally, existence of a graph homomorphism from M_1 and M_2 which satisfies all constraints presented in M_2). If some constraints are violated, the elements that cause the violation will be marked as an error. For example, the node **Service0** in Figure 3.9 violates the constraint [surj] on the arrow **request** in M_1, therefore, the node is marked as an error. Furthermore, the predicates in Σ_2 are also available in the `Palette` and are used to formulate constraints on the model M_2.

Remark 2 *Notice that, the workflow model M_2 only describes the structural information of the blood transfusion workflow, e.g., the tasks included in the workflow, the order among the tasks, their relations to services, etc. It does not contain information about the dynamic behavior of the workflow model that concerns about the transition among the states in a workflow run. On the fourth layer of the hierarchy in Figure 3.5, we show an excerpt of an instance of the workflow model M_2. In Section 3.3.2, we will use model transformation rules to specify its dynamic*

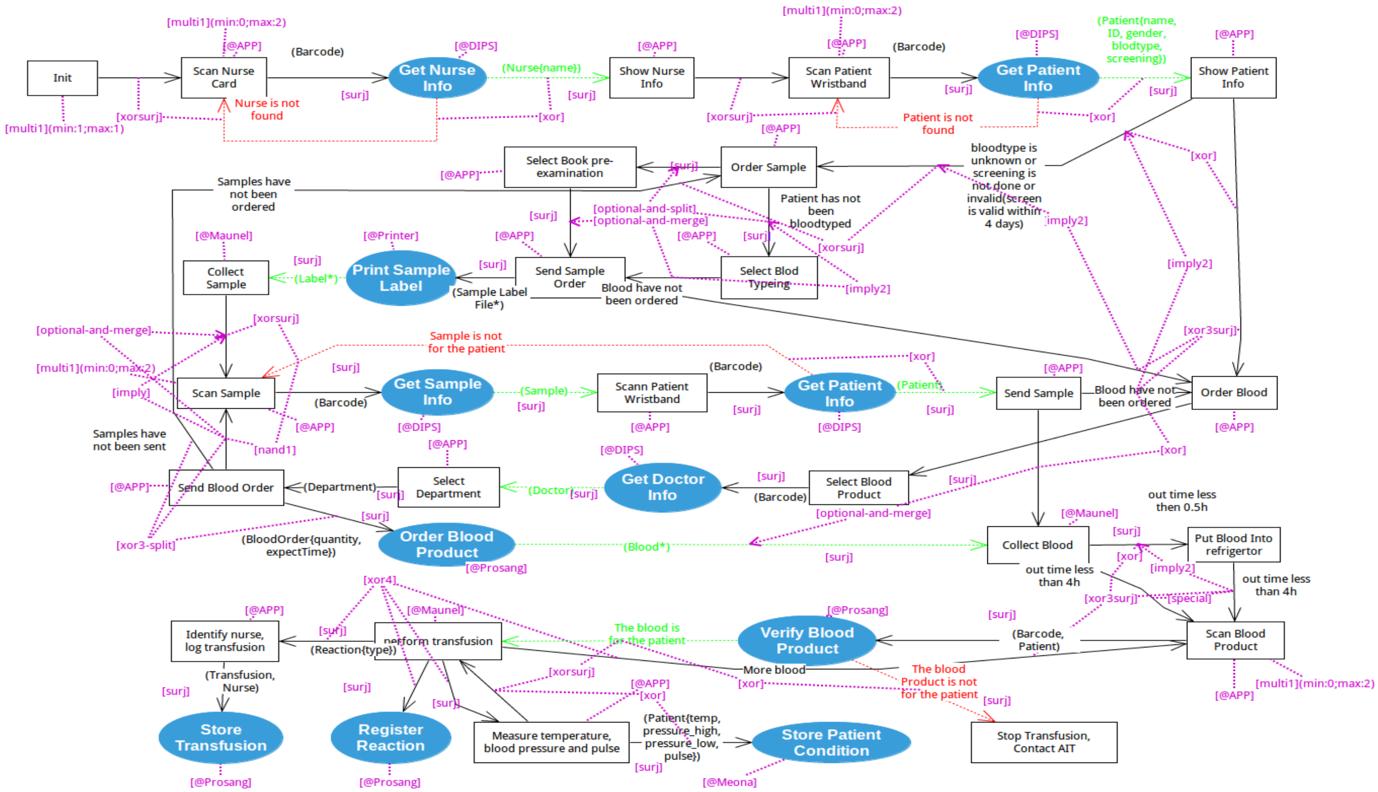


Figure 3.10: Blood transfusion workflow model M_2

behavior and verify its dynamic behavior to demonstrate another contribution of the thesis, namely, verification of model transformations.

Remark 3 *In M_2, some edges are named in special format. These names are used to generate service specifications from health workflow. For example, the* ***request Scan Nurse Card*** $\longrightarrow$ ***Get Nurse Info*** *is named as* `(Barcode)`*; the* ***response Get Nurse Info*** $\longrightarrow$ ***Show Nurse Info*** *is named as* `(Nurse{name})`*. These names are used to generate a service specification for the* ***Service Get Nurse Info****: the input is a* ***Barcode*** *and the output is the* `name` *of a* `Nurse`*. It may throw an exception* `Nurse is not found` *if error happens.*

3.3 Verification

In the previous sections, we presented the tool support for (meta)modelling in DPF. We will now illustrate the last two contributions of the thesis which focus on verification of models and model transformations. Since both of the contributions use Alloy as the underlying verification technique, we will first give a brief introduction to Alloy.

Alloy consists of a structural modelling language and a tool to analyse specifications. The modelling language is a declarative textual language, suited for describing complex model structures and constraints based on relational logic. Model analysis is performed by a constraint solver called Alloy Analyzer. It analyses a specification by first translating it into a SAT problem and then solving the problem by using some off-the-shelf SAT solvers, e.g., SAT4J [109]. The analyzer verifies whether a specification satisfies or violates a property by searching for instances (or counterexamples) which satisfy (or violate) the property. The Alloy Analyzer uses a bounded verification approach; it finds the instances or counterexamples within a search space which is determined by a user-defined scope. Bounded verification approaches can promise automation and termination. However, as a side effect, the verification with Alloy is incomplete, i.e., it cannot guarantee that a model does not satisfy (violate) a property if no instance within a search space satisfies (violates) the property. In addition, the verification approach encounters the scalability problem. It means that, when complex specifications (which consist of large number of concepts and relationships) are verified within a large scope, the verification may take quite long time or become intractable [97].

3.3.1 Verification of Models

In addition to tool support for (meta)modelling, we have provided tool support for verification of models. This is convenient for model designers, since they may want to know whether some constraints cause contradiction, or whether there exist redundant constraints; i.e. whether a constraint is already implied by or can be induced from other constraints. These features are especially crucial when models become large like the blood transfusion workflow model in Figure 3.10. Thus, it is necessary to provide verification functionality to assist model designers. Motivated by this requirement, the second contribution of the thesis presented in Paper B, focuses on a verification approach using Alloy. It includes three parts:

- encoding DPF models in Alloy,
- presenting verification result in Alloy to feedback in DPF, and
- providing an optimization technique.

```
//Signatures of nodes
sig NInit{}
sig NScanNurseCard{}
sig NGetNurseInfo{}
sig NScannPatientWristband{}
sig NGetSampleInfo{}
sig NShowPatientInfo{}
...

//Signatures of edges
sig ENInitNScanNurseCard{src:one NInit, trg:one NScanNurseCard}
sig EBarcode6{src:one NScanNurseCard, trg:one NGetNurseInfo}
sig ENurseisnotfound{src:one NGetNurseInfo, trg:one NScanNurseCard}
sig ENursename{src:one NGetNurseInfo, trg:one NShowNurseInfo}
sig EPatientisnotfound{src:one NGetPatientInfo, trg:one NScanPatientWristband}
sig ENShowNurseInfoNScanPatientWristband{src:one NShowNurseInfo, trg:one NScanPatientWristband}
...
```

Listing 3.1: Alloy Signatures for the workflow model M_2

Encoding of DPF Models

There is a formalisation difference between DPF and Alloy. DPF uses a diagrammatic language to specify models while Alloy is a declarative textual language for structural modelling. In order to verify DPF models using Alloy, we construct an automatic encoding of DPF models as Alloy specifications. Given a DPF model, its nodes and arrows are translated as node signatures and arrow signatures, respectively. Node signatures have no field and arrow signatures have two fields: `src` and `trg` representing the source and target nodes of the arrow, respectively.

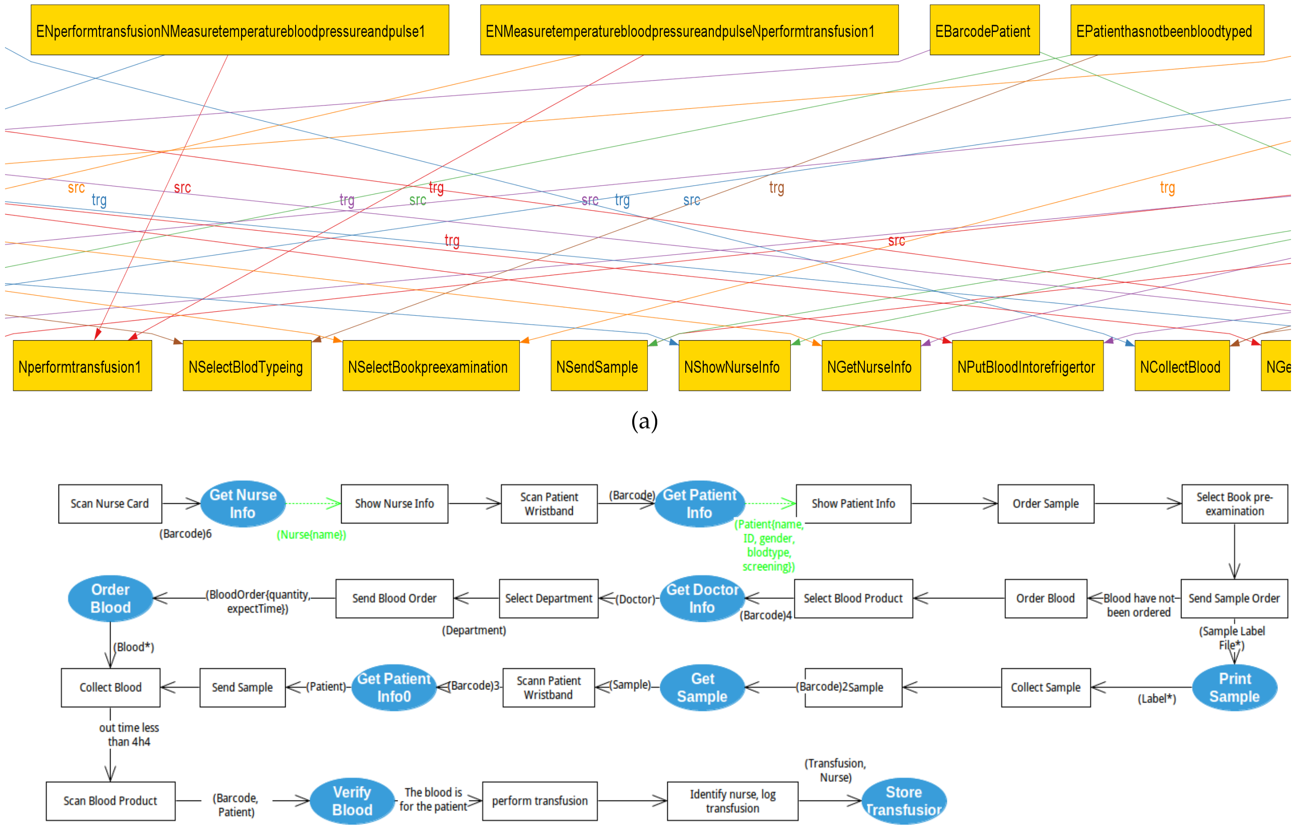
Example 10 (Encoding of the structure of M_2) *Recall the running example in Section 3.1. The structure of the workflow model M_2 is translated into the signatures in Listing 3.1. The nodes, e.g.,* **Init***,* **Scan Nurse Card***, etc, are encoded as node signatures on line 2-8. While the arrows, e.g.,* **Init⟶Scan Nurse Card***, etc, are encoded as edge signatures on line 11-17.*

```
1  //The definition of surjective predicate
2  fact surj_$XY${
3  all n:($Y$)| some e:($XY$)| e.trg=n
4  }
5  //surjective on (Barcode):Scan Patient Wristband->Get Patient Info
6  fact surj_EBarcode{
7  all n:(NGetPatientInfo)| some e:(EBarcode)| e.trg=n
8  }
9  //surjective on :Show Nurse Info->Scan Patient Wristband
10 fact surj_ENShowNurseInfoNScanPatientWristband{
11 all n:(NScanPatientWristband)| some e:(
       ENShowNurseInfoNScanPatientWristband)| e.trg=n
12 }
```

Listing 3.2: Alloy Facts for the workflow model M_2

The constraints in DPF models are encoded as `fact`s when verifying consistency, while they are encoded as `pred`s when searching for redundant constraints. The encoding of constraints is based on the semantics of predicates; the semantics of a predicate p is defined as a parameterised Alloy fact in the `Signature Editor` where the parameters arg are the elements in the arity of the predicate (see Figure 3.9). Given a constraint which is formulated based on a predicate p, the constraint can be encoded as a fact by substituting the parameters arg with the Alloy signature name of $\delta(arg)$, where δ is the mapping from the arity of the predicate to the structure of the model (see Section 1.2.3). The following example shows how the constraints based on predicates are encoded as facts in Alloy.

Example 11 (Encoding of constraints of M_2) *The semantics of the predicate* surjective *is defined as the expression on Line 2-4 in Listing 3.2. Two* [surj] *constraints in the workflow model M_2 are encoded as facts on line 5-12 in Listing 3.2 when verifying consistency. The* [surj] *on the arrow* **Scan Patient Wristband⟶Get Patient Info** *can be encoded as a fact on line 5-8 by replacing* `$XY$` *with* `EBarcode` *and* `$Y$` *with* `NGetPatientInfo`*.*



(a)

(b)

Figure 3.11: Part of an instance of M_2 represented in Alloy (a) and in DPF (b)

Presenting Verification Results in DPF

After DPF models are encoded as Alloy specifications, we can check the specifications by using the Alloy Analyzer to verify properties, e.g., consistency and lack of redundant constraints. However, the verification results in Alloy are presented as relations. To understand these results, model designers have to translate them into some representation in the design space. The representation difference between the design space and the verification space is another gap that our verification approach tries to bridge, as shown in Figure 3.12. We translate the verification results of Alloy as feedback in the DPF Workbench, hence the design choice of integrating the verification approach with the `DPF Model Editor` and the hiding of the underlying verification in Alloy.

If a model is verified consistent, an arbitrary instance will be produced by the Alloy Analyzer. We translate this instance to a DPF instance and present it as feedback in `DPF Model Editor`.

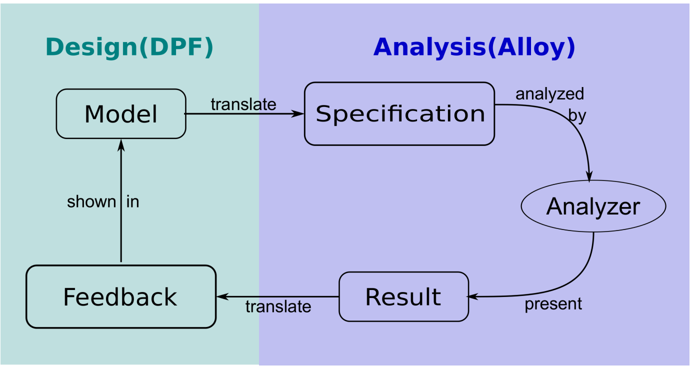


Figure 3.12: Bridge the gap between modelling and verification

Example 12 (Checking Consistency) *The blood transfusion workflow model M_2 (see Figure 3.10) is checked to be consistent, thus the Alloy Analyzer produces an arbitrary instance of M_2 in Alloy as shown in Figure 3.11a. The instance in Alloy is quite large, thus we just show part of the instance. Its corresponding instance in DPF is shown in Figure 3.11b.*

It should be mentioned that, if no instance of a model can be found, it means that there are contradictory constraints in the model. Alloy can collect a set of expressions which cause contradiction. In this approach, we use this information to find the corresponding constraints in DPF and highlight them. Thus, the model designers can also see the verification result in their domain or find the problematic part of the model quickly by using the feedback.

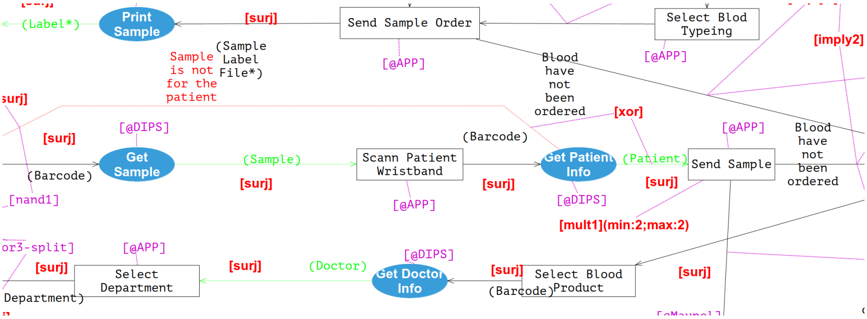


Figure 3.13: Highlighting the contradictory constraints in M_2

Example 13 (Result of Contradiction) *Assume that the model designer erroneously adds a constraint [mult1] (min:2, max:2) on task* ***Send Sample****. If we now check the consistency of M_2, the Alloy Analyzer can not find an instance of the workflow model. This indicates that the model is not consistent anymore due to contradiction of the multiplicity constraint with some other constraints. In this case, we will highlight the constraints that contradict to the multiplicity constraint on* ***Send Sample*** *as shown in Figure 3.13, and the model designer can use this as a guideline to fix the problem.*

Remark 4 (Consistency checking as property verification) *Note that the consistency checking mechanism above could also be used for property verification. For example in the model M_2, we require that the task* ***Send Sample*** *can be performed at most once. To verify this property, we add a multiplicity constraint [mult1] (min:2, max:2) on task* ***Send Sample*** *for verification intention. If we can find an instance of the model, it means that there is a workflow run in which* ***Send Sample*** *is performed twice. Otherwise, no such workflow run exists. In this case, we will highlight the constraints that contradict with the multiplicity constraint on* ***Send Sample*** *as shown in Figure 3.13.*

In Paper B, we also showed how to find redundant constraints. The process of finding redundant constraints is similar to the process of checking consistency; DPF models (structure and constraints) are encoded as Alloy specification; then an Alloy `run` command is used to execute the verification. However, there exists two fundamental differences:

- constraints are encoded as `preds`. These `preds` are used in the `run` command to check whether a constraint is redundant. Given a set of constraints $c_1, c_2, \ldots, c_n$, they are encoded as `preds` $p_{c_1}, p_{c_2}, \ldots, p_{c_n}$. We use the command `run`{p_{c_1} `and` p_{c_2} `and` $\ldots$ `and not` p_{c_i} `and` $\ldots$ `and` p_{c_n}} to check whether a constraint c_i where $1 \leq i \leq n$ is redundant. The command is used to find an instance which satisfies

$c_1, c_2, \ldots, c_n$ but violates c_i. If such an instance is found, it means that the constraint c_n cannot be induced from other constraints. Thus the constraint is not redundant. Otherwise, we can claim that the constraint is redundant.

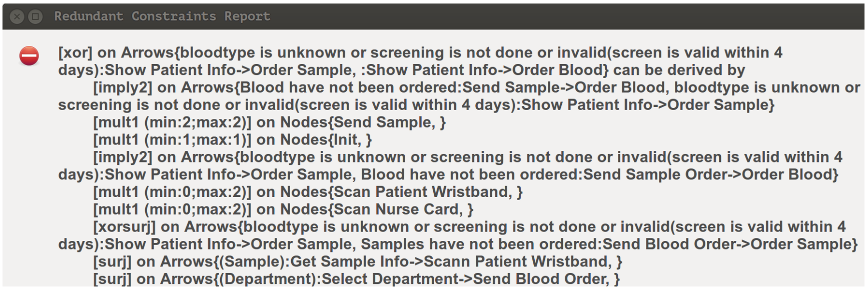
- we will list all the redundant constraints of a model in a dialogbox. In addition, for each redundant constraint c, we will show which constraints can induce c.

In the following example, we will use the blood transfusion workflow model M_2 to demonstrate the above mentioned differences.

```
 1 pred xor_Ebloodtypeisunknownorscreeningisnotdoneorinvalidscreen
 2 isvalidwithin4days_ENShowPatientInfoNOrderBlood[]{
 3     //XOR constraint between Show Patient Info->Order Sample and :
           Show Patient Info->Order Blood
 4     all n:(NShowPatientInfo)|let e1=(some e:
           Ebloodtypeisunknownorscreeningisnotdoneorinvalidscreenis
 5     validwithin4days|e.src=n), e2=(some e:
           ENShowPatientInfoNOrderBlood|e.src=n)|(e1 or e2) and not(e1
           and e2)
 6 }
 7
 8 run{not xor_Ebloodtypeisunknownorscreeningisnotdoneorinvalidscreen
 9 isvalidwithin4days_ENShowPatientInfoNOrderBlood[] and
       mult1_ENperformtransfusionNIdentifynurselogtransfusion[] and
       xor3surj_Eouttimelessthan4h_EMoreblood_Eouttimelessthan4h6[] and
       ... and imply_ENCollectSampleNScanSample_ESampleshavenotbeensent
       []} for 3
10
11 [xor] on Arrows{:Show Patient Info->Order Sample, :Show Patient Info
       ->Order Blood} can be induced by
12     [imply2] on Arrows{:Send Sample->Order Blood, :Show Patient Info
           ->Order Sample}
13     ...
14     [surj] on Arrows{(Doctor):Get Doctor Info->Select Department, }
15     [xor] on Arrows{(Patient):Get Patient Info->Send Sample, :Get
           Patient Info->Scan Sample}
16     [surj] on Arrows{(Patient):Get Patient Info->Send Sample, }
17     [surj] on Arrows{(Label*):Print Sample Label->Collect Sample, }
```

Listing 3.3: Alloy Preds for the workflow model M_2

Example 14 (Finding redundant constraints in M_2) *When we find redundant constraints in M_2, all the constraints are encoded as* `preds`*. For example, the constraint* [xor] *on the arrows* ***Show Patient Info*** $\longrightarrow$ ***Order Sample*** *and* ***Show Patient Info*** $\longrightarrow$ ***Order Blood*** *is encoded as the* `pred` *as shown on lines 1-6 in Listing 3.3. The* `preds` *can be used in the* `run` *command, e.g., on line 9 to check whether the constraint* [xor] *is redundant. The constraint* [xor] *is checked to be redundant. In this case, there will be a dialogbox as shown in Figure 3.14 displaying messages about*



```
[xor] on Arrows{bloodtype is unknown or screening is not done or invalid(screen is valid within 4
days):Show Patient Info->Order Sample, :Show Patient Info->Order Blood} can be derived by
      [imply2] on Arrows{Blood have not been ordered:Send Sample->Order Blood, bloodtype is unknown or
screening is not done or invalid(screen is valid within 4 days):Show Patient Info->Order Sample}
      [mult1 (min:2;max:2)] on Nodes{Send Sample, }
      [mult1 (min:1;max:1)] on Nodes{Init, }
      [imply2] on Arrows{bloodtype is unknown or screening is not done or invalid(screen is valid within 4
days):Show Patient Info->Order Sample, Blood have not been ordered:Send Sample Order->Order Blood}
      [mult1 (min:0;max:2)] on Nodes{Scan Patient Wristband, }
      [mult1 (min:0;max:2)] on Nodes{Scan Nurse Card, }
      [xorsurj] on Arrows{bloodtype is unknown or screening is not done or invalid(screen is valid within 4
days):Show Patient Info->Order Sample, Samples have not been ordered:Send Blood Order->Order Sample}
      [surj] on Arrows{(Sample):Get Sample Info->Scann Patient Wristband, }
      [surj] on Arrows{(Department):Select Department->Send Blood Order, }
```

Figure 3.14: The feedback for redundant constraints

which constraints can induce this constraint. The message for the [xor] *is shown on line 11-17 in Listing 3.3.*

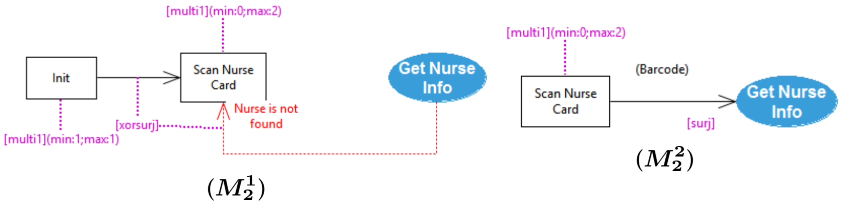
Remark 5 *Notice that, in the dialogbox in Figure 3.14, we will list all the redundant constraints. Since these redundant constraints may depend on each other, we cannot simply delete all the constraints to remove redundant informations. The model designer is responsible to figure out which constraints should be removed or kept.*

Optimization Technique

As mentioned, verification with Alloy encounters the scalability problem. In order to tackle this problem, we introduce a *model partition* technique (see Paper B). We use this technique to reduce the verification of a model into the verification of its submodel. The technique can be applied to verification of properties which can be expressed as graph formula in First-Order Logic (FOL) [88]. In this part, we will demonstrate the technique by showing how to reduce the consistency checking of $\boldsymbol{M_2}$ to the consistency checking of a submodel of $\boldsymbol{M_2}$.

Remark 6 *We have also developed another optimisation technique using scope graphs based on the syntax of the constraints to determine the maximum scope needed by Alloy [240]. This technique works in practise, however, the formal proof is left for future work.*

A model can be split into submodels based on the *factors* of the constraints, i.e., the model elements which are affected by the constraints. It means that if two elements are contained in the factor of a constraint, they cannot belong to different submodels. Since the splitting technique is used for verification, we do not need to consider the annotations (e.g., [@APP] on **Scan Nurse Card**). For example, some submodels of the model $\boldsymbol{M_2}$ are



(M_2^1) (M_2^2)

Figure 3.15: Some submodels of M_2

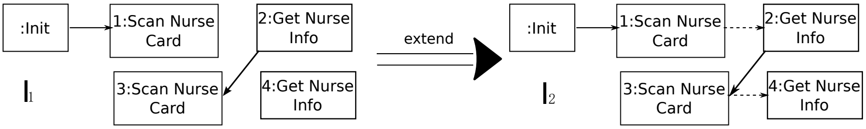


Figure 3.16: Extension of instance of submodels

shown in Figure 3.15. Notice that, we will have a hierarchy of submodels, i.e., some submodels may be split to other submodels. For example, $M_2^1 \cup M_2^2$ is a submodel which can be split into M_2^1 and M_2^2.

Within these submodels, we are interested in *left-total* submodels, i.e. the submodels of which every instance may be extended as instances of the whole model. Given an instance of a submodel, the instance can be extended by adding elements typed the types in other submodels. If a model M can be split into two submodels M' and M'' where $M' \cup M'' = M$, an instance of M' can be extended by adding elements typed by the types in $M'' - M'$.

Example 15 (Extension of Instance) *Consider $M_2^1 \cup M_2^2$ in Figure 3.15 as a model M. M_2^1 and M_2^2 are the two submodels of M. I_1 in Figure 3.16 is an instance of M_2^1. An extension of I_1 can be obtained by adding elements typed by* **Scan Nurse Card** $\xrightarrow{\textbf{(Barcode)}}$ **Get Nurse Info** *which is $M_2^2 - M_2^1$. I_1 can be extended to I_2 in Figure 3.16 which is an instance of M by adding two arrows typed by the arrow* **(Barcode)***. The two added arrows are depicted in dashed arrows. However, if I_1 has three elements typed by* **Get Nurse Info***, it can never be extended to an instance of M. This is because each service* **Get Nurse Info** *must be requested by a task* **Scan Nurse Card** *([surj] on the arrow* **(Barcode)***) while each task* **Scan Nurse Card** *can request at most one service* **Get Nurse Info** *([0..1] on the arrow* **(Barcode)***). In contrast, every instance of M_2^2 can be extended as an instance of M. Thus, M_2^2 is a left-total submodel of M.*

In order to find left-total submodels, we outline an approach based on *forbidden patterns* of constraints. Given a constraint c on a structure S, a

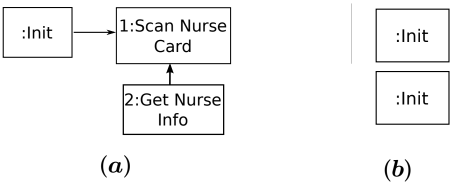


Figure 3.17: Forbidden patterns

forbidden pattern of c is a graph FP which is well-typed by S but violates c. In addition, any graph containing the graph FP violates c.

Example 16 (Forbidden Pattern) *For the constraint* [xorsuj] *on the arrows* **Init → Scan Nurse Card** *and* **Get Nurse Info → Scan Nurse Card***, a forbidden pattern of* [xorsuj] *is shown in Figure 3.17**a**. For the constraint* [mult1] (min:1;max:1)*, its forbidden pattern is shown in Figure 3.17**b**.*

In Paper B, we showed that, if a model M can be split into two submodels M' and M'' where $M' \cup M'' = M$ and the intersection of their structures (denoted as $S' \cap S''$) contains no more than one node, M' is a left-total submodel if

1. M' is not consistent, otherwise,
2. M'' is consistent and every forbidden pattern of M' which is typed by the intersection of M' and M'' is also a forbidden pattern of M''.

```
1 Model findLeftTotalSubmodel(Model $M$){
2     for each split of $M$ where $M = M' \cup M''$
3         $S'$=structure($M'$)
4         $S''$=structure($M''$)
5         if(consistent($M'$) && ! hasFPTypedBy($M'$, $S' \cap S''$))
6             return findLeftTotalSubmodel($M''$);
7     return $M$;
8 }
```

Listing 3.4: An algorithm to find left-total submodels

In other words, if the submodel M' is consistent and has no such forbidden pattern that is typed by $S' \cap S''$, then the submodel M'' is a left-total model. Based on this observation, we present an algorithm in Listing 3.4 to find left-total submodels. We try to find such a submodel M' (line 2-6). If we find the M' successfully, then M'' can be used as a left-total submodel. In order to find smaller sized left-total submodel, we invoke the algorithm again on M'' (line 6). If we cannot find such a M', then we just return the model itself (line 7).

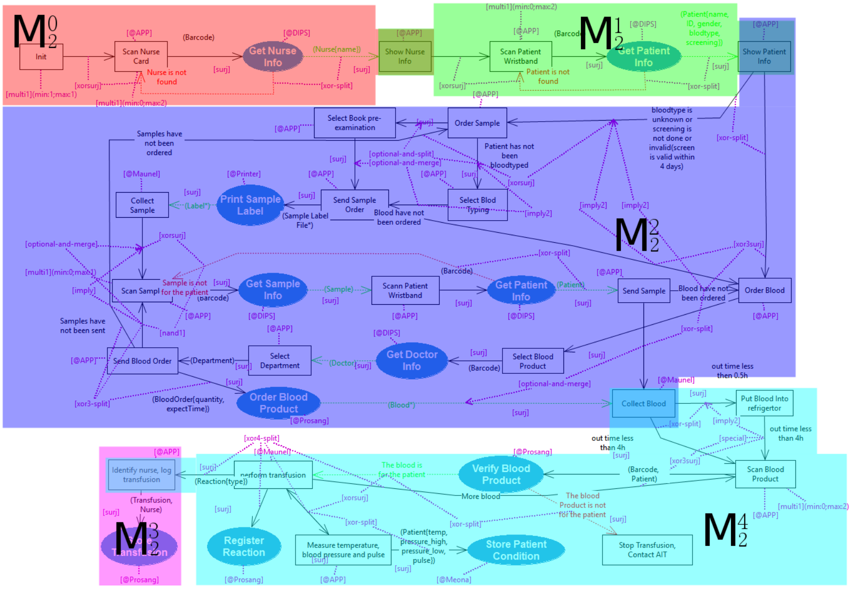


Figure 3.18: Submodels in M_2

Example 17 (Finding left-total submodels of M_2) *Based on the algorithm in Listing 3.4, we can find the submodel of M_2 in Figure 3.10. For simplicity, we only consider the M' and M'' where the intersection structure of the two submodels, $S' \cap S''$, contains no more than one node. The result is shown in Figure 3.18. First, we find $M' = M_2^3$ and $M'' = M_2^0 \cup M_2^1 \cup M_2^2 \cup M_2^4$ since M_2^3 is consistent and has no forbidden pattern typed by* **Identify nurse, log transfusion***. Then we can further find the submodel of M'' by identifying $M' = M_2^4$ and $M'' = M_2^0 \cup M_2^1 \cup M_2^2$ since M_2^4 is consistent and has no forbidden pattern typed by* **Collet Blood***. We invoke the algorithm to find the submodel of M'' and find that it has no left-total submodels. M_2^0 and M_2^1 cannot be used as M' even they are consistent, since they have forbidden patterns. For example, M_2^0 has a forbidden pattern which contains two tasks typed by* **Show Nurse Info***. The forbidden pattern is typed by* **Show Nurse Info***, which is the intersection of the M_2^0 with other submodels. Thus, the consistency verification of M_2 can be reduced to the verification of $M_2^0 \cup M_2^1 \cup M_2^2$.*

We have proven that the verification of a model can be reduced to the verification of its left-total submodels in Paper B. The splitting technique can alleviate the scalability problem.

Table 3.3: Experiment result for splitting technique

Scope	$\boldsymbol{M_2}$			$\boldsymbol{M_2^0 \cup M_2^1 \cup M_2^2}$		
	T	V	T+V	T	V	T+V
3	99	82	181	68	73	141
4	205	156	361	92	111	203
5	208	442	650	149	260	409
6	261	582	843	132	478	610
7	384	788	1172	183	667	850
8	402	1035	1437	265	866	1131
9	695	1756	2451	310	1382	1692
10	645	2159	2804	378	2062	2440
11	968	3904	4872	534	2756	3290

T: Translation Time; V: Verification Time

Example 18 (Experimental Result) *Table 3.3 shows the performance difference in time (ms) before and after splitting. At scope 11, the total time (V+T) is reduced from 4872ms to 3290 ms (32.47 % improvement).*

Remark 7 *We should emphasise that we have not found s systematic approach to derive forbidden patterns. Given a simple constraint, we can specify its forbidden patterns according to its semantics. But for a complex constraint or a set of constraints, its semantics may imply some forbidden patterns. For example, in Figure 3.10, there is a constraint [multi1] (min:0;max:1) on* ***Init****. The constraint and other constraints can imply [multi1] (min:0;max:1) on* ***Show Patient Info****. The implied constraint has a forbidden pattern. We will study how to derive forbidden patterns from constraints in the future. For now, the technique is performed manually. We will consider how to automate the technique in future work.*

3.3.2 Verification of Model Transformations

In this section, we will first give a short description of the semantics of workflow models. A workflow is used to describe a guideline to complete a procedure containing a sequence of tasks. Actors involved in the workflow follow the guidelines in order to achieve a certain goal described by the workflow. For example in the blood transfusion domain, nurses start the procedure and perform tasks (e.g., login to the system, scanning patient wristband, etc.) according to the workflow until all the required tasks are performed. Performing the tasks according to a workflow description is called a *workflow run*.

A workflow model represents a real-life workflow and its semantics is given by all possible runs of the workflow[3]. These runs constitute the state space of the model. Formally, we define the state space of a transition system which is derived by a set of transformation rules (see Paper E). A state is basically an instance of the model which is reachable from the initial state of the model by applying a sequence of transformation rules; these rules describe how a state of the model may evolve into another one. Relating it to a workflow run, an instance represents a snapshot in which certain tasks may have been performed. On the fourth layer of the hierarchy in Figure 3.5, we show an excerpt of an instance of the workflow model $\boldsymbol{M_2}$.

Representing Dynamic Behaviour as Model Transformation

Paper C presents the verification of model transformations using Alloy. In this work, we focus on the model transformations which are executed according to the DPO approach. Take the workflow model $\boldsymbol{M_2}$ in Figure 3.10 as an example. It is a structural model; it does not cover the dynamic behavior of a workflow. In order to specify this feature, we use a variant of the workflow modelling framework (so called DERF) in [234, 241, 242]. In this framework, workflow models are specified in a modelling hierarchy which is similar to the one in Figure 3.5, except that

1. in the workflow metamodel $\boldsymbol{M_1}$ we have also services;
2. in the workflow model $\boldsymbol{M_2}$ we have annotation constraints to show where a service will be running.

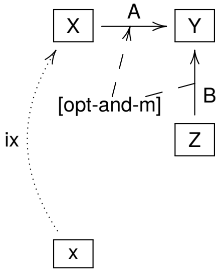
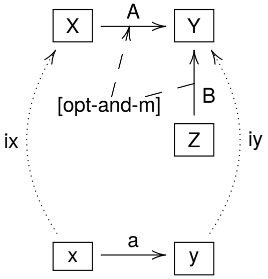
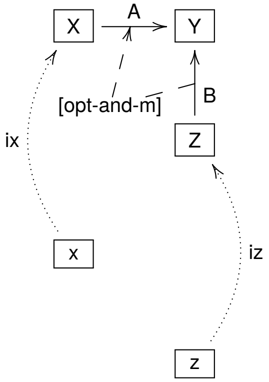
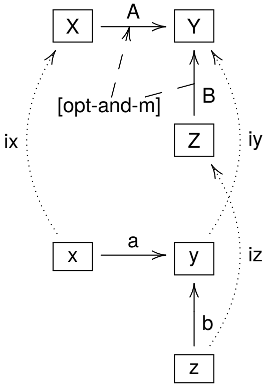
In DERF, a workflow modelling language is created from a metamodel, a set of routing predicates and a set of transformation rules which describe the dynamic behavior of workflow models. A workflow run can be performed by applying these rules to instances of workflow models defined by the language.

In DERF, the authors use coupled model transformation rules which are generic in the sense that they can be applied to instances of all workflow models defined in the language [242]. For instance, the coupled model transformation rules for the routing predicate optional-and-merge can be specified as the two rules in Table 3.4. According to the semantics of these rules, this routing predicate is interpreted as follows:

- $\boldsymbol{t_1}$ whenever an instance $\boldsymbol{x : X}$ is found, regardless the name of $\boldsymbol{X}$, if $\boldsymbol{X}$, $\boldsymbol{Y}$ and $\boldsymbol{Z}$ are related as in $\boldsymbol{L = K}$, then in the next step we create $\boldsymbol{x : X \xrightarrow{a} y : Y}$

[3] A model usually also prescribes a software system which supports the actors in executing the workflow e.g. guiding the actors to perform the right tasks at the right time; a possible run of the workflow is given by an execution of the software system.

Table 3.4: The coupled transformation rules for the routing predicate [optional-and-merge]

t	$L = K$	R
t_1		 X A Y [opt-and-m] B ix Z iy x a y
t_2		 X A Y [opt-and-m] B ix Z iy x a y iz b z

- t_2 whenever the instances $x : X$ and $z : Z$ are found, regardless the names of X and Z, if X, Y and Z are related as in $L = K$, then in the next step we create $x : X \xrightarrow{a} y : Y \xleftarrow{b} z : Z$

In this thesis, we derive a set of transformation rules from the coupled transformation rules provided in DERF for each occurrence of the routing predicates. For instance, the routing predicate optional-and-merge has 3 occurrences in M_2:

1. the arrows **Select Book pre-examination** ⟶ **Send Sample Order** and **Select Blood Typing** ⟶ **Send Sample Order**
2. the arrows **Send Blood Order** ⟶ **Scan Sample** and **Collect Sample** ⟶ **Scan Sample**
3. the arrows **Send Sample** ⟶ **Collect Blood** and **Order Blood Product** ⟶ **Collect Blood**

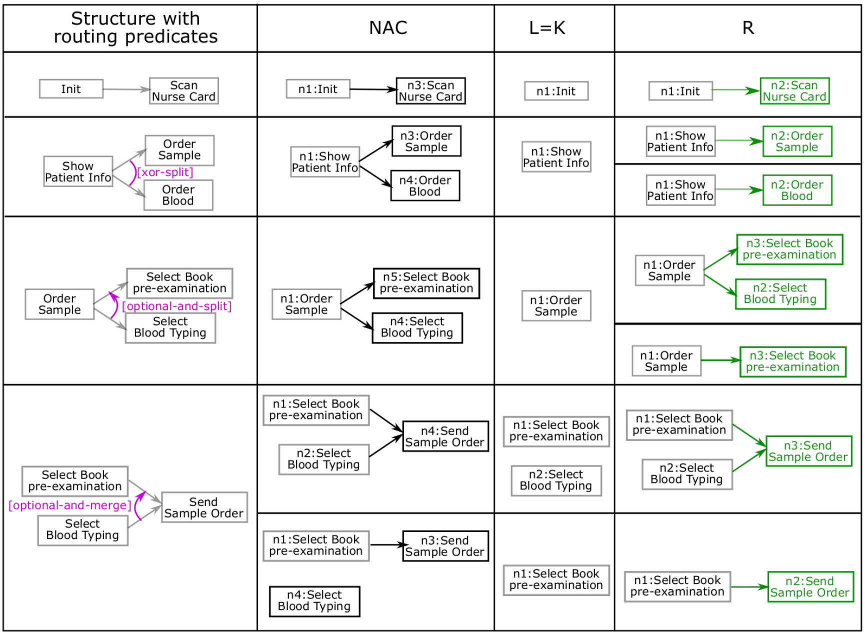


Figure 3.19: Rules describing the dynamic behavior of M_2

For the 3 occurrences of optional-and-merge, we will generate $3 \times 2 = 6$ rules by replacing x, y, z with corresponding task names. Some derived transformation rules for M_2 are shown in Figure 3.19, where we only show the derived rules for the first occurrence of the routing predicate optional-and-merge.

Example 19 (Transformation rules for M_2) *An arrow* ***Init*** $\longrightarrow$ ***Scan Nurse Card*** *in M_2 specifies that* ***Scan Nurse Card*** *is performed exactly after* ***Init****; some routing predicates, e.g.,* xor-split*,* optional-and-split *and* optional-and-merge*, are used to formulate constraints which specify splits or merges of workflow branches. For instance,* [xor-split] *on node* ***Show Patient Info*** *specifies that, after showing patient information, exactly one of the branches (****Order Sample*** *or* ***Order Blood****) is performed;* [optional-and-split] *on node* ***Order Sample*** *specifies that, after ordering sample, either both branches are performed or only a designated branch,* ***Select Book pre-examination*** *(indicated by the arrow), is performed. The NACs are used to control the application of transformation rules. In same cases, the NAC of a rule may be used to avoid nondeterminism between rule applications. For example, for* optional-and-merge*, the NAC of the second rule requires that there is no task typed by* ***Select Blood Typing*** *in a workflow running. This avoids that both rules become applicable at the same time. In addition, a NAC which is equal to the right side of a rule can be used to ensure that the rule can be applied only once via a match.*

Verification of Dynamic Behaviour

```
sig Graph{
    nodes: set NScanPatientWristband+NSendSampleOrder+...+
        NOrderBloodProduct+NInit,
    arrows: set ANOrderSampleNSelectBookpreexamination+...+
        ASampleLabelFile+ALabel+ANInitNScanNurseCard
}
sig Trans{
    rule:one Rule,
    source,target:one Graph,
    dnodes, anodes:set NScanPatientWristband+
        NSendSampleOrder+...+NOrderBloodProduct+NInit,
    darrows, aarrows:set
        ANOrderSampleNSelectBookpreexamination+...+
        ASampleLabelFile+ALabel+ANInitNScanNurseCard
}
fact{
    all trans:Trans|(
    rule_GetNurseInfoScanNurseCard[trans] or
        rule_CollectBloodScanBloodProduct[trans] or ...
    or rule_performtransfusionRegisterReaction[trans] or
        rule_performtransfusionScanBloodProduct[trans])
}
pred rule_InitScanNurseCard[trans:Trans]{
    some trans.rule&InitScanNurseCard{

    some e0:ANInitNScanNurseCard&trans.aarrows|let n0=e0.
        src,n1=e0.trg|
    (n0 in (trans.source.nodes-trans.dnodes) and n1 in
        trans.anodes and (no nac:ANInitNScanNurseCard&(trans
        .source.arrows-trans.darrows)|nac.src=n0))

    #NScanNurseCard&trans.anodes=1
    #ANInitNScanNurseCard&trans.aarrows=1
    no trans.dnodes
    no (NScanPatientWristband+NSendSampleOrder+...+
        NOrderBloodProduct+NInit)&trans.anodes
    no trans.darrows
    no (ANOrderSampleNSelectBookpreexamination+...+ALabel+
        ANInitNScanNurseCard)&trans.aarrows
}
```

Listing 3.5: Encoding of Transformations

Given a set of transformation rules, we are interested in target conformance, i.e., for every source model, is it possible to produce a target model after transformations? In order to verify this property in Alloy, we propose an automatic encoding of metamodels and model transformation rules as Alloy specifications. The transformation can be viewed as an extension of the encoding in Paper B. In addition to encoding metamodels (encoding nodes, arrows and constraints as node signatures, edge signatures and facts/preds), direct model transformations are encoded as signatures which record the changes during the transformations based on transformation rules.

Example 20 (Encoding of M_2 with transformation rules) *An excerpt of the encoding generated from M_2 and its transformation rules is presented in Listing 3.5. Models are encoded as the signature* `Graph` *containing nodes and arrows (line 1-4). Direct model transformations between models are encoded as the signature* `Trans` *which has a pair of graphs representing models before and after transformation (*`source` *and* `target` *on line 7); deleted or added nodes and arrows are encoded as the four fields,* `dnodes`*,* `anodes`*,* `darrows` *and* `aarows` *on line 8-9, of the signature* `Trans`*; each transformation is an application of a transformation rule (line 11-15). Each transformation rule is specified as a pred in Alloy to check whether a transformation applies the rule. For example, the rule derived from the arrow* **Init⟶ Scan Nurse Card** *(the rule in the first row in Figure 3.19) can be encoded as the pred on line 16-28. The pred states which rule is applied (line 17), the matching of the left and right sides of the rule (line 19-20) and the changes caused by applying the rule (line 22-27).*

After encoding transformations in Alloy, we can verify target conformance by checking two conditions: *direct condition*, i.e., every direct model transformation produces a valid target model from a valid source model, and *sequential condition*, i.e., if a direct model transformation t produces an invalid target model from a source model, then there exists a sequence of direct model transformations succeeding the transformation t that produces a target model. The direct condition can be checked automatically in Alloy while the sequential condition must be performed manually.

We use `check` commands in Alloy to check the direct condition; that is to check whether the target model after each transformation satisfies a constraint. If a counterexample is not found within the given scope, it means that the target model of every transformation satisfies the constraint. Otherwise, it means that some transformation may generate a target model which violate the constraint. For the latter case, we can use Alloy to further check the sequential condition. When checking the sequential condition, we need to consider a path consisting of a sequence of transformations. In addition, users have to specify manually how a constraint is violated and how to fix the violation.

Example 21 (Verifying the Dynamic Behaviour of M_2) *The* `check` *command on line 1-7 in Listing 3.6 is used to check whether the target model after each transformation satisfies the constraint[imply2] on edges* **Order Sample Task ⟶ Select Blood Typing** *and* **Select Blood Typing ⟶ Send Sample Order***.*

```
 1 check{
 2     all trans: Trans|//imply2 Blood have not been ordered
           bloodtype is unknown or screening is not done or
           invalid(screen is valid within 4 days)
 3     all y1:NOrderBlood&trans.target.nodes, y2:NOrderSample&
           trans.target.nodes|let e1=(some e:ABloodhavenot
 4     beenordered4&trans.target.arrows|e.trg=y1),
 5     e2=(some e:
           Abloodtypeisunknownorscreeningisnotdoneorinv
 6     alidscreenisvalidwithin4days &trans.target.arrows|e.trg
           =y2)| e1 implies e2
 7 }for 4 but exactly 1 Trans, exactly 2 Graph, exactly 1 Rule

 8 sig Path{trans0, trans1:Trans}{trans0.target=trans1.source}
 9 fact{
10     all path:Path|some n:NGetNurseInfo&path.trans0.target.
           nodes|(no e1:ANursename&path.trans0.target.arrows|e1
           .src=n) and (no e1:ANurseisnotfound&path.trans0.
           target.arrows|e1.src=n)
11
12     all path:Path|some n:NGetNurseInfo&path.trans1.source.
           nodes|(some e1:ANursename&path.trans1.target.aarrows
           |e1.src=n) or (some e1:ANurseisnotfound&path.trans0.
           target.aarrows|e1.src=n)
13 }
14 check{all path:Path|all n:NGetNurseInfo&path.trans1.target.
       nodes|let b1=(some e1:ANursename&path.trans1.target.
       arrows|e1.src=n), b2=(some e1:ANurseisnotfound&path.
       trans1.target.arrows|e1.src=n)| (b1 or b2) and not (b1
       and b2)}
```

Listing 3.6: Verficiaton Commands

Most of the constraints on M_2 are satisfied after transformations except some [xor-split] *constraints, e.g., on the arrows* **Get Nurse Info** ⟶ **Show Nurse Info** *and edges* **Get Nurse Info** ⟶ **Scan Nurse Card**. *The constraint requires that for each* **Get Nurse Info**, *there should be one of the workflow branches, but not both. However, after* **Scan Nurse Card**, *the task* **Get Nurse Info** *is performed. At this point, there is no branch after* **Get Nurse Info**, *which violates* [xor-split]. *For this case, we can use the command on line 14 in Listing 3.6 to check the sequential condition. Here we consider a path consisting of two consecutive transformations (line 8). In addition, we specify that the constraint* [xor-split] *is violated because there is no branch typed by* **Get Nurse Info** ⟶ **Show Nurse Info** *and* **Get Nurse Info** ⟶ **Scan Nurse Card** *(line 10) and the violation can be fixed by adding the corresponding branches typed by the two arrows (line 13). Notice that, we do not specify which rules should be applied to fix the vi-*

olation. Instead, we only specify the possible solution to fix the violation by adding or removing elements. In the way, we can check whether some rules in Figure 3.19 can be applied in the second step to generate an instance satisfying [xor-split] constraints. For the workflow model M_2, the sequential condition is verified correct. It means that there exists a sequence of transformation which can fix the violation of the constraint [xor-split].

We have to manually specify how a constraint is violated and how to fix the violation when checking the sequential condition. In the following section, we present an approach to automatically derive such information from constraints as repair rules.

Repair Rules

When an instance does not conform to its model, we want to repair the instance such that the repaired instance conforms to its model. Here, we proposed an approach to repair instance by using transformation rules. In this approach, we specify semantics of predicates as graph constraints and generate these transformation rules, called repair rules, from the graph constraints. For the constraints based on these predicates, these repair rules can be used to specify how the constraints are violated and how the violation can be fixed. This may enable the automatic checking the sequential condition in the future. Currently, we only consider the graph constraints which conform to two syntactic formats: $gc_1 : L \to R$ and $gc_2 : L \to \neg R$. A structure satisfies gc_1 (gc_2) if for each match of L, $m : L \to S$, there is (not) a match of R, $n : R \to S$, such that $m = gc_1; n$ ($m = gc_2; n$). According to the semantics, the two kinds of graph constraints can be written as $\forall L \to \exists R$ and $\forall L \to \neg\exists R$ respectively.

Example 22 (Predicates Specified as Graph Constraints) *Table 3.5 shows several predicates of which the semantics can be specified as graph constraints. For example, the semantics of* surjective *is specified as a graph constraint in the form of $\forall L \to \exists R$ (in the first row). It says that for each node y typed by* Y*, there should be an arrow typed by* X → Y *where the target of the arrow is y. The predicate* inverse *has also its semantics specified as a graph constraint in the form of $\forall L \to \exists R$ while the other two predicates have semantics specified as graph constraints in the form of $\forall L \to \neg\exists R$.*

Table 3.5: The semantics of predicates in graph constraints

p	$\alpha^{\Sigma}(p)$	Proposed Visualization	Graph Constraint	
			L	R
`surjective`	$1 \xrightarrow{f} 2$	$\boxed{X} \xrightarrow[\text{[surj]}]{f} \boxed{Y}$	$\boxed{:Y}$	$\boxed{:X} \xrightarrow{:f} \boxed{:Y}$
`inverse`	$$\begin{tikzcd} 1 \arrow[r, bend left, "f"] & 2 \arrow[l, bend left, "g"] \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, bend left, "f"] & \boxed{Y} \arrow[l, bend left, "g"] \arrow[l, phantom, "\text{[inv]}"] \end{tikzcd}$$	$\boxed{:X} \xrightarrow{:f} \boxed{:Y}$	$$\begin{tikzcd} \boxed{:X} \arrow[r, bend left, ":f"] & \boxed{:Y} \arrow[l, bend left, ":g"] \end{tikzcd}$$
			$\boxed{:X} \xleftarrow[:g]{} \boxed{:Y}$	$$\begin{tikzcd} \boxed{:X} \arrow[r, bend left, ":f"] & \boxed{:Y} \arrow[l, bend left, ":g"] \end{tikzcd}$$
`injective`	$1 \xrightarrow{f} 2$	$\boxed{X} \xrightarrow[\text{[inj]}]{f} \boxed{Y}$	$\emptyset$	$$\begin{tikzcd} \neg\ \boxed{1{:}X} \arrow[r, "1{:}f"] & \boxed{:Y} \\ \boxed{2{:}X} \arrow[ur, "2{:}f"] & \end{tikzcd}$$
`reflexive`	$$\begin{tikzcd} 1 \arrow[loop above, "f"] \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[loop above, "\text{[inj]}"] \end{tikzcd}$$	$\emptyset$	$\neg\ \boxed{1{:}X} \xrightarrow{:f} \boxed{2{:}X}$

Algorithm 1 (The algorithm to derive repair rules from graph constraints)

```
if the graph constraints is of the form $\forall L \to \exists R$ then
    for each subgraph $r$ of $R$ where $L \subseteq r \subset R$ do
        create $REP_r^a = NAC_r^a \leftarrow L_r^a \to R_r^a$ where
        $NAC_r^a = R$
        $L_r^a = r$
        $R_r^a = R$
    end
    for each subgraph $r$ of $R$ where $r \subset R$ do
        create $REP_r^d$=$NAC_r^d \leftarrow L_r^d \to R_r^d$ where
        $NAC_r^d = R$
        $L_r^d = L$
        $R_r^d = r$
    end
end
if the graph constraints is of the form $\forall L \to \neg\exists R$ then
    for each subgraph $r$ of $R$ where $r \subset R$ do
        create $REP_r^d$=$NAC_r^d \leftarrow L_r^d \to R_r^d$ where
        $NAC_r^d = \emptyset$
        $L_r^d = L$
        $R_r^d = r$
    end
end
```

Table 3.6: Repair rules for graph constrains $\forall \boldsymbol{L} \exists \boldsymbol{R}$

Graph Constraint		Repair rules		
$\mathbf{L}$	$\mathbf{R}$	$\mathbf{N_r = R}$	$\mathbf{L \subseteq L_r^A \subset R}$ $\mathbf{L_r^D = L}$	$\mathbf{R_r^A = R}$ $\mathbf{R_r^D \subset L}$
$\boxed{:Y}$	$\boxed{:X} \xrightarrow{:f} \boxed{:Y}$	$\boxed{:X} \xrightarrow{:f} \boxed{:Y}$	$\boxed{:Y}$	$\emptyset$
			$\boxed{:Y}$	$\boxed{:X} \xrightarrow{:f} \boxed{:Y}$
			$\boxed{:X} \quad \boxed{:Y}$	
$\boxed{:X} \xrightarrow{:f} \boxed{:Y}$	$\boxed{:X} \underset{:g}{\overset{:f}{\rightleftarrows}} \boxed{:Y}$	$\boxed{:X} \underset{:g}{\overset{:f}{\rightleftarrows}} \boxed{:Y}$	$\boxed{:X} \xrightarrow{:f} \boxed{:Y}$	$\emptyset$
				$\boxed{:X} \quad \boxed{:Y}$
				$\boxed{:X}$
				$\boxed{:Y}$
			$\boxed{:X} \xrightarrow{:f} \boxed{:Y}$	$\boxed{:X} \underset{:g}{\overset{:f}{\rightleftarrows}} \boxed{:Y}$

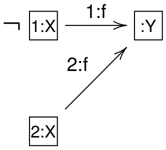
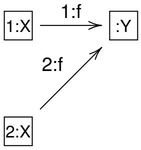
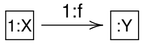
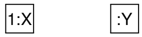
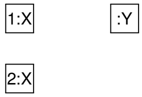
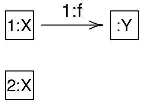
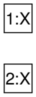
We proposed an algorithm as shown in Algorithm 1 to derive repair rules for the graph constraints in the mentioned two forms. According to the semantics of $\boldsymbol{gc_1}$, if a structure $\boldsymbol{S}$ violates the constraint, there exists some match $\boldsymbol{m} : \boldsymbol{L} \rightarrow \boldsymbol{S}$ where no match $\boldsymbol{n} : \boldsymbol{R} \rightarrow \boldsymbol{S}$ exists such that $\boldsymbol{gc_1};\boldsymbol{n} = \boldsymbol{m}$. In this case, we can fix the violation by adding some elements into $\boldsymbol{S}$ to match $\boldsymbol{R}$ or deleting some elements from $\boldsymbol{S}$ to make such a match $\boldsymbol{m}$ disappear. Based on this observation, for each subgraph $\boldsymbol{r}$ of $\boldsymbol{R}$ where $\boldsymbol{L} \subseteq \boldsymbol{r} \subset \boldsymbol{R}$, we create a rule $\boldsymbol{REP_r^a} = \boldsymbol{R} \leftarrow \boldsymbol{r} \rightarrow \boldsymbol{R}$; for each subgraph $\boldsymbol{r}$ of $\boldsymbol{R}$ where $\boldsymbol{r} \subset \boldsymbol{R}$, we create a rule $\boldsymbol{REP_r^d} = \boldsymbol{R} \leftarrow \boldsymbol{L} \rightarrow \boldsymbol{r}$. As for $\boldsymbol{gc_2}$, if a structure $\boldsymbol{S}$ violates the constraint, there exists some match $\boldsymbol{m} : \boldsymbol{L} \rightarrow \boldsymbol{S}$ and a match $\boldsymbol{n} : \boldsymbol{R} \rightarrow \boldsymbol{S}$ such that $\boldsymbol{gc_2};\boldsymbol{n} = \boldsymbol{m}$. For this case, the algorithm creates rules, for each subgraph $\boldsymbol{r}$ of $\boldsymbol{R}$ where $\boldsymbol{r} \subset \boldsymbol{R}$, $\boldsymbol{REP_r^d} = \emptyset \leftarrow \boldsymbol{L} \rightarrow \boldsymbol{r}$. These rules delete elements to make the matching $\boldsymbol{n}$ disappear.

Example 23 (Repair Rules for Graph Constraints) *Based on the algorithm, we can derive repair rules for the graph constraints in Table 3.5. These repair rules are shown in Table 3.6 and Table 3.7 for the graph constraints in the form of* $\forall \boldsymbol{L} \exists \boldsymbol{R}$ *and* $\forall \boldsymbol{L} \neg \exists \boldsymbol{R}$ *respectively.*

Optimization for Verification of Model Transformations

In Paper C, the verification approach is illustrated by verifying a small model transformation system. The result shows that the verification approach encounters a performance problem. In order to solve the problem, in Paper D, we proposed two techniques to improve the performance of the verification of model transformation systems. The performance of the

Table 3.7: Repair rules for graph constrains $\forall \boldsymbol{L} \neg \exists \boldsymbol{R}$

Graph Constraint		**Repair rules**		
L	**R**	$\mathbf{N_r = \emptyset}$	$\mathbf{L_r^D = L}$	$\mathbf{R_r^D \subset R}$
$\emptyset$		$\emptyset$		$\emptyset$
				
				
				
				
				
				1:X
				:Y

verification is highly affected by the complexity of the metamodel, i.e., the number of nodes and edges (or relations). Since most of the edges in our sample were used to define state information, one technique is to give a more efficient representation of the metamodels by using annotations instead of relations.

Another optimisation technique is to change the encoding of transformation rules. Among deleted or added elements in model transformations, there are *unique elements* whose types are different from the types of other deleted or added elements. If more than two unique elements are connected, the match of one unique element can be derived from the match of other unique elements. Based on this observation, the encoding of the match of transformation rules can be split based on the unique elements.

We applied the two techniques mentioned above to the verification of the same example in Paper C. The result shows that the techniques improve the performance of verification. However, both techniques cannot be applied to the verification of the workflow model $\boldsymbol{M_2}$ in Figure 3.10, since this model does not have state information and each rule in Figure 3.19 has no more than two unique elements.

In Paper E, we applied the verification approach to analyse workflow in healthcare domain. This paper is similar to what we present in this section 3.3.2; workflows are specified as workflow models in the workflow modelling language DERF [234]. The dynamic semantics of the workflow models are specified as coupled model transformation rules. To analyse workflow models by using the verification approach of model transformations, we translate workflow models and their dynamic semantics into Alloy specifications. Then some properties of the DERF language and general properties of workflow models, e.g., absences of deadlocks and termination are verified by finding counterexamples. If such counterexamples are found, they are visualized by the Alloy Analyzer showing how a property is violated.

CHAPTER 4

Conclusion

In this chapter, we will summarize the thesis and discuss some research directions in the future.

4.1 Summary

In this thesis, we firstly presented the `DPF Model Editor` which implemented Diagram Predicate Framework (DPF). The `Signature Editor` was also provided to specify user-defined predicates. In addition, an editor could be generated to construct instances of models. The typing and conformance between models and instances could also be checked. Furthermore, the modelling environment supported multi-level modelling.

The thesis also presented an automatic bounded verification approach of structural models by using Alloy. The approach is integrated into the `DPF Model Editor` such that model designers can verify their models under construction without knowing underlying verification techniques. We also presented a splitting technique which reduces verification of models into verification of submodels. Experimental results showed that the technique tackled the scalability problem to some extent.

The last part in the thesis was about a bounded verification approach of model transformations by using Alloy. We provided an automatic transformation of metamodels and model transformation rules into Alloy specifications. Then the Alloy Analyzer was used to verify the conformance of target models with respect to their metamodels by checking two conditions: direct condition and sequential condition. In order to tackle the scalability problem, we optimized the transformation from model transformation systems to Alloy specifications by decomposing patterns of transformation rules into subpatterns and applying annotation to specify state information. We also applied the approach to verification of workflow models.

4.2 Future Work

The DPF modelling workbench supports diagrammatic (meta)modelling. But many features still remain unexplored. From (meta)modelling aspect, inheritance between classes are quite common. However, it is not supported in the `DPF Model Editor`. We will explore this feature in DPF in the future. From the tooling aspect, the first desired task in order to improve usability is to customize visualization of model elements. They are depicted as rectangles and arrows uniformly now. In addition, we have a web version with the DPF workbench. In the future, we will integrate the two versions together.

The `DPF Model Editor` has integrated the verification approach of structural models as presented in the thesis. However, we only allow users to verify consistence of models and find redundant constraints. In the future, we will provide solutions such that users can specify arbitrary properties to be verified. In addition, we have proposed a technique to split models into submodels. But we have not found a suitable solution to automate the process. We will study algorithms to implement the technique and integrate it into the tool. In order to automate further the process, we shall consider how to automatically derive repair rules from graph constraints. Another direction of model verification using Alloy is to consider how to derive a suitable scope to verify a model.

There are also some interesting research directions for the verification approach of model transformations. The direct condition can be checked automatically. But to check the sequential condition, users have to specify manually how a constraint is violated and how to fix the violation. In the future, we will study how to derive this information automatically from constraints. In addition, currently, we encode model transformation steps. Since some model transformation steps can be executed concurrently, we will investigate how to encode concurrent model transformation encoding.

Bibliography

[1] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Wendy MacCaull, and Adrian Rutle. DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment. In *Computer and Information Science*, volume 429 of *SCI*, pages 37–52. Springer, 2012.

[2] Xiaoliang Wang, Adrian Rutle, and Yngve Lamo. Towards User-Friendly and Efficient Analysis with Alloy. volume 1514 of *CEUR Workshop Proceedings*, pages 28–37, 2015.

[3] Xiaoliang Wang, Fabian Büttner, and Yngve Lamo. Verification of Graph-based Model Transformations Using Alloy. *ECEASST*, 67, 2014.

[4] Xiaoliang Wang, Adrian Rutle, and Yngve Lamo. Scalable Verification of Model Transformations. In *Proceedings of the 11th Workshop on Model-Driven Engineering, Verification and Validation co-located with 17th International Conference on Model Driven Engineering Languages and Systems*, volume 1235 of *CEUR Workshop Proceedings*, pages 29–38. CEUR-WS.org, 2014.

[5] Xiaoliang Wang and Adrian Rutle. Model Checking Healthcare Workflows Using Alloy. In *MMHS*, volume 37 of *Procedia Computer Science*, pages 481–488. Elsevier, 2014.

[6] Bruce Eckel. *Thinking in Java (4th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. ISBN 0131872486.

[7] Stanley B. Lippman, Jose Lajoie, and Barbara E. Moo. *C++ Primer*. Addison-Wesley Professional, 5th edition, 2012. ISBN 0321714113, 9780321714114.

[8] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition, 2013. ISBN 0133807460, 9780133807462.

[9] Dino Esposito and Andrea Saltarello. *Microsoft .NET - Architecting Applications for the Enterprise*. Microsoft Press, 2nd edition, 2014. ISBN 0735685355, 9780735685352.

[10] Andreas Vogel and Keith Duddy. *Java Programming with CORBA*. John Wiley & Sons, Inc., New York, NY, USA, 1997. ISBN 0-471-17986-8.

[11] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In William E. Riddle, Robert M. Balzer, and Kouichi Kishida, editors, *Proceedings, 9th International Conference on Software Engineering, Monterey, California, USA, March 30 - April 2, 1987.*, pages 328–339. ACM Press, 1987. ISBN 0-89791-216-0.

[12] James Martin. *Rapid Application Development*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1991. ISBN 0-02-376775-8.

[13] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN 0135974445.

[14] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006. DOI 10.1109/MC.2006.58.

[15] Tom Mens. On the Complexity of Software Systems. *Computer*, 45 (8):79–81, August 2012. ISSN 0018-9162. DOI 10.1109/MC.2012.273.

[16] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003. DOI 10.1109/MS.2003.1231146.

[17] John D. Poole. Model-Driven Architecture: Vision, Standards and Emerging Technologies. In *In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.

[18] Object Management Group. *Web site*, 2015. `http://www.omg.org`.

[19] OMG Model Driven Architecture. *Web Site*, 2015. `http://www.omg.org/mda/`.

[20] Object Management Group. *MDA Guide*, June 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[21] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[22] David S. Frankel and John Parodi. *The MDA Journal: Model Driven Architecture Straight From The Masters*. Meghan-Kiffer Press, 2004. ISBN 978-0-929652-25-2.

[23] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 032119442X.

[24] Object Management Group. *Unified Modeling Language Specification*, February 2009. http://www.omg.org/spec/UML/2.2/.

[25] Object Management Group. *Meta-Object Facility Specification*, January 2006. http://www.omg.org/spec/MOF/2.0/.

[26] Object Management Group. *XML Metadata Interchange Specification*, December 2007. http://www.omg.org/spec/XMI/2.1.1/.

[27] Eclipse Modeling Framework. *Project Web Site*, 2015. http://www.eclipse.org/modeling/emf/.

[28] Object Management Group. *Query/View/Transformation Specification*, January 2011. http://www.omg.org/spec/QVT/1.1.

[29] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008. DOI 10.1109/TCAD.2008.923410.

[30] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312, 2012. DOI 10.1002/stvr.456.

[31] Gerard J. Holzmann and Rajeev Joshi. Model-Driven Software Verification. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 76–91. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21314-7. DOI 10.1007/978-3-540-24732-6_6.

[32] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. DiVinE 3.0 - An Explicit-State Model Checker for Multithreaded C & C++ Programs. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868. Springer, 2013. ISBN 978-3-642-39798-1. DOI 10.1007/978-3-642-39799-8_60.

[33] Tom Mens. On the Complexity of Software Systems. *IEEE Computer*, 45(8):79–81, 2012. DOI 10.1109/MC.2012.273.

[34] Frederick P. Brooks Jr. No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987. DOI 10.1109/MC.1987.1663532.

[35] Luciano Baresi and Reiko Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In *Proceedings of ICGT 2004: 2^{nd} International Conference on Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 431–433. Springer, 2004. ISBN 978-3-540-23207-0. DOI 10.1007/978-3-540-30203-2_30.

[36] Cambridge. *Dictionaries Online*, 2015. `http://dictionary.cambridge.org`.

[37] Thomas Kühne. Matters of (meta-)modeling. *Software and Systems Modeling*, 5(4):369–385, 2006. DOI 10.1007/s10270-006-0017-9.

[38] Richard H. Thayer. Software System Engineering: A Tutorial. *IEEE Computer*, 35(4):68–73, 2002. DOI 10.1109/MC.2002.993773.

[39] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973. ISBN 978-3-211-81106-1.

[40] Jan Reineke and Stavros Tripakis. Basic Problems in Multi-View Modeling. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2014. ISBN 978-3-642-54861-1. DOI 10.1007/978-3-642-54862-8_15.

[41] World Wide Web. *Extensible Markup Language*, September 2006. `http://www.w3.org/TR/xml11/`.

[42] Alloy. *Project Web Site*, 2009. `http://alloy.mit.edu/community/`.

[43] Gary T. Leavens. Tutorial on JML, the java modeling language. In Stirewalt et al. [243], page 573. ISBN 978-1-59593-882-4. DOI 10.1145/1321631.1321747.

[44] Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# Programming System: Challenges and Directions. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories,*

Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer, 2005. ISBN 978-3-540-69147-1. DOI 10.1007/978-3-540-69149-5_16.

[45] Michael Barnett, Manuel Fähndrich, Peli de Halleux, Francesco Logozzo, and Nikolai Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 401–402. IEEE, 2009. ISBN 978-1-4244-3494-7. DOI 10.1109/ICSE-COMPANION.2009.5071032.

[46] Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. DOI 10.1145/320434.320440. URL `http://doi.acm.org/10.1145/320434.320440`.

[47] Corin A. Gurr. Effective Diagrammatic Communication: Syntactic, Semantic and Pragmatic Issues. *J. Vis. Lang. Comput.*, 10(4):317–342, 1999. DOI 10.1006/jvlc.1999.0130.

[48] Marian Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Commun. ACM*, 38(6):33–44, 1995. DOI 10.1145/203241.203251.

[49] Luc Engelen and Mark van den Brand. Integrating Textual and Graphical Modelling Languages. *Electr. Notes Theor. Comput. Sci.*, 253 (7):105–120, 2010. DOI 10.1016/j.entcs.2010.08.035.

[50] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Softw., Pract. Exper.*, 41(2):133–142, 2011. DOI 10.1002/spe.1006.

[51] Business Process Modeling Notation (BPMN) Information. *Web site*, 2014. `http://www.omg.org/bpmn/`.

[52] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000. DOI 10.1109/32.825767.

[53] Object Management Group. *Unified Modeling Language Specification*, June 2015. `http://www.omg.org/spec/UML/2.5/`.

[54] Object Management Group. *Object Constraint Language Specification*, February 2010. `http://www.omg.org/spec/OCL/2.2/`.

[55] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

[56] Michael Barr and Charles Wells. *Category Theory for Computing Science (2^{nd} Edition)*. Prentice Hall, 1995. ISBN 978-0-13-323809-9.

[57] Zinovy Diskin. *Practical foundations of business system specifications*, chapter Mathematics of UML: Making the Odysseys of UML less dramatic, pages 145–178. Kluwer Academic Publishers, 2003. ISBN 978-1-4020-1480-2.

[58] Zinovy Diskin and Boris Kadish. Generic Model Management. In *Encyclopedia of Database Technologies and Applications*, pages 258–265. Idea Group, 2005. ISBN 978-1-59140-560-3.

[59] Zinovy Diskin and Jürgen Dingel. Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of MoDELS 2006: 9^{th} International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2006. ISBN 978-3-540-45772-5. DOI 10.1007/11880240_17.

[60] Alessandro Rossini. *Diagram Predicate Framework meets Model Versioning and Deep Metamodelling*. PhD thesis, Department of Informatics, University of Bergen, Nov 2011.

[61] Florian Mantz. *Coupled Transformations of Graph Structures applied to Model Migration*. PhD thesis, Department of Mathematics and Informatics, Philipps University in Marburg, Germany, 2014.

[62] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Øyvind Bech, Anders Sandven, and Adrian Rutle. DPF Workbench: A Multi-Level Language Workbench for MDE. *Proceedings of the Estonian Academy of Sciences*, 62:3–15, March 2013. DOI 10.3176/proc.2013.1.02.

[63] Object Management Group. *Unified Modeling Language Specification*, November 2007. `http://www.omg.org/spec/UML/2.1.2/`.

[64] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[65] Tony Clark, Andy Evans, Paul Sammut, and James Willans. *Applied Metamodelling, A Foundation for Language Driven Development*. Ceteva, 2004.

[66] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for Software Engineering*. Wiley, 2008. ISBN 978-0-470-03036-3.

[67] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12 (4):290–321, 2002. DOI 10.1145/643120.643123.

[68] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. DOI 10.1109/MS.2003.1231149.

[69] Thomas Kühne. Clarifying matters of (meta-)modeling: an author's reply. *Software and Systems Modeling*, 5(4):395–401, 2006. DOI 10.1007/s10270-006-0034-8.

[70] Colin Atkinson and Thomas Kühne. Processes and Products in a Multi-Level Metamodeling Architecture. *International Journal of Software Engineering and Knowledge Engineering*, 11(6):761–783, 2001. DOI 10.1142/S0218194001000724.

[71] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003. DOI 10.1109/MS.2003.1231150.

[72] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. DOI 10.1016/j.entcs.2005.10.021.

[73] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006. DOI 10.1147/sj.453.0621.

[74] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software and System Modeling*, 14(1):241–269, 2015. DOI 10.1007/s10270-012-0309-1.

[75] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In 2^{nd} *OOPSLA Workshop on Generative Techniques in the Context of MDA*, 2003.

[76] Matthias Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, July 2010.

[77] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006. ISBN 978-3-540-31187-4. DOI 10.1007/3-540-31188-2.

[78] Juan de Lara and Gabriele Taentzer. Automated Model Transformation and Its Validation Using AToM^3 and AGG. In Alan F. Blackwell, Kim Marriott, and Atsushi Shimojima, editors, *Proceedings of Diagrams 2004: 3rd International Conference on Diagrammatic Representation and Inference*, volume 2980 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2004. ISBN 3-540-21268-X. DOI 10.1007/978-3-540-25931-2_18.

[79] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007. DOI 10.1016/j.scico.2007.05.004.

[80] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the EASST*, 1, 2006.

[81] Yngve Lamo, Florian Mantz, Adrian Rutle, and Juan de Lara. A declarative and bidirectional model transformation approach based on graph co-spans. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 1–12. ACM, 2013. ISBN 978-1-4503-2154-9. DOI 10.1145/2505879.2505900.

[82] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Rozenberg [244], pages 247–312. ISBN 9810228848.

[83] Reiko Heckel, Hartmut Ehrig, Uwe Wolter, and Andrea Corradini. Double-Pullback Transitions and Coalgebraic Loose Semantics for Graph Transformation Systems. *Applied Categorical Structures*, 9(1): 83–110, 2001. DOI 10.1023/A:1008734426504.

[84] Andrea Corradini, Tobias Heindel, Frank Hermann, and Barbara König. Sesqui-Pushout Rewriting. In Corradini et al. [245], pages 30–45. ISBN 3-540-38870-2. DOI 10.1007/11841883_4.

[85] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A formal approach to the specification and transformation of constraints in MDE. *Journal of Logic and Algebraic Programming*, 81(4): 422–457, 2012. ISSN 1567-8326. DOI 10.1016/j.jlap.2012.03.006.

[86] Arend Rensink. Representing First-Order Logic Using Graphs. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and

Grzegorz Rozenberg, editors, *Proceedings of ICGT 2004: 2^{nd} International Conference on Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335. Springer, 2004. ISBN 3-540-23207-9. DOI 10.1007/978-3-540-30203-2_23.

[87] Annegret Habel and Karl-Heinz Pennemann. Nested Constraints and Application Conditions for High-Level Structures. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling, Essays dedicated to Hartmut Ehrig, on the occasion of his 60^{th} Birthday*, volume 3393 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005. ISBN 3-540-24936-2. DOI 10.1007/978-3-540-31847-7_17.

[88] Bruno Courcelle. The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. In Rozenberg [244], pages 313–400. ISBN 9810228848.

[89] Carlos A. González and Jordi Cabot. Formal verification of static software models in MDE: A systematic review. *Information & Software Technology*, 56(8):821–838, 2014. DOI 10.1016/j.infsof.2014.03.003.

[90] IEEE Computer Society. IEEE Standard for Software Verification and Validation. *IEEE Std 1012-2012*, 2012.

[91] Barry W. Boehm. Software Engineering Economics. *IEEE Trans. Software Eng.*, 10(1):4–21, 1984. DOI 10.1109/TSE.1984.5010193.

[92] Barry W. Boehm. Software Risk Management. In Carlo Ghezzi and John A. McDermid, editors, *ESEC '89, 2nd European Software Engineering Conference, University of Warwick, Coventry, UK, September 11-15, 1989, Proceedings*, volume 387 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 1989. ISBN 3-540-51635-2.

[93] Dorota M. Huizinga, David Phung, and Tae Ryu. Automated Defect Prevention with Visual Studio Team System: a Case Study for Software Engineering. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008, July 14-17, 2008, Las Vegas Nevada, USA, 2 Volumes*, pages 32–38. CSREA Press, 2008. ISBN 1-60132-088-4.

[94] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif M. Memon. GUITAR: an innovative tool for automated testing of gui-driven software. *Autom. Softw. Eng.*, 21(1):65–105, 2014. DOI 10.1007/s10515-013-0128-9.

[95] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, Thomas Bruckmann, and Claude Poull. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information & Software Technology*, 57:705–722, 2015. DOI 10.1016/j.infsof.2014.05.007.

[96] Dorothy Graham, Erik Van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification*. Intl Thomson Business Pr, 2008. ISBN 9781844803552, 9781844809899.

[97] Alloy. *Project Web Site*, 2014. `http://alloy.mit.edu/alloy/`.

[98] Oleg Sokolsky and Grigore Rosu. Introduction to the special issue on runtime verification. *Formal Methods in System Design*, 41(3):233–235, 2012. DOI 10.1007/s10703-012-0174-0. URL `http://dx.doi.org/10.1007/s10703-012-0174-0`.

[99] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3): 651–674, 2010. DOI 10.1093/logcom/exn075. URL `http://dx.doi.org/10.1093/logcom/exn075`.

[100] Clare Cini and Adrian Francalanza. An LTL proof system for runtime verification. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 581–595. Springer, 2015. ISBN 978-3-662-46680-3. DOI 10.1007/978-3-662-46681-0_54. URL `http://dx.doi.org/10.1007/978-3-662-46681-0_54`.

[101] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009. DOI 10.1016/j.jlap.2008.08.004. URL `http://dx.doi.org/10.1016/j.jlap.2008.08.004`.

[102] Clarke, Jr., Edmund M. and Grumberg, Orna and Peled, Doron A. *Model Checking*. The MIT Press, 1999.

[103] Jean-Christophe Filliâtre. Deductive software verification. *STTT*, 13 (5):397–403, 2011. DOI 10.1007/s10009-011-0211-0.

[104] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model Checking and the State Explosion Problem. In Meyer and Nordio [246], pages 1–30. ISBN 978-3-642-35745-9. DOI 10.1007/978-3-642-35746-6_1.

[105] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. DOI 10.1145/5397.5399. URL `http://doi.acm.org/10.1145/5397.5399`.

[106] Bernhard Beckert and Reiner Hähnle. Reasoning and Verification: State of the Art and Current Trends. *Intelligent Systems, IEEE*, 29(1): 20–29, Jan 2014. ISSN 1541-1672. DOI 10.1109/MIS.2014.3.

[107] Natarajan Shankar. Automated deduction for verification. *ACM Comput. Surv.*, 41(4), 2009. DOI 10.1145/1592434.1592437.

[108] Sandip Ray. *Scalable Techniques for Formal Verification*. Springer US, 2010. ISBN 978-1-4419-5998-0.

[109] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[110] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. ISBN 3-540-20851-8. DOI 10.1007/978-3-540-24605-3_37.

[111] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An Efficient SAT Solver. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2004. ISBN 3-540-27829-X. DOI 10.1007/11527695_27.

[112] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014. ISBN 978-3-319-08866-2. DOI 10.1007/978-3-319-08867-9_49.

[113] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest,*

Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. ISBN 978-3-540-78799-0. DOI 10.1007/978-3-540-78800-3_24.

[114] Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. ISBN 978-3-540-71065-3. DOI 10.1007/978-3-540-71067-7_6.

[115] Matt Kaufmann and J. Strother Moore. Enhancements to ACL2 in Versions 6.2, 6.3, and 6.4. In Freek Verbeek and Julien Schmaltz, editors, *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014.*, volume 152 of *EPTCS*, pages 1–7, 2014. DOI 10.4204/EPTCS.152.1.

[116] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994. ISBN 3-540-58244-4. DOI 10.1007/BFb0030541.

[117] Christine Paulin-Mohring. Introduction to the Coq Proof-Assistant for Practical Software Verification. In Meyer and Nordio [246], pages 45–95. ISBN 978-3-642-35745-9. DOI 10.1007/978-3-642-35746-6_3. URL `http://dx.doi.org/10.1007/978-3-642-35746-6_3`.

[118] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005. DOI 10.1007/s10009-004-0183-4.

[119] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings* DBL [247], pages 73–80. ISBN 978-0-7695-3388-9. DOI 10.1109/ICSTW.2008.54.

[120] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 21–43. Springer, 2009. ISBN 978-3-540-70999-2. DOI 10.1007/978-3-540-92673-3_1.

[121] Marco Cadoli, Diego Calvanese, and Giuseppe De Giacomo. Towards Implementing Finite Model Reasoning in Description Logics. In Volker Haarslev and Ralf Möller, editors, *Proceedings of the 2004*

International Workshop on Description Logics (DL2004), Whistler, British Columbia, Canada, June 6-8, 2004, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[122] Marco Cadoli, Diego Calvanese, Giuseppe De Giacomo, and Toni Mancini. Finite model reasoning on UML class diagrams via constraint programming. *Intelligenza Artificiale*, 7(1):57–65, 2013. DOI 10.3233/IA-130045.

[123] Marco Cadoli, Diego Calvanese, Giuseppe De Giacomo, and Toni Mancini. Finite Model Reasoning on UML Class Diagrams Via Constraint Programming. In Roberto Basili and Maria Teresa Pazienza, editors, *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence, Rome, Italy, September 10-13, 2007, Proceedings*, volume 4733 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007. ISBN 978-3-540-74781-9. DOI 10.1007/978-3-540-74782-6_5.

[124] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: A Decidable (Yet Expressive) Fragment of OCL. In Yevgeny Kazakov, Domenico Lembo, and Frank Wolter, editors, *Proceedings of the 2012 International Workshop on Description Logics, DL-2012, Rome, Italy, June 7-10, 2012*, volume 846 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[125] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.*, 73:1–22, 2012. DOI 10.1016/j.datak.2011.09.004.

[126] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007. DOI 10.1016/j.websem.2007.03.004.

[127] Anna Queralt and Ernest Teniente. Decidable Reasoning in UML Schemas with Constraints. In Antonio Vallecillo and Goiuria Sagardui, editors, *XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2009), San Sebastián, Spain, September 8-11, 2009*, pages 354–254, 2009. ISBN 978-84-692-4211-7.

[128] Anna Queralt, Guillem Rull, Ernest Teniente, Carles Farré, and Toni Urpí. AuRUS: Automated Reasoning on UML/OCL Schemas. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson C. Woo, and Yair Wand, editors, *Conceptual Modeling - ER 2010, 29th International Conference on Conceptual Modeling, Vancouver, BC, Canada, November 1-4, 2010. Proceedings*, volume 6412 of *Lecture Notes in Computer Science*, pages 438–444. Springer, 2010. ISBN 978-3-642-16372-2. DOI 10.1007/978-3-642-16373-9_32.

[129] Guillem Rull, Carles Farré, Ernest Teniente, and Toni Urpí. Providing Explanations for Database Schema Validation. In Sourav S. Bhowmick, Josef Küng, and Roland Wagner, editors, *Database and Expert Systems Applications, 19th International Conference, DEXA 2008, Turin, Italy, September 1-5, 2008. Proceedings*, volume 5181 of *Lecture Notes in Computer Science*, pages 660–667. Springer, 2008. ISBN 978-3-540-85653-5. DOI 10.1007/978-3-540-85654-2_56.

[130] Kim Marriott and Peter J. Stuckey. *Introduction to Constraint Logic Programming*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262133415.

[131] Roman Barták. Rina Dechter, Constraint Processing, Morgan Kaufmann Publisher (2003) ISBN 1-55860-890-7, Francesca Rossi, Peter van Beek and Toby Walsh, Editors, Handbook of Constraint Programming, Elsevier (2006) ISBN 978-0-444-52726-4. *Computer Science Review*, 2(2):123–130, 2008. DOI 10.1016/j.cosrev.2008.05.001.

[132] ECL^{i}PS^e. *Web site*, 2015. `http://eclipseclp.org/`.

[133] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In Stirewalt et al. [243], pages 547–548. ISBN 978-1-59593-882-4. DOI 10.1145/1321631.1321737.

[134] ArgoUML. *Project Web Site*, 2015. `http://argouml.tigris.org/`.

[135] Pérez González, Alberto Carlos, Fabian Buettner, Robert Clarisó, and Jordi Cabot. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *Proceedings of FormSERA 2012: 1st Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, pages 44–50, Zurich, Suisse, June 2012. IEEE Computer Society. ISBN 978-1-4673-1907-2. DOI 10.1109/FormSERA.2012.6229788.

[136] Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-driven slicing of UML/OCL models. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 185–194. ACM, 2010. ISBN 978-1-4503-0116-9. DOI 10.1145/1858996.1859038.

[137] Lintao Zhang and Sharad Malik. The Quest for Efficient Boolean Satisfiability Solvers. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002. ISBN 3-540-43931-5. DOI 10.1007/3-540-45620-1_26.

[138] Hachemi Bennaceur. A Comparison between SAT and CSP Techniques. *Constraints*, 9(2):123–138, 2004. DOI 10.1023/B:CONS.0000024048.03454.c0.

[139] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9(1):69–86, 2010. DOI 10.1007/s10270-008-0110-3.

[140] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to Alloy and Back Again. In Sudipto Ghosh, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2009. ISBN 978-3-642-12260-6. DOI 10.1007/978-3-642-12261-3_16.

[141] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *49th International Conference, TOOLS Proceedings*, pages 290–306, 2011. DOI 10.1007/978-3-642-21952-8_21.

[142] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005. DOI 10.1007/s10270-005-0089-y.

[143] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007. ISBN 978-3-540-71208-4. DOI 10.1007/978-3-540-71209-1_49.

[144] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using Boolean satisfiability. In *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, pages 1341–1344. IEEE, 2010.

[145] Marina Egea and Vlad Rusu. Formal executable semantics for conformance in the MDE framework. *ISSE*, 6(1-2):73–81, 2010. DOI 10.1007/s11334-009-0108-1.

[146] Manuel Clavel and Marina Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In Michael Johnson

and Varmo Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*, pages 368–373. Springer, 2006. ISBN 3-540-35633-9. DOI 10.1007/11784180_28.

[147] Achim D. Brucker and Burkhart Wolff. HOL-OCL: A formal proof environment for UML/OCL. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer, 2008. ISBN 978-3-540-78742-6. DOI 10.1007/978-3-540-78743-3_8.

[148] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. An MDA Framework Supporting OCL. *ECEASST*, 5, 2006.

[149] David A. Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006. DOI 10.1145/1125808.1125810.

[150] L.A. Rahim. Mapping from OCL/UML metamodel to PVS metamodel. In *Information Technology, 2008. ITSim 2008. International Symposium on*, volume 1, pages 1–8, Aug 2008. DOI 10.1109/ITSIM.2008.4631599.

[151] Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking Unsatisfiability for OCL Constraints. *ECEASST*, 24, 2009.

[152] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the Object Constraint Language into First-order Predicate Logic. In *In Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC*, pages 113–123, 2002.

[153] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY Approach: Integrating Object Oriented Design and Formal Verification. In Manuel Ojeda-Aciego, Inman P. de Guzmán, Gerhard Brewka, and Luís Moniz Pereira, editors, *Logics in Artificial Intelligence, European Workshop, JELIA 2000 Malaga, Spain, September 29 - October 2, 2000, Proceedings*, volume 1919 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2000. ISBN 3-540-41131-3. DOI 10.1007/3-540-40006-0_3.

[154] David Roe, Krysia Broda, Alessandra Russo, and Ra Russo. Mapping UML models incorporating OCL constraints into Object-Z. Technical report, Imperial College of Science, Technology and Medicine, Department of Computing, 2003.

[155] Soon-Kyeong Kim and David A. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. In Jonathan P. Bowen, Steve Dunne, Andy Galloway, and Steve King, editors, *ZB 2000: Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, August 29 - September 2, 2000, Proceedings*, volume 1878 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 2000. ISBN 3-540-67944-8. DOI 10.1007/3-540-44525-0_2.

[156] Arvinder Kaur, Samridhi Gulati, and Sarita Singh. A comparative study of two formal specification languages: Z-notation & B-method. In Natarajan Meghanathan and Michal Wozniak, editors, *The Second International Conference on Computational Science, Engineering and Information Technology, CCSEIT '12, Coimbatore, India, October 26-28, 2012*, pages 524–531. ACM, 2012. ISBN 978-1-4503-1310-0. DOI 10.1145/2393216.2393304.

[157] Rafael M. Kamenoff and Nicole Lévy. Using B formal specifications for analysis and verification of UML/OCL models. In *Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, Dresden, Germany, September 2002.

[158] Marcin Szlenk. Formal Semantics and Reasoning about UML Class Diagram. In *Dependability of Computer Systems, 2006. DepCos-RELCOMEX '06. International Conference on*, pages 51–59, May 2006. DOI 10.1109/DEPCOS-RELCOMEX.2006.27.

[159] Moussa Amrani, Levi Lucio, Gehan M. K. Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *IEEE 5th International Conference on Software Testing, Verification and Validation*, pages 921–928, 2012.

[160] Daniel Calegari and Nora Szasz. Verification of Model Transformations: A Survey of the State-of-the-Art. *Electr. Notes Theor. Comput. Sci.*, 292:5–25, 2013. DOI 10.1016/j.entcs.2013.02.002.

[161] Lukman Ab. Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software and Systems Modeling*, 2014.

[162] H. J. Sander Bruggink. Towards a Systematic Method for Proving Termination of Graph Transformation Systems. *Electr. Notes Theor. Comput. Sci.*, 213(1):23–38, 2008. DOI 10.1016/j.entcs.2008.04.072.

[163] Jochen Malte Küster. Definition and validation of model transformations. *Software and System Modeling*, 5:233–259, 2006.

[164] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination Analysis of Model Transformations by Petri Nets. In *Graph Transformations, Third International Conference, ICGT Proceedings*, pages 260–274, 2006.

[165] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2002. ISBN 3-540-44310-X. DOI 10.1007/3-540-45832-8_14.

[166] The Attributed Graph Grammar System. *Project Web Site*, 2015. `http://user.cs.tu-berlin.de/~gragra/agg/`.

[167] Hartmut Ehrig, Karsten Ehrig, Gabriele Taentzer, Juan de Lara, Dániel Varró, and Szilvia Varró-Gyapay. Termination Criteria for Model Transformation. In *Transformation Techniques in Software Engineering*, 2005.

[168] Bruno Barroca, Levi Lucio, Vasco Amaral, Roberto Félix, and Vasco Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *The 3rd International Conference on Software Language Engineering*, pages 296–305, 2010.

[169] Gabriele Taentzer. *Parallel and distributed graph transformation - formal description and application to communication-based systems*. Berichte aus der Informatik. Shaker, 1996. ISBN 978-3-8265-1636-8.

[170] Thomas Baar and Slavisa Markovic. A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules. In Irina Virbitskaite and Andrei Voronkov, editors, *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers*, volume 4378 of *Lecture Notes in Computer Science*, pages 70–83. Springer, 2006. ISBN 978-3-540-70880-3. DOI 10.1007/978-3-540-70881-0_9.

[171] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 668–682, 2011.

[172] Julia Padberg, Magdalena Gajewsky, and Claudia Ermel. Rule-Based Refinement of High-Level Nets Preserving Safety Properties. In *FASE*, pages 221–238, 1998.

[173] Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal Refactoring for UML Class Diagrams. In *19TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES)*, pages 152–167, 2005.

[174] Detlef Plump. Termination of Graph Rewriting is Undecidable. *Fundam. Inform.*, 33(2):201–209, 1998. DOI 10.3233/FI-1998-33204.

[175] Detlef Plump. Confluence of Graph Transformation Revisited. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel C. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 280–308. Springer, 2005. ISBN 3-540-30911-X. DOI 10.1007/11601548_16.

[176] Paul C. Jorgensen. *Software testing - a craftsman's approach (4. ed.)*. Taylor & Francis, 2013.

[177] Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model transformation testing challenges. In *Proceedings of the IMDDMDT workshop at ECMDA'06*, 2006.

[178] Junhua Wang, Soon-Kyeong Kim, and David A. Carrington. Verifying Metamodel Coverage of Model Transformations. In *ASWEC*, pages 270–282. IEEE Computer Society, 2006.

[179] *Validation in model-driven engineering: testing model transformations*, 2004.

[180] Ingo Stuermer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic Testing of Model-Based Code Generators. *IEEE Trans. Softw. Eng.*, 33:622–634, 2007.

[181] Maher Lamari. Towards an automated test generation for the verification of model transformations. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors,

Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007, pages 998–1005. ACM, 2007. ISBN 1-59593-480-4. DOI 10.1145/1244002.1244220.

[182] Jochen M. Küster and Mohamed Abd-El-Razik. Validation of Model Transformations – First Experiences using a White Box Approach. In *IN PROCEEDINGS OF MODEVA'06 (MODEL DESIGN AND VALIDATION WORKSHOP ASSOCIATED TO MODELS'06*. Springer, 2006.

[183] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008*, pages 328–337. IEEE Computer Society, 2008. ISBN 978-0-7695-3127-4. DOI 10.1109/ICST.2008.62.

[184] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Model transformation testing: oracle issue. In *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings* DBL [247], pages 105–112. ISBN 978-0-7695-3388-9. DOI 10.1109/ICSTW.2008.27.

[185] Esther Guerra, Juan Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated Verification of Model Transformations Based on Visual Contracts. *Automated Software Engg.*, 20(1):5–46, March 2013. ISSN 0928-8910. DOI 10.1007/s10515-012-0102-y.

[186] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pages 13–20, 2006.

[187] Fernando Orejas and Martin Wirsing. On the Specification and Verification of Model Transformations. In *Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*, pages 140–161, 2009.

[188] Yuehua Lin, Jing Zhang, and Jeff Gray. A Testing Framework for Model Transformations. In *in Model-Driven Software Development - Research and Practice in Software Engineering. 2005*, pages 219–236. Springer, 2005.

[189] Pau Giner and Vicente Pelechano. Test-Driven Development of Model Transformations. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 2009. ISBN 978-3-642-04424-3.

[190] Andrea Darabos, András Pataricza, and Dániel Varró. Towards Testing the Implementation of Graph Transformations. *Electr. Notes Theor. Comput. Sci.*, 211:75–85, 2008. DOI 10.1016/j.entcs.2008.04.031.

[191] Reiko Heckel. Compositional Verification of Reactive Systems Specified by Graph Transformation. In *FASE*, pages 138–153, 1998. DOI 10.1007/BFb0053588.

[192] GRaphs for Object-Oriented VErification. *Project Web Site*, 2006. http://groove.sourceforge.net/groove-index.html.

[193] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *Applications of Graph Transformations with Industrial Relevance, Second International Workshop*, pages 479–485, 2003.

[194] Arend Rensink. Explicit State Model Checking for Graph Grammars. *Concurrency, Graphs and Models*, pages 114–132, 2008. DOI 10.1007/978-3-540-68679-8_8.

[195] Harmen Kastenberg. Towards Attributed Graphs in Groove: Work in Progress. *Electr. Notes Theor. Comput. Sci.*, 154(2):47–54, 2006. DOI 10.1016/j.entcs.2005.03.030.

[196] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In Antti Valmari, editor, *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer, 2006. ISBN 3-540-33102-6. DOI 10.1007/11691617_19.

[197] Arend Rensink and Eduardo Zambon. Neighbourhood Abstraction in GROOVE. *ECEASST*, 32, 2010.

[198] The EMF Henshin Transformation Tool. *Project Web Site*, 2010. http://www.eclipse.org/modeling/emft/henshin/.

[199] Levi Lucio, Bruno Barroca, and Vasco Amaral. A Technique for Automatic Validation of Model Transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2010. ISBN 978-3-642-16144-5.

[200] Ákos Schmidt and Dániel Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In *The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference Proceedings*, pages 92–95, 2003.

[201] Fernando Luís Dotti, Luciana Foss, Leila Ribeiro, and Osmar Marchi dos Santos. Verification of Distributed Object-Based Systems. In *International Conference on Formal Methods for Open Object-Based Distributed Systems Proceedings*, pages 261–275, 2003.

[202] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[203] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10:5: 1–29, 2011. DOI 10.5381/jot.2011.10.1.a5.

[204] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2): 31–39, 2008. DOI 10.1016/j.scico.2007.08.002.

[205] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. ISBN 978-3-540-71940-3.

[206] Miguel García and Ralf Möller. Certification of transformation algorithms in model-driven software development. In Wolf-Gideon Bleek, Jörg Raasch, and Heinz Züllighoven, editors, *Software Engineering 2007, Fachtagung des GI-Fachbereichs Softwaretechnik, 27.-30.3.2007 in Hamburg*, volume 105 of *LNI*, pages 107–118. GI, 2007. ISBN 978-3-88579-199-7.

[207] Leslie Lamport. The $^{+}$CAL Algorithm Language. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006.*, volume 4229 of *Lecture Notes in Computer Science*, page 23. Springer, 2006. ISBN 3-540-46219-8. DOI 10.1007/11888116_2.

[208] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA^{+} Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999. ISBN 3-540-66559-5. DOI 10.1007/3-540-48153-2_6.

[209] Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to*

Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, volume 5503 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009. ISBN 978-3-642-00592-3. DOI 10.1007/978-3-642-00593-0_2.

[210] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.*, 28:626–643, 1996.

[211] Márk Asztalos, László Lengyel, and Tihamer Levendovszky. Towards Automated, Formal Verification of Model Transformations. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 15–24. IEEE Computer Society, 2010. ISBN 978-0-7695-3990-4. DOI 10.1109/ICST.2010.42.

[212] Visual Modeling and Transformation System. *Web site*, 2015. `http://vmts.aut.bme.hu/`.

[213] SWI-Prolog. *Web site*, 2015. `http://www.swi-prolog.org/`.

[214] Daniel Calegari, Carlos Luna, Nora Szasz, and Álvaro Tasistro. A Type-theoretic Framework for Certified Model Transformations. In *Proceedings of the 13th Brazilian Conference on Formal Methods: Foundations and Applications*, pages 112–127, 2011.

[215] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN 978-3-642-05880-6. DOI 10.1007/978-3-662-07964-5.

[216] Kevin Lano and Shekoufeh Kolahdouz Rahimi. Specification and Verification of Model Transformations Using UML-RSDS. In Dominique Méry and Stephan Merz, editors, *Integrated Formal Methods - 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings*, volume 6396 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2010. ISBN 978-3-642-16264-0. DOI 10.1007/978-3-642-16265-7_15.

[217] Kevin Lano and Howard Haughton. *Specification in B: An Introduction Using the B Toolkit*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996. ISBN 1860940080.

[218] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010. DOI 10.1016/j.jss.2009.08.012.

[219] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2008. ISBN 978-3-540-87404-1. DOI 10.1007/978-3-540-87405-8_28.

[220] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006. ISBN 3-540-45772-0. DOI 10.1007/11880240_31.

[221] Kurt Stenzel, Nina Moebius, and Wolfgang Reif. Formal verification of QVT transformations for code generation. *Software and System Modeling*, 14(2):981–1002, 2015. DOI 10.1007/s10270-013-0351-7.

[222] KIV. *Web site*, 2015. `http://www.isse.uni-augsburg.de/en/software/kiv/`.

[223] Holger Giese, Sabine Glesner, Johannes Leitner, Wilhelm Schäfer, and Robert Wagner. Towards Verified Model Transformations. In *MoDeVVA2006, Proceedings*, pages 78–93, 2006.

[224] Fujaba Developer Team. *The Fujaba Tool Suite*, 2012. `http://www.fujaba.de/`.

[225] Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Graph-transformation verification using monadic second-order logic. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 17–28. ACM, 2011. ISBN 978-1-4503-0776-5. DOI 10.1145/2003476.2003482.

[226] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic Second-Order Logic in Practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995, Proceedings*, volume 1019 of *Lecture Notes*

in Computer Science, pages 89–110. Springer, 1995. ISBN 3-540-60630-0. DOI 10.1007/3-540-60630-0_5.

[227] Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB J.*, 9(1):76–110, 2000. DOI 10.1007/s007780050084.

[228] Fabian Büttner, Marina Egea, and Jordi Cabot. On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 432–448. Springer, 2012. ISBN 978-3-642-33665-2. DOI 10.1007/978-3-642-33666-9_28.

[229] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL Transformations Using Transformation Models and Model Finders. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, volume 7635 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2012. ISBN 978-3-642-34280-6. DOI 10.1007/978-3-642-34281-3_16.

[230] Gehan M. K. Selim, Fabian Büttner, James R. Cordy, Jürgen Dingel, and Shige Wang. Automated Verification of Model Transformations in the Automotive Industry. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 690–706. Springer, 2013. ISBN 978-3-642-41532-6. DOI 10.1007/978-3-642-41533-3_42.

[231] Fabian Büttner, Marina Egea, Esther Guerra, and Juan de Lara. Checking Model Transformation Refinement. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, volume 7909 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2013. ISBN 978-3-642-38882-8. DOI 10.1007/978-3-642-38883-5_15.

[232] Luciano Baresi and Paola Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In Corradini et al. [245], pages 306–320. ISBN 3-540-38870-2. DOI 10.1007/11841883_22.

[233] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M. Küster. Analysis of Model Transformations via Alloy. In *MoDeVVa'2007, Proceedings*, 2007.

[234] Hao Wang, Adrian Rutle, and Wendy MacCaull. A formal diagrammatic approach to timed workflow modelling. In Tiziana Margaria, Zongyan Qiu, and Hongli Yang, editors, *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, pages 167–174. IEEE Computer Society, 2012. ISBN 978-0-7695-4751-0. DOI 10.1109/TASE.2012.14. URL `http://dx.doi.org/10.1109/TASE.2012.14`.

[235] World Health Organization. *Blood safety and availability*, 2015. `http://www.who.int/mediacentre/factsheets/fs279/en/`.

[236] Øyvind Bech. DPF Editor: A Multi-Layer Modelling Environment for Diagram Predicate Framework in Eclipse. Master's thesis, Department of Informatics, University of Bergen, Norway, May 2011.

[237] Graphical Editing Framework. *Project Web Site*. `http://www.eclipse.org/gef/`.

[238] Claudia Ermel, Enrico Biermann, Johann Schmidt, and Angeline Warning. Visual Modeling of Controlled EMF Model Transformation using Henshin. *Electronic Communications of the EASST*, 32, 2010.

[239] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A Formalisation of Constraint-Aware Model Transformations. In David Rosenblum and Gabriele Taentzer, editors, *Proceedings of FASE 2010: 13^{th} International Conference on Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2010. ISBN 978-3-642-12028-2. DOI 10.1007/978-3-642-12029-9_2.

[240] Xiaoliang Wang, Adrian Rutle, and Yngve Lamo. Towards User-Friendly and Efficient Analysis with Alloy. `http://dpf.hib.no/wp-content/uploads/draft2014.pdf`, 2014.

[241] Adrian Rutle, Hao Wang, and Wendy MacCaull. A Formal Diagrammatic Approach to Compensable Workflow Modelling. In Zhiming Liu and Alan Wassyng, editors, *Foundations of Health Informatics Engineering and Systems*, volume 7789 of *Lecture Notes in Computer Science*, pages 194–212. Springer Berlin Heidelberg, 2013. DOI 10.1007/.

[242] Adrian Rutle, Wendy MacCaull, Hao Wang, and Yngve Lamo. A metamodelling approach to behavioural modelling. In *Proceedings*

of the Fourth Workshop on Behaviour Modelling - Foundations and Applications, BM-FA '12, pages 5:1–5:10, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1187-8. DOI 10.1145/2325276.2325281. URL `http://doi.acm.org/10.1145/2325276.2325281`.

[243] R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors. *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, 2007. ACM. ISBN 978-1-59593-882-4.

[244] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, 1997. World Scientific. ISBN 9810228848.

[245] Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors. *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-38870-2.

[246] Bertrand Meyer and Martin Nordio, editors. *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, 2012. Springer. ISBN 978-3-642-35745-9. DOI 10.1007/978-3-642-35746-6.

[247] *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*, 2008. IEEE Computer Society. ISBN 978-0-7695-3388-9.

Part II

Scientific Contributions

PAPER E

Model Checking Healthcare Workflows using Alloy

ELSEVIER

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 00 (2014) 000–000

Procedia
Computer Science

www.elsevier.com/locate/procedia

5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014)

Model Checking Healthcare Workflows using Alloy

Xiaoliang Wang[a], Adrian Rutle[a,*]

[a]Bergen University College, Bergen, Norway

Abstract

Workflows are used to organize business processes, and workflow management tools are used to guide users in which order these processes should be performed. These tools increase organizational efficiency and enable users to focus on the tasks and activities rather than complex processes. Workflow models represent real life workflows and consist mainly of a graph-based structure where nodes represent tasks and arrows represent the flows between these tasks. From workflow models, one can use model transformations to generate workflow software. The correctness of the software is dependent on the correctness of the models, hence verification of the models against certain properties like termination, liveness and absence of deadlock are crucial in safety critical domains like healthcare. In previous works we presented a formal diagrammatic framework for workflow modelling and verification which uses principles from model-driven engineering. The framework uses a metamodelling approach for the specification of workflow models, and a transformation module which creates DiVinE code used for verification of model properties. In this paper, in order to improve the scalability and efficiency of the model checking approach, we introduce a new encoding of the workflow models using the Alloy specification language, and we present a bounded verification approach for workflow models based on relational logic. We automatically translate the workflow metamodel into a model transformation specification in Alloy. Properties of the workflow can then be verified against the specification; especially, we can verify properties about loops. We use a running example to explain the metamodelling approach and the encoding to Alloy.

Keywords: Workflow modelling, Efficient verification, Alloy, Model checking, Model-driven engineering.

1. Introduction

Healthcare is the domain which cost states and local governments a considerable portion of their budgets. Furthermore, mistakes in almost any aspect of a healthcare-related system may cause severe damages. This has lead to an increasing pressure on making processes and procedures in healthcare safer and more effective. Clinical guidelines, dictating how processes should be organized, have been provided by health authorities to guide and unify healthcare processes across institutions. These guidelines are in constant changes due to updates in regulations and advances in treatment methods and medications. Unfortunately, the guidelines are traditionally written in natural languages, which can run to hundreds of pages, incorporating heavily annotated diagrams which use non-standard and confusing notations[1].

[*] Corresponding author. Tel.: +47-5558-7791 ; fax: +47-5558-7789.
E-mail addresses: xwa@hib.no (Xiaoliang Wang)., aru@hib.no (Adrian Rutle).

Workflow models may be used to formally describe clinical guidelines. A workflow model consists mainly of a graph-based structure where nodes represent tasks and arrows represent the flow between these tasks. In earlier work[2,3,4] we proposed a diagrammatic framework (called DERF) for the specification of workflow models using model-driven engineering (MDE)[5,6] techniques. The diagrammatic models are easily understood by domain-experts, and the metamodelling approach allows models to be easily customized to deal with new treatment procedures and other changes in clinical guidelines.

From workflow models, one can use model transformations to generate workflow software. Workflow software are used to guide users in which order these processes should be performed, and to resolve dependencies between tasks. These tools improve organizational efficiency and enable users to focus on the tasks and activities rather than complex processes. The correctness of the software is dependent on the correctness of the models, hence verification of the models against certain properties like termination, liveness and absence of deadlock are crucial in safety critical domains like healthcare. In[7] we proposed a verification approach for models specified in DERF, in which the workflow models were transformed to DVE, the language of the DiVinE model checker. The approach also incorporated a user-friendly editor for specification of model properties, as well as a module for visualization of counter-examples in case some properties did not hold. In this paper, we extend upon our earlier work, and introduce a new, efficient encoding of the workflow models using the Alloy specification language. Furthermore, we present a bounded verification approach for workflow models based on relational logic. We automatically translate the workflow metamodel into a model transformation specification in Alloy. Properties of the workflow can then be verified against the specification; especially, we can verify properties about loops. In case a property does not hold, a counter-example is generated automatically by the Alloy and visualized as a graph. We use a running example (adopted from[7]) to explain the metamodelling approach and the encoding to Alloy.

In Section 2 we review our workflow modelling language. In Section 4 we discuss correctness of workflow models, explain our encoding to the Alloy specification language, and visualize counter-examples. Sections 5 and 6 present some related and future work and conclude the paper.

2. Metamodeling for Healthcare Workflows

Workflow models may be used to document and analyse complex work processes in clinical guidelines and to ensure their correctness. In previous work, we presented a diagrammatic modelling framework used for workflow modelling[2,3,4,7]. A design goal of the framework has been to make the modelling tools intuitive enough to be used by healthcare practitioners and formal enough to be used to specify and verify interesting properties of healthcare workflows. Here, we only present the most important details of the framework, the details can be found in the references above. This short presentation of the modelling language and the running example are adopted from[7].

The workflows are represented as graph-based structures describing in which order specific tasks should be executed. Each task is represented by a node. If there is an arrow $T_1 \xrightarrow{e} T_2$ from a task T_1 to a task T_2, then task T_1 must be performed before task T_2. Special binary constraints on forks (joins) specify splits (respectively, merges) of workflow branches. In fact, joins and forks could be extended in the standard way to arbitrary triples, quadruples, etc. The most used splits (e.g. [and_split], [or_split] or [xor_split]) and merges (e.g. [and_merge], [xor_merge] or [or_merge]) are formulated as predicates in our framework. The meaning of these constraints are as usual: both branches have to be executed in an [and_split]; exactly one branch has to be executed in an [xor_split] and one or two branches have to be executed in an [or_split].

Fig. 1 shows a sample of a workflow from the healthcare domain. The workflow illustrates a simplified scenario for cancer treatment. After an initial examination, the patient will have an MRI examination *and* a blood test. According to the results of the two tests, the physician will decide which procedure the patient should follow (*either* Procedure A *or* Procedure B). After finishing the chosen procedure, the result shall be evaluated to determine whether the patient should use drug treatment or not. If drug treatment is chosen, then when the drugs are finished a blood test is taken and the result is evaluated to determine whether the patient should be given further drug treatment or not. Hence if the drug treatment is repeated, the blood test and the evaluation will be repeated as well; i.e., the workflow will be in a loop. The workflow ends when the evaluation shows that the drug treatment should terminate.

The syntax and semantics of the workflow modelling language is given in[2,3,4,7]; here we only recall some of the details. The modelling language is defined using the Diagram Predicate Framework (DPF)[8] and implemented using the DPF Workbench[9]. In DPF, a modelling language is given by a metamodel and a diagrammatic predicate signature

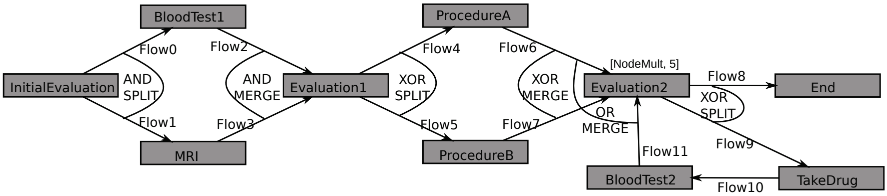


Figure 1: Sample workflow model. Adopted from[7].

(see Fig. 2). The metamodel defines the types and the signature defines the predicates that are used to formulate constraints by the users. A model in DPF consists of an underlying graph, and a set of constraints. DPF supports a multi-level metamodelling hierarchy, in which a model at any level can be regarded the metamodel for models at the level below it. In DERF, we have three modelling levels: M2, M1 and M0. The metamodel of our workflow modelling language (which is at level M2) consists of a node Task and an arrow Flow. This means that we can define a set of tasks together with the flow relations between these tasks. The signature Σ_2 of the workflow modelling language consists of a set of routing predicates such as `[and_split]`, `[and_merge]`, `[xor_merge]`, etc. Tasks which are involved in a cycle in the workflow are marked with a predicate `[NodeMult,n]` where n specifies how many instances that task can have at most. We call these tasks "loop tasks", and we call flows within a loop for "loop flows".

From the metamodel at level M2 and the signature Σ_2 with routing predicates, we can create a modelling language for the definition of "workflow models". These workflow models, which conform to the metamodel at level M2, are located at level M1. Given a specific workflow model at level M1 (like the one in Fig. 1) and the predicates <E>, <R> and <F> (where <E>, <R>, and <F> denotes that a task instance is enabled, running, and finished, respectively) collected in a signature Σ_1 (see Fig. 2) We refer to <E>, <R> and <F> as "task states". Note that in an earlier version of the language[2,3] we had 4 states, <D>, <E>, <R> and <F>, thereof the name DERF. These workflow states are located at level M0, and conform to the workflow model. Beginning with a state at level M0 (that may be referred to as an instance of the workflow model) we generate states by applying model transformation rules (see[4] for the complete set of rules). For example rule r_1 takes an instance of a task from <E> to <R> and rule r_2 takes an instance of a task from <R> to <F>. A workflow run is represented by an execution path in the state space of the workflow model; i.e., by a sequence of rule applications. The state space which can be generated by the transformation rules comprises the dynamic semantics of the workflow.

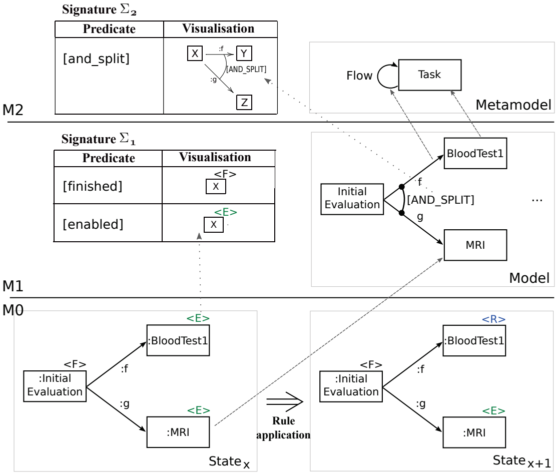


Figure 2: Workflow modelling hierarchy: dashed arrows indicate types of some model elements, dotted arrows indicate relations between signatures and models. Adopted from[7].

3. Encoding of workflow model

In this section, we will cover how to encode a workflow model and its corresponding transition system as an Alloy specification. The specification represents a model transformation system which simulates the dynamic semantics (each task can change from a state to another). However, the state information is not represented in the generated specification. The encoding procedure is adapted based on our encoding of model transformation systems detailed in[10]. It is implemented as a code generation module in DPF and can derive the Alloy specification automatically from a workflow model and the coupled transformation rules. Before presenting the encoding procedure, we give a brief introduction to Alloy.

Alloy[11] is a structural modelling language, based on first-order logic, for expressing complex structure and constraints. The Alloy Analyzer is a constraint solver translating Alloy specifications written in relational logic to a boolean satisfiability problem which is automatically evaluated by a SAT solver. For a given specification F, the Alloy Analyzer attempts to find an instance which satisfies F or find a counterexamples which violates F by running run or $check$ command. The instance or counter-example is displayed graphically, and their appearance can be customized for the domain at hand.

3.1. Encoding of the metamodel at M2 level

Recall that each model in DPF (and also in DERF) consists of an underlying graph and a set of constraints. Given a workflow model, for the underlying graph, each task $t{:}Task$ is encoded as a task signature S_t; each flow $f{:}Flow$ is encoded as a flow signature S_f with two fields src and trg denoting the source task and the target task of the flow. The encoding procedure handles the loop tasks specially. In order to count how many times the task is performed, a field $count$ is added to the loop task's signature. Thus the workflow model can be encoded as a graph signature S_G containing two fields: the field $nodes$ denoting the tasks; the field $arrows$ denoting the flows. Since the structure is a graph, it should satisfy that if a flow is contained by a graph g, its source and target tasks should also be contained by g. The structure encoding is shown in the following listing: (assuming the structure contains m tasks and n flows.)

```
sig $S_{t_i}${count:one Int//The field is optional depending if the task is a loop task or
    within a loop.
}//For each task $t_i$, $i \in \{1..m\}$
sig $S_{f_j}${src:one $S^s_{f_j}$, trg:one $S^t_{f_j}$}{//For each flow $f_j$, $j \in \{1..n\}$, $S^s_{f_j}$/$S^t_{f_j}$ is the flow's
    source/target task
sig $S_G${nodes:set $S_{t_1}$+...+$S_{t_m}$,edges:set $S_{f_1}$+...+$S_{f_n}$}
fact{all g:$S_G$|all e:g.edges|(e.src in g.nodes and e.trg in g.nodes)}
```

Besides the structural information, the workflow model contains also constraints restricting the set of valid instances. The constraints are of two types:

General Constraints These constraints are implicitly contained in each workflow model and must be satisfied by all workflow states. In the DPF jargon, we specify these constraints using universal constraints[8].

1. Each task instance may enable at most one instance of the same subsequent task. This is forced by a multiplicity constraint `mult[0..1]` on each flow in workflow models. Similarly, two instances of the same task cannot enable the same instance of a subsequent task. This is forced by injective constraint `[inj]` on each flow.
2. A task instance cannot be enabled before its preceding task is finished. To specify this constraint, when a task has only one incoming flow, the flow will be constrained with surjective constraint `[surj]`. However, if the task has multiple incoming flows and the model designer has not put any routing constraint on these, the constraint `[or_merge]` is put on the flows.
3. If a task has incoming flows mixing loop flows and ordinary flows, two separate `[or_merge]` (or `[sur]` if the sets contain only one) are put on each of these two sets.

Specific Constraints These constraints are specified in a workflow model explicitly by designers. These constraints are formulated using predicates from Σ_2. Since there is a limited number of predicates for the workflow modelling language, these predicates are hard-coded in the implementation and used to formulate different constraints in the models. For example, the `[xor_split,`c`]` constraints in Fig. 1 are encoded as:

```
pred fact_E1_xor_split[g:Graph]{//For Evaluation1
    all n:NE1&g.nodes|not ((some e:AE1_PB&g.arrows|e.src=n) and (some e:AE1_PA&g.
        arrows|e.src=n))
}
pred fact_E2_xor_split[g:Graph]{//For Evaluation2
    all n:NE1&g.nodes|not ((some e:AE1_PB&g.arrows|e.src=n) and (some e:AE1_PA&g.
        arrows|e.src=n))
}
```

3.2. *Encoding of model transformation*

In DERF, we use coupled transformation rules to define the dynamic semantics of workflow models. We adopt a variant of the encoding procedure for transformation rules detailed in [10]. First, we derive the graph transformation rules by finding the matching of each coupled transformation rule. For example, for the rule r_4 in [4] defining the semantics of `[xor_split,c]`, two matches are found: one on Evaluation1 and one on Evaluation2 (See rules $E1^1_{xs}, E1^2_{xs}, E2^1_{xs}, E2^2_{xs}$ in Table 1). Note that this step of deriving the graph transformation rules is performed implicitly in the encoding procedure. Then each derived rule r is encoded as a predicate *pre apply_r*$[tran{:}Trans]$ as in [10] stating that a transformation applies the rule. The signature $Trans$, as in [10], encodes the direct model transformations which contains four fields: the rule applied $rule$, the source workflow $source$, the target workflow $target$, and, the deleted and added elements during the transformation $dnodes$, $anodes$, $darrows$, $aarrows$. Assuming there are nr derived rules, the following $fact$ statement asserts that every transformation should apply exactly one of the derived rules.

```
fact {all t:Trans | apply_r_1[t] or ... or apply_r_nr[r]}
```

Since in the workflow modelling language loops are represented as tasks with predicate `[MultNode,n]`, the loop tasks can be repeated a finite number n of times. That is, the loop tasks may have up to n instances. Therefore, when deriving the graph transformation rules for this case, several points should be considered:

- For the incoming flows of a loop task which are not loop flows, the rule creates a new instance of the loop task with $count = 0$ (see rules $E2^1_{xm}$ and $E2^2_{xm}$ in Table 1).
- For the flow loops which are not coming into a loop task, the rule creates a new instance of the flow's target task with $count$ equals to the flow's source task. In addition, for the flow coming out of a loop task, a precondition should check if its count is less than the upper limit n in `[MultNode,n]` (see rule $E2^2_{xs}$ in Table 1).
- For the loop flows coming into a loop task, the rule shall create a new instance of the loop task with $count = count' + 1$, where $count'$ is the count of the flow's source task (see rule $Flow11$ in Table 1).

Rule	L	K	R
$E1^1_{xs}$	$:E1$	$:E1$	$:E1 \rightarrow :PA$
$E1^2_{xs}$	$:E1$	$:E1$	$:E1 \rightarrow :PB$
$E2^1_{xs}$	$:E2$	$:E2$	$:E2 \rightarrow :End$
$E2^2_{xs}$	$c<5$ $:E2$	$c<5$ $:E2$	$c<5$ $:E2 \rightarrow :TD$ c
$E2^1_{xm}$	$:PA$	$:PA$	$:PA \rightarrow :E2$ $c=0$
$E2^2_{xm}$	$:PB$	$:PB$	$:PB \rightarrow :E2$ $c=0$
Flow11	c $:BT2$	c $:BT2$	c $:BT2 \rightarrow :E2$ $c+1$
Flow10	c $:TD$	c $:TD$	c $:TD \rightarrow :BT2$ c

Table 1: Derived graph transformation rules

4. Verification of Healthcare Workflow

After a workflow is encoded as an Alloy specification, the Alloy Analyzer could be used to verify its properties. In this work, we want to verify whether the workflow model satisfies *generic properties* such as: 1) absence of deadlocks, and, 2) terminatoin (when loops are present). The Alloy Analyzer performs a bounded check and can prove whether the workflow system is without error w.r.t. the properties within a user-defined scope. Hence, the approach can find bugs in a workflow model efficiently. In addition, the Alloy Analyzer can visualize the counterexamples if they exist. To verify these properties, we firstly verify that each instance of the workflow model (except the start instance where only instances of tasks without incoming flows present) can be derived by applying a rule on a workflow instance; i.e., we have to verify that each reachable instance of the workflow model can be generated by applying a transformation rule. Note that this property is related to the DERF language, that is, we are verifying that the workflow metamodel satisfy the property. Therefore, the property is only needed to be verified once. If this is verified, it means that each workflow instance contains path information; i.e., there exists a sequence of transformations applied on the start state to get such an instance.

To verify the property with the encoded Alloy specification, we check the *Direct Condition*[10] to show that each transformation from a valid source state could produce a valid target state. In addition, a similar condition should also be verified: if the result of a transformation is a valid state the source is also a valid state.Similar to the verification method in[10], these two properties are verified by running the commands in the following listing. The scope we use is $for\ 10\ but\ exactly\ 1\ Trans, exactly\ 2\ Graph$. It means that in each workflow instance, at most 10 instances of each task (such as Evaluation1 and Evaluation2) are present.

```
check{all trans:Trans|valid[trans.source] and not valid[trans.target]} for 10 but
    exactly 1 Trans, exactly 2 Graph
check{all trans:Trans|not valid[trans.source] and valid[trans.target] and not isStart[
    trans.target]} for 10 but exactly 1 Trans, exactly 2 Graph
```

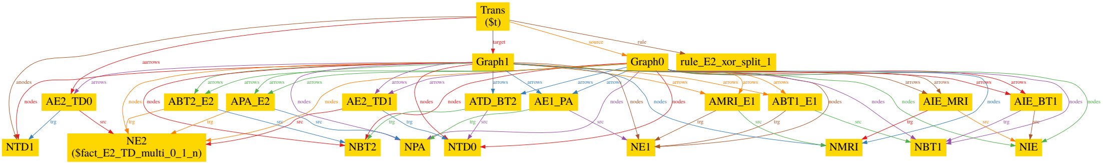


Figure 3: Counterexample of `xor_split`

The verification result shows several counterexamples; e.g. the [xor_split,c] constraint is violated. One violation is shown in Fig. 3. To correct this problem the rule for [xor_split,c] should use the two split branches as NAC to avoid reapplying the rules multiple times (see Table 1). The errors and counterexamples disappear after the encoding is revised and the . This means that the encoded Alloy specification correctly simulates the dynamics of the workflow model.

Now we can prove the properties like absence of deadlock or termination for loops. To verify the absence of deadlock property, we try to find a transformation where the source state is valid $valid[trans.source]$, the target state is not in finished state $not\ finished[trans.target]$ (which means the workflow terminates,) and no rule can be applied on the target model $not\ rules\_applicable[trans.target]$. If such transformation is found, it means there is deadlock in the workflow model. The Alloy Analyzer finds an instance by the command in the following listing.

```
run{all trans:Trans|valid[trans.source] and not finished[trans.target] and not
    rules_applicable[trans.target]} for 10 but exactly 1 Trans, exactly 2 Graph
```

We can verify that a workflow will terminate although it contains a loop. It means each time a workflow enters a loop, it will terminate in the future. We can use the Alloy Analyzer to find counterexamples. That is, a workflow has entered a loop but have not finished or have further applicable rule. Actually, this is a special case of deadlock verification. The result shows there is no deadlock or loop without termination for the workflow model.

```
run{all trans:Trans|has_enter_loop[trans.source] and valid[trans.source] and not
    finished[trans.target] and not rules_applicable[trans.target]} for 10 but exactly
    1 Trans, exactly 2 Graph
```

5. Related Work

We shortly present some efforts using model checking for verification of safety critical systems. Pérez et al.[12] use MDE-based tool chain semi-automatically to process manually created clinical guideline specifications and generate the input model of a model checker from the specifications. The approach uses Dwyer patterns[13] to specify commonly occurring types of properties. In[14] the authors propose an approach to the verification of clinical guidelines, which is based on the integration of a computerized guidelines management system with a model-checker. Advanced Artificial Intelligence techniques are used to enhance verification of the guidelines. The approach is first presented as a general methodology and then instantiated by loosely coupling the guidelines management system GLARE[15] and the model checker SPIN[16]. A similar approach was presented by Rabbi et al.[17] to model compensable workflows using the Compensable Workflow Modelling Language (CWML) and its verification by an automated translator to the DiVinE model checker. In[18] a method to minimize the risk of failure of business process management systems from a compliance perspective is presented. Business process models expressed in the Business Process Execution Language (BPEL) are transformed into pi-calculus and then into finite state machines. Compliance rules captured in the graphical Business Property Specification Language (BPSL) are translated into linear temporal logic. Thus, process models can be verified against these compliance rules by means of model checking technology.

Most of these works use model checking to verify the workflow system while we use Alloy, based on relation logic and a satisfiability solver. These works are complete since the model checker work on the whole state space. However, our approach is bounded and incomplete, i.e., the properties verified is only valid in some scope. But our approach could find bugs in the system more efficiently. In addition, the above mentioned works have their own patterns and languages to specify the properties and verify different kinds of properties, while in our work, we only verify those mentioned properties if they are expressed in first-order logic. Furthermore, we can also derive the model checker input file (semi-)automatically.

6. Conclusion and Future Work

In this paper, we apply a bounded verification approach based on Alloy to the verification of healthcare workflow models. We build on our MDE-based workflow modelling language for the definition of diagrammatic workflow models. In order to verify a workflow, the dynamic semantic of the workflow is simulated as a model transformation system, encoded as a specification in Alloy. Then the Alloy Analyzer is used to verify general properties of the workflow by finding counterexamples. If such counterexamples are found, they are visualized by the Alloy Analyzer showing how a property is violated.

One of the main contributions of the paper is that we use a new technique to verify workflow models. Comparing with other approaches with model checking techniques, the approach is bounded and incomplete. But the approach enable the designer quickly find the bugs in the models and correct them with the feedback from the verification result. In[10], the verification approach based on Alloy encounter a scalability problem when the relations in metamodel or transformations rules are too complex. But as we can see from the workflow metamodel and the derived transformation rules, this may not be a problem; because the arity of the relations in the coupled model transformations are at most 2.

In this work, we only applied the approach to one workflow model. In the future, larger models will be used to study the performance of the approach. Right now, limited properties are verified with the approach. More study should be continued to see whether other properties can be verified. In[7] we used a user-friendly editor for the specification of properties. We plan on translating properties defined in this editor so that they can be verified against the Alloy specifications using Alloy Analyzer. Furthermore, we abstract out the state information in the encoding procedure. Actually, some flows, like `TakeDrug` to `Evaluation2`, could be also omitted. We will check if any systematical approach could make the encoding result simpler.

References

1. D. Méry, N. Singh, Medical protocol diagnosis using formal methods, in: Z. Liu, A. Wassyng (Eds.), Foundations of Health Informatics Engineering and Systems, Vol. 7151 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 1–20. doi:10.1007/978-3-642-32355-3_1.
2. H. Wang, A. Rutle, W. MacCaull, A Formal Diagrammatic Approach to Timed Workflow Modelling, in: Proceedings of TASE 2012: 6^{th} International Conference on Theoretical Aspects of Software Engineering, Vol. 0, IEEE Computer Society, 2012, pp. 167–174.
3. A. Rutle, H. Wang, W. MacCaull, A Formal Diagrammatic Approach to Compensable Workflow Modelling, in: Z. Liu, A. Wassyng (Eds.), Foundations of Health Informatics Engineering and Systems, Vol. 7789 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 194–212.

4. A. Rutle, W. MacCaull, H. Wang, Y. Lamo, A Metamodelling Approach to Behavioural Modelling, in: Proceedings of BM-FA 2012: 4^{th} Workshop on Behavioural Modelling: Foundations and Applications, ACM, 2012, pp. 5:1–5:10.
5. B. Selic, The pragmatics of model-driven development, IEEE Softw. 20 (5) (2003) 19–25. doi:10.1109/MS.2003.1231146.
6. T. Stahl, M. Völter, Model-Driven Software Development: Technology, Engineering, Management, Wiley, 2006.
7. A. Rutle, F. Rabbi, W. MacCaull, Y. Lamo, A user-friendly tool for model checking healthcare workflows, Procedia Computer Science 21 (0) (2013) 317 – 326, the 4th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2013) and the 3rd International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH). **Best paper award**. doi:/10.1016/j.procs.2013.09.042.
8. A. Rutle, Diagram Predicate Framework: A Formal Approach to MDE, Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2010).
9. Y. Lamo, X. Wang, F. Mantz, W. MacCaull, A. Rutle, DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment, in: R. Lee (Ed.), Computer and Information Science 2012, Vol. 429 of Studies in Computational Intelligence, Springer Berlin Heidelberg, 2012, pp. 37–52. doi:10.1007/978-3-642-30454-5_3.
10. X. Wang, Y. Lamo, F. Büttner, Verification of graph-based model transformation using alloy, in: In: Proc. of GTVMT, 2014.
11. Alloy, Project Web Site, http://alloy.mit.edu/community/.
12. B. Pérez, I. Porres, Authoring and verification of clinical guidelines: A model driven approach, Journal of Biomedical Informatics 43 (4) (2010) 520–536.
13. M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in property specifications for finite-state verification, in: Proceedings of the 21st international conference on Software engineering, ICSE '99, ACM, New York, NY, USA, 1999, pp. 411–420.
14. A. Bottrighi, L. Giordano, G. Molino, S. Montani, P. Terenziani, M. Torchio, Adopting model checking techniques for clinical guidelines verification, Artificial Intelligence in Medicine 48 (1) (2010) 1 – 19. doi:10.1016/j.artmed.2009.09.003.
15. L. Anselma, A. Bottrighi, G. Molino, S. Montani, P. Terenziani, M. Torchio, Supporting knowledge-based decision making in the medical context: The glare approach, IJKBO 1 (1) (2011) 42–60.
16. SPIN, Project Web Site, http://spinroot.com/.
17. F. Rabbi, H. Wang, W. MacCaull, Compensable Workflow Nets, in: Proceedings of ICFEM 2010: 12^{th} International Conference on Formal Engineering Methods, Vol. 6447 of Lecture Notes in Computer Science, Springer, 2010, pp. 122–137. doi:/10.1007/978-3-642-16901-4_10.
18. Y. Liu, S. Müller, K. Xu, A static compliance-checking framework for business process models, IBM Syst. J. 46 (2) (2007) 335–361.