MODEL TRANSFORMATION DEPENDABILITY EVALUATION BY THE AUTOMATED CREATION OF MODEL GENERATORS

by

SEYYED MADASAR ALI SHAH

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
April 2012

Abstract

This thesis is on the automatic creation of model generators to assist the validation of model transformations. The model driven software development methodology advocates models as the main artefact to represent software during development. Such models are automatically converted, by transformation tools, to apply in different stages of development. In one application of the method, it becomes possible to synthesise software implementations from design models. However, the transformations used to convert models are man-made, and so prone to development error. An error in a transformation can be transmitted to the created software, potentially creating many invalid systems.

Evaluating that model transformations are reliable is fundamental to the success of modelling as a principle software development practice. Models generated via the technique presented in this thesis can be applied to validate transformations. In several existing transformation validation techniques, some form of conversion is employed. However, those techniques do not apply to validate the conversions used there-in. A defining feature of the current presentation is the utilization of transformations, making the technique self-hosting. That is, an implementation of the presented technique can create generators to assist model transformations validation and to assist validation of that implementation of the technique.

For my family.

ACKNOWLEDGEMENTS

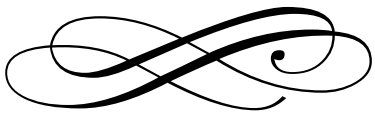
My deepest thanks and love go to my family, to whom I dedicate this work. The loving care of my mother has made me who I am today, I would be lost without you. Gratitude goes to my brothers, sisters and cousins, who have brought me very much inspiration and happiness this world. To my nephews: I love you. Also I must thank my aunts and uncles for all the great family get-togethers, and wish to enjoy many more with them. My older brother deserves a special mention, he has supported me in this pursuit of this work.

Dr. Behzad Bordbar, my thesis advisor: sincerely, thank you. Without his prolific support and guidance, this work would not be as it is; he has provided the opportunities that have resulted in my achievements. Special thanks go to Dr. Kyriakos Anastasakis, Dr. Rami Bahsoon, Arif Ahmedeen, Mohammed Alodib, Imran Bajwaa and all those in the group who have provided stimulating discussions over the years. Thanks also to my thesis group members Dr. Mark Lee and Dr. Georgios Theodoropoulos, who have provided very useful feedback on my work. Discussion has been the seed of inspiration from which my work has grown.

My sincere thanks go to all the tutors and demonstrators who I have had the pleasure of teaching with on the software workshop module. I have very much enjoyed demonstrating and tutoring all those who have gone on to bigger and better things. The hard work and dedication of the instructors and students on the module made teaching in the department a pleasure. I would also like to thank Dr. Martín Escardó, Dr. Jim Yandle and Dr. Jon Rowe for their commitment to the development of others. I have learnt a great deal from you all.

Saving the best until now, dear friends: Peter Lewis, Mohammed Alodib, Hana Shamsudin made for a fun and enlightening working environment; it really was the best office on campus. In the dungeon, my thanks go to Anh Dinh, Hasan Qunoo, Edd Robinson, Shou Wang, Noor Sadawi, Noel Welsh, Mohamed Menaa, Zeyn Saigol, Amina Zidouk, Thanh Nguyen, you have all shown me the meaning of determination; making dark winter days much brighter. I have been honoured to know and befriend Vivek Nallur, Ben Jones, Chris Staite, Philip Rolfshagen, Tebogo Seiepone, Laura Ewers, Arjun Chandra, Damien Jade Duff and Peter Oliveto on the path of this work. The friendship all of you have given has been an immeasurable gift.

To all those who have helped me on my journey, I can not begin to thank you enough.



CONTENTS

1 Introduction **1**

1.1 Motivation 6

1.2 Approach 7

1.3 Contributions 8

1.4 Overview 9

2 Model Driven Software Development, Modelling, Model Transformation and Software Dependability **13**

2.1 Synopsis 15

2.2 Overview of Model Driven Software Development 15

2.3 Modelling Software Systems 18

2.4 Defining Modelling Formalisms by Meta Models 20

2.5 Conversion of Software Models by Model Transformation 24

2.6 Software Dependability: Validation and Verification for Improving Reliability 30

2.7 Summary 34

3 Existent Techniques to Evaluate the Dependability of Model Transformations **37**

3.1 Synopsis 39

3.2 Dependability: Validation and Verification of Model Transformation Correctness to Improve Reliability 39

3.3 Importance of Transformation Dependability Evaluation 41

3.4 Difficulties of Model Transformation Dependability Evaluation 42

3.5 Comparison of Model Transformation Dependability Techniques 45
3.6 Classification of Model Transformation Dependability Evaluation Techniques . 48
3.7 Basis for Model Transformation Validation Technique 52
3.8 Outline of Properties for Proposed Validation Technique 55
3.9 Summary 55

4 A Technique for the Automatic Creation of Model Generators to Assist Model Transformation Validation 57

4.1 Synopsis 59
4.2 Meta Models: Complex, Non-Generative Input Specification 59
4.3 Generation of Models from Meta Models - Architecture for Model Generation 64
4.3.1 Bounded "Model Finding" Software Verification Tools Exploiting SAT Solvers 64
4.3.2 Model Generation using SAT based "Model Finders" 66
4.3.3 Meta Model Re-Representation towards Model Generation 67
4.3.4 Meta Transformation of Traces to Automate to Conversion of Generate Instances 71
4.3.5 Transformation of Generated Instances 73
4.3.6 Evaluating the Dependability of the Model Transformations used in the Technique to Create Model Generators 75
4.4 Model Transformation Validation using Generated Instances 78
4.4.1 Automating Application of the Technique 78
4.4.2 Oracles to Support Evaluating Results of Validation 81
4.5 Integrating Coverage for Generating Models 86
4.6 Summary 87

5 Implementing Self-Validating and Repeatable Creation of Model Generators from Meta Models 89

5.1 Synopsis 91

5.2 Outline: Creating a Model Generator 91
5.3 Stage One: Conversion to Kodkod Re-Representation 93
5.3.1 Meta Model Transformation : EMF2Fur 93
5.3.2 Fur Meta Model to Textual Representation : Fur2Kodkod 99
5.4 Kodkod Instance Conversion: Rational for Model Transformation based on Trace 105
5.5 Stage Two : Creation of Kodkod Instance Converter from Trace 110
5.6 Outcome: Model Generator by Model Transformation to Support Transformation Validation 115
5.7 Direction for Future Implementation 116
5.7.1 Issues of Setting and Automating Oracles for Validation 116
5.7.2 Tools and Modelling Technologies 119
5.7.3 Scalability and Tractability of Meta Model Analysis using Kodkod . . 120
5.7.4 Practical Observations: Advantages and Limitations of the Proposal . 122
5.8 Summary 123

6 An Evaluation of the Application of Model Generators to Model Transformation Validation 127

6.1 Synopsis 129
6.2 Aim and Methodology of Evaluation 129
6.3 Rational for Chosen Case Studies 131
6.4 Case Study One: Families2Persons Transformation 133
6.5 Case Study Two: Tree2List Transformation 141
6.5.1 Apparent Error in Tree2List Transformation 144
6.6 Case Study Three: EMF2Fur Model Transformation Self-Validation 146
6.7 Comparison to Existing Model Generation Technique for Transformation Validation 151
6.8 Updated Classification of Model Transformation Dependability Evaluation Techniques 154
6.9 Summary 156

7 Conclusion **159**

7.1 Summary of Contributions 164

7.2 Future Work 165

LIST OF FIGURES

2.1 Ideals of the model driven software development methodology: models directly inform multiple stages of development. 16

2.2 Sample class diagram, model of software. 19

2.3 Example models in a meta modelling hierarchy for model driven software development. 21

2.4 Three- and four-level modelling hierarchies in model driven software development. 22

2.5 Overview of rule-based model-to-model transformation. 26

2.6 Model transformation: advanced applications. 27

(a) Meta model transformation. 27

(b) Transformation execution tracing. 27

2.6 Model transformation: advanced applications. 28

(c) Higher order model transformation. 28

(d) Chains of model transformation. 28

2.7 General overview of software verification against dependability properties. . . 30

2.8 General overview of software validation by demonstration of software features. 33

4.1 Meta models define an infinite number of structural models. 62

(a) Example meta model. 62

(b) Combinations of valid models with multiple meta elements. 62

(c) Relationships between models with multiple meta elements. 62

(d) Attributes of meta-elements allow for more valid instance of the element. 62

4.2 Meta models constraints arbitrarily restrict the valid models in a modelling formalism. 63
(a) Example meta model with relationship cardinality and constraints. . . . 63
(b) Relationship cardinalities restrict the valid models. 63
(c) Logical constraints further restrict the valid models. 63
4.3 SAT based analysis of complex software abstractions. 66
4.4 Overview of meta model transformation : conversion of meta models to model finding formalism. 69
4.5 Modelling hierarchy : models, meta models and meta meta models. 70
4.6 Higher order transformation to convert trace to model transformation. 73
4.7 Conversion of generated instances to models of the original meta model. . . . 74
4.8 Self validation by generating meta models. 77
4.9 Validation using model generators created by the technique presented in this work. 79

5.1 Overview of the implementation and the artefacts produced by the implementation. 92
5.2 Meta models used in EMF2Fur model transformation. 94
(a) ECore kernel meta model. 94
(b) Fur meta model. 94
5.3 Velocity template for Fur2Kodkod model-to-text transformation. 101
5.4 Example conversion of OCL constraints to Kodkod logic. 102
5.5 Kodkod to ECore instance conversion scenarios. 107
(a) Simple Kodkod to ECore instance conversion scenario. 107
(b) Kodkod to ECore instance conversion: problematic due to duplicate atoms and hierarchy. 107
(c) Kodkod to ECore instance conversion: problematic due to duplicate, arbitrary names and binary relations. 107
5.6 Meta models used in Trace2MT Model Transformation. 111

(a) TraceMM: Trace Meta Model. 111
(b) MTransformation: Transformation Meta Model. 111
5.7 Velocity template for MTranformation2SiTra model-to-text transformation. . . 114
6.1 Families2Persons meta models. 134
(a) Families meta model. 134
(b) Persons meta model. 134
6.2 Families2Persons model transformation in ATLAS. 135
6.3 Families2Persons provided sample family model and equivalent, created by transformation application. 136
(a) Families sample model, provided with transformation. 136
(b) Persons model produced by application of transformation to the model in figure 6.3a. 136
6.4 Model generator created for the Families meta model for use in validation. . . 137
6.5 Generated models and application to validation. 138
(a) Generated model and equivalent: working situation. 138
(b) Generated model and equivalent: unexpected situation. 138
(c) Generated model and equivalent: error-causing situation. 138
6.6 Tree2List model transformation and meta models. 142
(a) MMTree meta model. 142
(b) MMElementList meta model. 142
(c) Tree2List model transformation definition in ATLAS. 142
6.7 Generated model : error-causing situation. 144
6.8 Meta model generator created for the ECore meta meta model and application to self-validation. 147
6.9 Extended ECore meta meta model. 149

LIST OF TABLES

3.1 Features of model transformation quality evaluation techniques. 51

4.1 Six generic oracles for validation of model transformation. 82

5.1 EMF2Fur: rules to convert elements of ECore meta models to elements of Fur meta model. 98

5.2 OCL and corresponding Kodkod expressions: outline of OCL2Kodkod transformation. 103

5.3 Trace2MT: rules to convert trace to model transformation. 112

6.1 Features of selected model transformation quality evaluation techniques. . . . 155

CHAPTER 1

INTRODUCTION

A man's errors are his portals of discovery.
— James Joyce

The software revolution has been of great benefit to human-kind, driving change in our increasingly network– and computer– reliant world. General purpose computers are only possible because the software programs applied to them can be re-programmed; allowing existing systems to be improved and new systems developed for previously unthought of applications. Personal and mobile computing, the Internet and online social networks are only few of the new tools available to us because of re-programmable software running on general purpose computational machines.

Re-programmable computers have a further benefit: the seemingly continual cost reductions. Economies of scale dictate that as general purpose computers are more widely used, the cost to produce such hardware is reduced. This means that with every generation, more and more people can benefit from the software revolution. Furthermore, the software that is programmed on to a computer can be created long after the computer has been fabricated, allowing for new and more innovative uses of the same hardware.

However, the development of software is a difficult and costly undertaking. Software is created by developers, who must be trained to create systems using languages that computers can understand. And as with any complex man-made system, software programs are prone to error. More than half the development time and cost of a software system can be attributed to evaluating that the software will work as anticipated. A testament to the advantages of general purpose computers is fact that software tools are created to aid the development processes. New programming tools, methodologies and languages are developed using existing software; raising the level of abstraction to support the human understanding of computational problems.

Evaluating the dependability of developed software is a key challenge in computer science. Particularly for ensuring that software tools used in the development of new software will work as anticipated. Dependability properties of software have be defined and classified as availability, reliability, safety, integrity, maintainability [22]. However, this work is concerned with one aspect of dependability: reliability; specifically, dealing with pre-emptive detection of errors (from the taxonomy defined of [22]) in model transformations.

Model transformations are software tools used to automatically convert models of software between diverse modelling formalisms. Model transformations are a central principle of the emerging model driven software development methodologies; which aim to raise the level of software abstraction used during development. This thesis presents a technique to automate model generation, a key process in evaluating the dependability of model transformation tools.

In the literature, several techniques have been proposed to evaluate dependability of model transformations. Evaluating dependability of model transformation is found to be a complex and difficult undertaking. Existing techniques solve the problem in diverse ways, promising solutions use model generation by applying automated software analysis formalisms. Other promising techniques automate the conversion of artefacts to software analysis formalisms. Several interesting techniques generate models that are applicable to transformation validation. By analysis of existing techniques, the desirable features of model generation techniques are uncovered.

In model driven software development, abstract models are the primary artefact of software development. Models are used to represent software during development, using an appropriate modelling notation. Abstractly, models define the allowed scenarios in a software system and such models can have both textual and visual notations. Models in model driven software development are created and manipulated by software tools at each stage of the development of software, thus bringing tool support and automation to the stages used in software development.

Models are created to conform to an appropriate modelling formalism. Each modelling formalism is suited to a specific phase of software development and kind of software represented. A diverse range of modelling formalisms are available to represent software for the design, implementation, analysis or specification of software systems. Meta models are the models of modelling formalisms and to specify modelling formalisms in model driven software development. Software tools are used with meta models to support the creation of models and also to automatically convert a models between modelling formalisms. Con-

version allows models created in a given stage to directly effect the other stages of software development.

Model transformation are created by experts to convert models and form an essential mechanism in model driven software development. Model transformations automate the conversion of models from one modelling formalism to another, allowing developers to automatically apply a given model of software to different purposes. Model transformations are created for several purposes in model driven software development. For example, transformation can be applied to convert abstract models to another specific formalism, or to interpret models, or to refine models to a required form, or to abstract salient details on from complex models, or to analysis of the quality of the models.

The model driven software development methodology focuses development on models as the central artefacts of towards the creation of software systems. The automation that model transformation affords is a key advantage to model driven software development. Crucially transformations are used by software tools to support the developers of software. However, model transformations are developed by humans and as such can have errors. Errors in model transformations imply that a model can be transformed incorrectly, so errors in transformation can be transmitted to the software systems created by model transformation. Therefore analysing and evaluating the dependability of model transformation is vital to supporting model driven software development.

This thesis presents a novel technique to create models generators, to support model transformation validation. Meta model are the specification of modelling formalisms and are typically used to specify the input of model transformations. Usually, software models applied to model transformation are created with a great deal of manual interaction, to develop a software system. The presented technique creates a generative representation from a given meta model. The models generated by the presented technique do not represent meaningful software systems. Instead, the generated models apply to evaluate the dependability of model transformations. When generated models are used as input to a transformation, situations where the transformation works as expected are demonstrated; as well as plausibly discovering

situations where the transformation does not work as anticipated.

This chapter presents a summary of the objectives and the motivation for creating the presented thesis. An overview of the contributions is given along with the approach that was taken in the course of research. A structure for the thesis is presented along with a brief summary of each chapter.

1.1 Motivation

A transformation is a complex software system for converting models between modelling formalisms, created once and used to convert many models of software. There are many modelling languages and several model transformation languages. A transformation may be defined in any transformation formalism and between any pair of modelling formalisms. As with any software artefact that is man-made, a transformation can have errors. The automation benefits of model transformation and model driven software development are nullified if the transformation has errors. The evaluation of dependability of model transformation is essential to supporting model driven software development.

Several aspects of model transformations conspire to make model transformation quality evaluation a difficult proposition. Model transformations are complex software systems. The input of model transformation are structurally complex models, the input space is typically infinite in most modelling languages. In meta models, there is additional complexity in the logical constraints that arbitrarily define the elements and relationships allowed in valid model of the modelling formalism. To validate a model transformation requires the creation of very many models of software. Model generation automatically from meta models is difficult; modelling formalisms define complex, expressive notations for representing software. The objective of this work is to address the issue of generating models for automating part of the process of model transformation validation.

The overarching research questions that this thesis intends to investigate are:

- To what extent can the evaluation of model transformation dependability be automated?

Transformations must be analysed as part of the process of evaluation. Due to the number and complexity of model transformations, it is highly desirable to automate the evaluation process, where possible.

- Can the evaluation of transformation dependability be performed in-practice? Model Transformation tools are used in-practice by software developers. Any evaluation of model transformations should be possible using current computational capabilities, so that evaluation can be done in practice.

- How can the tools used in transformation evaluation also be put under evaluation? The evaluation of model transformation dependability may conceivably be solved by some software tools. As such tools are also software, the dependability of those tools must also be evaluated, some-how.

1.2 Approach

This work uses the term dependability to refer to the techniques of model transformation validation and verification. Model transformations are software systems where there must be a high degree of confidence in the correctness of the implementation. Correctness can be determined from two different perspectives: validation where the implementation is inspected to detect defects or verification where the implementation is compared against a specification of properties that must hold. In either case, the aim is to perform an evaluation of the correctness of the transformation. However, the term correctness has a much stronger mathematical definition and so is not used. Dependability may have different meaning depending on the context, however it is taken to mean evaluation of the correctness of model transformations in the current work. This section compares and contrasts the two approaches of validation and verification, justifying the use of the generic term dependability.

However, model generation techniques from the literature have apparent deficiencies. Techniques rely on interaction for the conversion of meta models to analysis formalisms. Ex-

pert intervention may also be required to guide analysis, depending on the analysis formalism used. In other techniques, automated analyses and conversions are used to generate models but the complex constraints and relationships of a meta model are not considered, so many invalid models are created. The most advanced techniques employ automated conversion, which can be seen as a form of model transformation. In such techniques, the dependability of the transformations used can not be evaluated by those techniques.

The current thesis intends to assist the validation of model transformation by generating models from complex meta models. Model generation does not consider the internal structure or formalism of the transformation, so model transformations in diverse languages may be validated. By application of best-practice from model driven software development, the method can be automated by exploiting existing analysis tools for the generation of models. However, as transformations are used in the technique, it must be possible to validate those transformations. To evaluate the technique, case study will demonstrate how the technique is applied in practice to assist model transformation validation. Further evaluation can draw a comparison existing state-of-the-art model generation techniques.

1.3 Contributions

The following is a summary of the contributions of this thesis.

- Results:
  - A unifying classification of model transformation dependability evaluation techniques, in chapter 3.
  - A novel technique to create model generators from meta models, using model transformation and exploiting software verification tools; in chapter 4.
  - The applicability of technique to assist the self-validation implementations of the technique, in chapter 5 and 6.

  - The evaluation of the technique by application to model transformation reliability evaluation and a comparison to existing state-of-the-art techniques, in chapter 6.

- Publications arising from this thesis :

  - [131] S. M. A. Shah, K. Anastasakis, and B. Bordbar. Using Traceability for Reverse Instance Transformations with SiTra. In Design and Architectures for Signal and Image Processing (DASIP 2008). Special Session on Formal Models, Transformations and Architectures for Reliable Embedded System Design., Bruxelles, Belgium, 2008.

  - [132] S. M. A. Shah, K. Anastasakis, and B. Bordbar. From UML to Alloy and Back Again. In Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa '09, pages 4:1–4:10, New York, NY, USA, 2009. ACM. **(Awarded best paper prize.)**

  - [133] S. M. A. Shah, K. Anastasakis, and B. Bordbar. From UML to Alloy and Back Again. In S. Ghosh, editor, Models in Software Engineering, volume 6002 of Lecture Notes in Computer Science, pages 158–171. Springer Berlin / Heidelberg, 2010. **(Invited paper extended version of the above.)**

1.4 Overview

This chapter has set out the overview of the objectives, motivation and contributions of this work. Model driven software development and software dependability concepts are used throughout, a brief introduction to relevant topics are given in chapter 2. The evaluation of model transformation dependability is important to the success of model driven software development but is a difficult task, as found in chapter 3. Chapter 3 also presents a classification of existing model transformation dependability evaluation techniques and proposes a unique selection of desirable features, towards a validation technique.

Chapter 4 presents a novel technique to create model generators from meta models, using

several model transformations. The technique requires a multi-stage implementation using advanced model transformation concepts, as described in chapter 5. The presented technique is evaluated in chapter 6 by several case studies and comparison to existing model generation techniques. The desirable features proposed in chapter 3 are used as the basis for the evaluation in chapter 6. Finally, a discussion is made on the outcomes of this thesis, along with possible advancement of the technique, in chapter 7.

The audience of this thesis is intentionally broad and several paths are possible when traversing this work. Readers acquainted to model driven software development and software dependability evaluation can avoid chapter 2. Chapter 3 presents a unifying classification of existing model transformation validation and verification techniques, motivating this work. Developers of software modelling and model transformation tools may focus on chapter 5 and chapter 6. Chapter 4 presents the proposed technique of this work so is considered essential in all cases.

To summarise, the structure of this thesis is as follows:

- **Chapter 1 - Introduction**.

  The current chapter introducing the thesis, outlining the topic of the work, contributions and the structure of this thesis.

- **Chapter 2 - Model Driven Software Development, Modelling, Model Transformation and Software Dependability**.

  Concepts from model driven software development that are of particular importance to this work. A very brief overview is given of verification and validation as software dependability evaluation methods.

- **Chapter 3 - Existent Techniques to Evaluate the Dependability of Model Transformations**.

  A discussion is made of the issues of evaluating transformation dependability. Furthermore, a review and classification of existing techniques for model transformation dependability evaluation is presented. A unique combination of desirable properties is discovered for a novel dependability evaluation technique.

- **Chapter 4 - A Technique for the Automatic Creation of Model Generators to Assist Model Transformation Validation**.

Presents the proposed technique evaluate model transformation dependability, by automatically creating a model generator from a given meta model.

- **Chapter 5 - Implementing Self-Validating and Repeatable Creation of Model Generators from Meta Models**.

  Describes the combination of advanced concepts required to create an implementation of the proposed technique.

- **Chapter 6 - An Evaluation of the Application of Model Generators to Model Transformation Validation**.

  The presented technique applied to model transformation dependability. Further evaluation is carried out by comparison to existing similar techniques.

- **Chapter 7 - Conclusion**.

  This chapter summarises the outcomes of this thesis.

CHAPTER 2

MODEL DRIVEN SOFTWARE DEVELOPMENT, MODELLING, MODEL TRANSFORMATION AND SOFTWARE DEPENDABILITY

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.
— Alan Kay

Design and programming are human activities; forget that and all is lost.
— Bjarne Stroustrup

2.1 Synopsis

This chapter gives context to this thesis by introducing the relevant concepts from model driven software development methodologies and software dependability. In model driven software development, models are a first class entity, used to represent software in an appropriate modelling formalism during development. A single abstract model is applied for a variety of software development purposes. Several modelling formalisms are available, each modelling formalism is suitable for a specific purpose. Models are converted from one modelling formalism to another by model transformations, to apply a model for numerous software creation purposes.

Evaluation of the dependability of software is a difficult task. To evaluate software dependability, the two generally disparate techniques of validation and verification are introduced. As with any complex man-made system, software is prone to defects that affect the normal intended operation. Software validation evaluates the reliability of a software system by demonstrating the system has some required functionality. Software verification evaluates the reliability of a software system by analysing the software system is consistent with regards to specified dependability properties. By detecting faults found in verification and validation, the errors in software systems can be corrected, improving the dependability of software.

2.2 Overview of Model Driven Software Development

In software development, several stages are required towards the creation of software. Model driven software development [135, 126, 19, 134, 101] is a relatively novel proposed software development methodology. As the name implies, abstract models form the basis for the creation of software systems. In existing software creation methodologies, artefacts produced at each stage of development have an indirect effect on the produced software. So the outcomes of requirements, specification and design stages have an informal role in creating the resulting software systems; usually facilitating communication between software developers [135]. The Model Driven Architecture [134, 101, 78] is a set of standards and the Eclipse Modelling

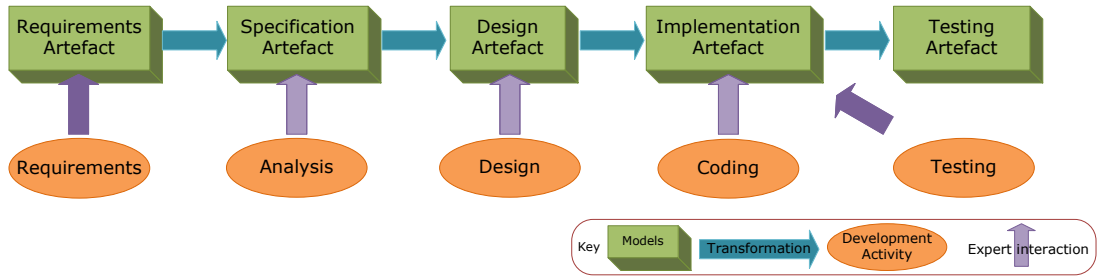


Figure 2.1: Ideals of the model driven software development methodology: models directly inform multiple stages of development.

Framework [137] is an implementation towards model driven software development. In model driven software development, each stage of software development is embodied in a model appropriate for the system and development stage. Models produced in a given stage of development directly inform the development stages that follow, by the conversion of models from one development stage to another. An out of model driven software development is shown in figure 2.1.

The are several benefits to model driven software development. Model driven software development brings tool support to each development stage including design, specification and requirements. Models in model driven software development are also directly effect the created software systems. Diverse modelling formalism exist, each suitable for a specific stage of development and software system under development. In model driven software development, a model in a given modelling formalism represents a software system. By transformation of models between modelling formalisms, a model can directly inform several stages of development. Model driven software development aims to bring automation and tool support to each stage used in software creation.

The modelling formalisms in the Unified Modelling Language "UML" [110] group and the related standards of Model Driven Architecture "MDA" [134, 101, 78] work towards model driven software development. Both the UML and the MDA are maintained by an industrial consortium, the Object Management Group. The UML is a standard collection of modelling formalisms for various development stages, used for the definition of models of software. The MDA defines a collection of related standards that support model driven software develop-

ment. The Object Constraint Language "OCL" [114] is used to create logical constraints over models, that define the subtleties of a software system. The Queries, Views and Transformations "QVT" [115] is a standard for the definition of model transformations, to convert models between modelling formalisms. The Meta Object Facility "MOF" is the standard used defining modelling formalisms, including the UML, QVT and OCL. The Models Driven Architecture adopts the XML Metadata Interchange "XMI" [111] format, as a physical format to exchange models between software modelling tools. The MDA standards are not implemented by any single provider, instead the standards are used by tool producers to create interoperable implementations. For details on the Unified Modelling Language, Model Driven Architecture and related standards, refer to the literature in [134, 110, 114, 115, 111].

The Eclipse Modelling Framework "EMF" [137] consists of several projects towards implementing software tools for model driven software development. The core of the Eclipse Modelling Framework implements a modelling hierarchy, based on, but subtly different to the Meta Object Facility standard. The Eclipse Modelling Framework also implements the Object Constraint Language, for the creation of logical constraints over models. Several EMF projects implement model transformation formalisms and tools, to support model driven software development. Models in the framework are interchanged via XML Metadata Interchange. The Eclipse Modelling Framework projects implement tools towards an integrated development environment for model driven software development. For details on the Eclipse Modelling Framework, refer to the literature in [137].

In the following sections the relevant concepts of model driven software development are explored, as a foundation for the topic of this research. In model driven software development, models are used to represent software systems, as described in section 2.3. A given software model conforms to a modelling formalism. Modelling formalisms are defined using meta models, as described in section 2.4. A key principle of model driven software development is the conversion of models from one modelling formalism to another, by model transformation, as described in section 2.5.

2.3 Modelling Software Systems

In model driven software development, models are a first class entity in the development of software [78, 137]. Several modelling formalisms are available for representing software as a model, each with features suited to a specific development purpose. Modelling formalisms are defined using meta models, which specify the allowed models of a modelling language. Modelling formalisms are available for several purposes including requirements, design, analysis and implementation. Models of software can be converted to apply in a different context by model transformation. Models, modelling formalisms and model transformation enable model driven software development.

Models of software are abstractions to describe only the problem under study. Visual and textual elements are used to appropriately represent the software system under development. This is contrast to software in textual programming languages, which typically require "boilerplate" code definitions and coding conventions, not related to the software developed [134, 101, 78]. A model abstractly represents only the developed software, so the constructs of programming languages are not necessary.

Modelling formalisms are created and used to represent software appropriately during development [82]. Static modelling formalisms are used to represent the fixed, structural parts of a software system. Examples of static models of software include class diagrams, architecture diagrams and object diagrams [110], a sample is shown in figure 2.2. Dynamic models are used to represent the behaviour of software systems. For example sequence diagrams, state machines or activity diagrams (also in [110]), abstractly representing the parts of a software that change over time. Several models, both structural and static models can be combined to model the different parts of a single software system. Modelling formalisms are specified by meta models, described in section 2.4. A variety of modelling formalisms are available for the representation of software, in an appropriate form.

A model is an abstract definition of the scenarios that are allowable in the developed software system [127]. A software model is formed of the variable elements and relationships allowed in the software system. Models can be instanced so that each element of the model

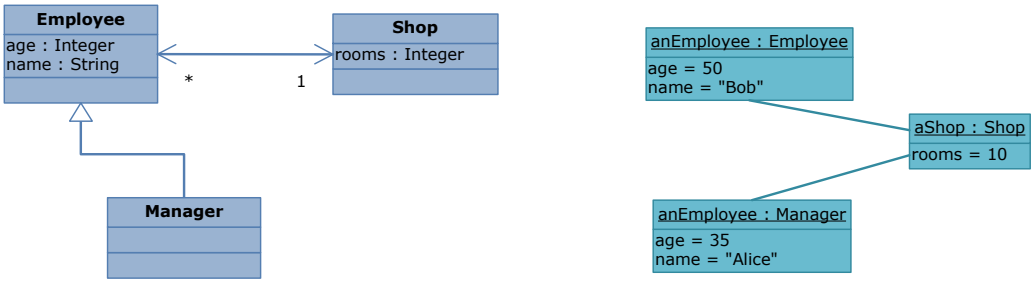


Figure 2.2: Sample class diagram, model of software.

is given some value. An instance of a model represents a particular scenario of the described software system. A model of a software systems typically defines a large number of allowed scenarios. To accurately represent the complexities of software systems, the elements of a model may be related subtle and arbitrary ways. In software modelling, logical constraints are used to express subtleties of software systems. Constraints are applied to static or behavioural models, to represent the intricate and arbitrary restrictions on the allowed scenarios in the software system represented. By abstract models and logical constraints, the permitted scenarios of a developed software system are specified.

Software modelling tools allow for viewing, editing and developing of software models in model driven software development. Modelling tools are designed to be specific to a particular kind of modelling formalism. Model driven software development tools can interoperate to exchange models via XML Meta-data Interchange [111]. Several model driven software development tools are available including the Eclipse Modelling Framework [137], ArgoUML [14] and AndroMDA [116]. Software modelling tools interoperate to allow the development of software models.

In model driven software development, models can be applied for more than one purpose. Although an abstract model does not apply as a working software system, an abstract model can be converted to an implementation of the software system. A single model may be converted to apply in multiple stages of development, including analysis, design or implementation. The conversion of models is also done via software development tools, that employ model transformation (described further in section 2.5). Model transformation tools allow for the conversion of models to support model driven software development.

This section has presented an overview of modelling and the role of models in model driven software development. Models abstractly represent the allowed scenarios in a software system, the instances of the model. Modelling formalism allow for the abstract representation of a wide range of software, including the structural and behavioural parts of a system. Model of complex systems can be restricted by arbitrary constraints on elements, relationships and values; to further define the allowed scenarios of a software system. Software development tools are used to support the creation of models of software. A single abstract model of a system can be applied for several purposes in model driven software development. Model transformation tools support the conversion of models between modelling formalisms. In model driven software development, abstract models are the central artefact for the creation of software systems.

2.4 Defining Modelling Formalisms by Meta Models

Model driven software development promotes model to become the central artefacts in the creation of software [135]. In the model driven software development methodology, meta models are used to define modelling formalisms [31]. A meta model defines the allowed models in a modelling formalism, specifying the allowed attributes and relationships of valid models in the formalism. A meta model is also a model: a model of the modelling formalism. A meta model as the abstract definition of a particular modelling formalism, several meta models are available, each one is suited to a particular purpose [31].

A model of a software system is created to conform to a modelling formalism. Abstract models are used to represent the allowed scenarios in software systems; instances of a software model represent particular allowed situations in the system. The elements allowed in a software model depend on the meta model that the model conforms to. Just as models describe the allowed elements and relationships in a software system, meta models describe the allowed elements and relationships in a modelling formalism. Models, instances and meta models are the hierarchy of modelling formalisms in model driven software development.

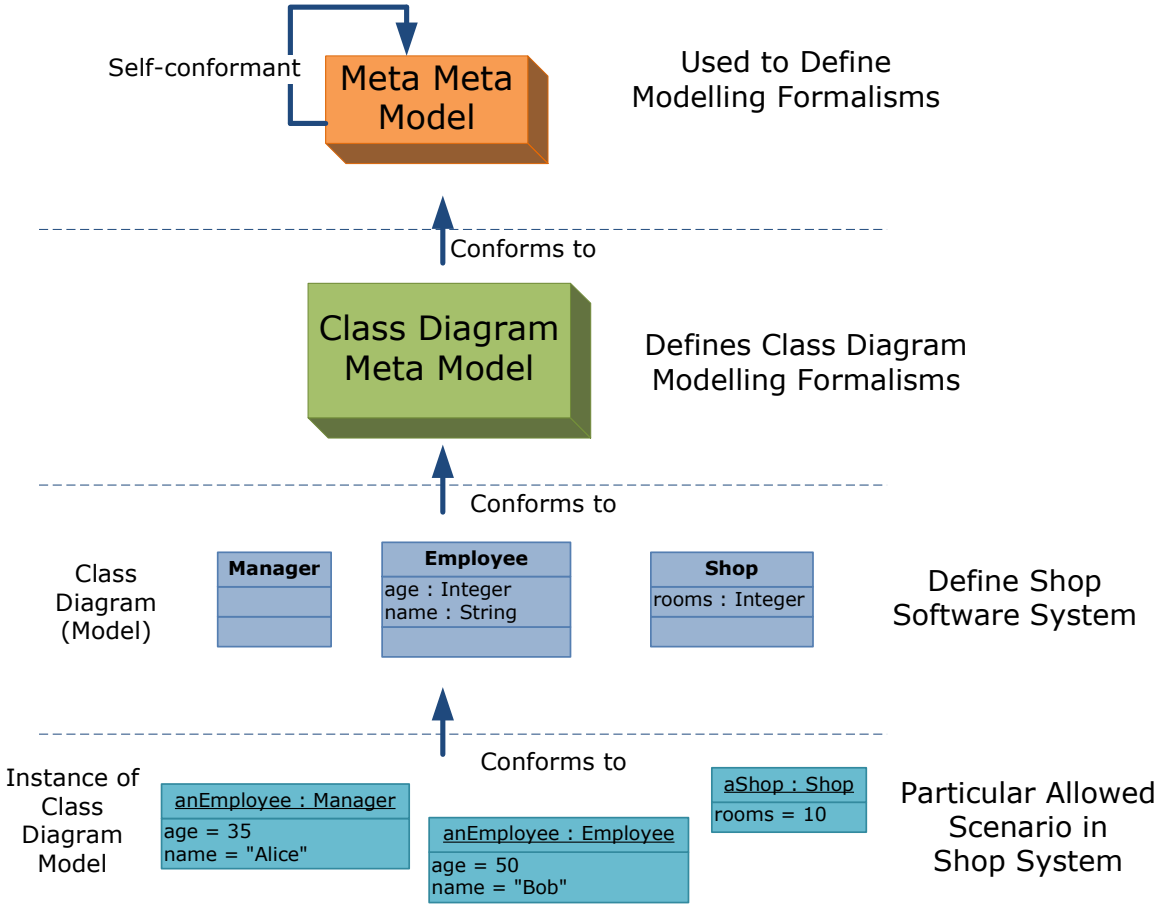


Figure 2.3: Example models in a meta modelling hierarchy for model driven software development.

Models in a modelling hierarchy are shown in figure 2.3.

Several modelling formalisms are available and each formalism is created as suited to represent a particular kind of software as a model. Meta models are created for static modelling formalisms such as class diagrams as well as dynamic modelling formalisms message sequence charts, state machines. In the definition of either static or dynamic modelling formalisms, the meta models are static. Meta models are used to define diverse modelling formalisms in model driven software development, as shown in figure 2.4.

In meta models, the elements allowed in valid models of a modelling formalism can be further restricted to allow for more subtle and expressive definitions within the formalism. As with models of software, meta models can have complex and arbitrary relationships between elements. Logical constraints restrict the allowed elements, relationships and attribute

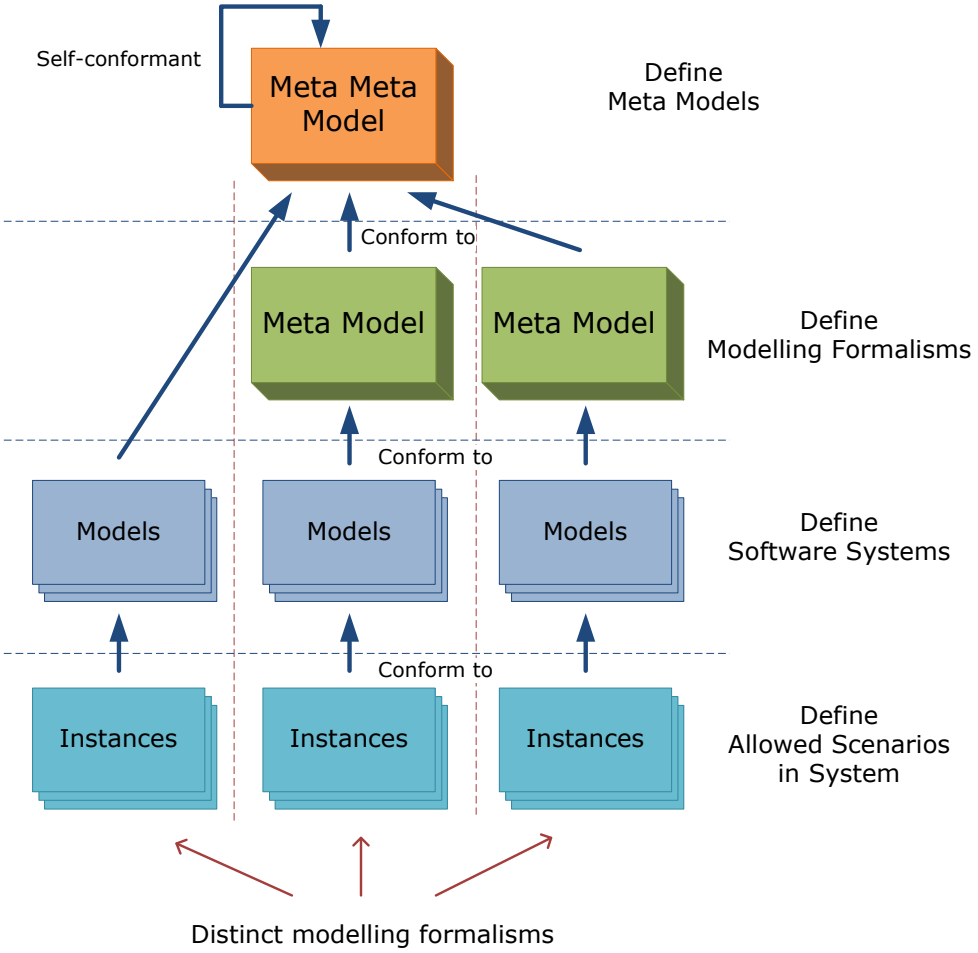


Figure 2.4: Three- and four-level modelling hierarchies in model driven software development.

values in a meta model. Similar to software models, meta models constraints in the Eclipse Modelling Framework and the Model Driven Architecture are created using the textual, first order logic based Object Constraint Language [121, 114]. Complex and expressive modelling formalisms are further defined by the application of constraints to a meta model.

Model driven software development relies on software development tools to support the creation of models. Meta models are used in model driven software development to define modelling formalism. As with models, meta models can be interchanged by XML Meta-data Interchange [111]. In software modelling tools, representations of software created using only the meta model elements of the modelling formalism. Software modelling tools use the meta model to allow developers to interactively create models of software systems. Using meta models, tools to support modelling and model driven software development are created.

Meta models are also models, as such meta models are defined using a modelling formalism: *the meta meta model*. In meta modelling, three or four level modelling hierarchies of modelling are possible, as shown in figure 2.4. A meta model for meta meta models is not necessary; a defining feature of meta meta models is the ability to self represent. That is, a meta meta model conforms to the elements defined in the meta meta model. Meta meta models define the modelling formalisms and terminate the modelling hierarchy in model driven software development. Further details of meta meta models can be found in [137, 113].

Several meta meta models are available. Notably, the Meta Object Facility "MOF" [113] is the common meta meta model used to specify the meta models of the group of modelling notations in the UML standard [110]. There are a several of implementations of MOF [137, 116, 14], as tools using the UML modelling notations are typically MOF compliant. In the Eclipse Modelling Framework "EMF" [137], ECore is the meta meta model. Modelling formalisms in the EMF projects typically conform to the ECore meta meta model. ECore is also used to create an implementation of the UML standard in the EMF [137]. ECore and MOF have similar roles in meta modelling hierarchies however, there are subtle differences between the two; details of the differences can be found in [71]. As ECore has a single canonical implementation [137], ECore is self-defining in practice, the ECore meta model is created such that it defines the ECore meta meta model.

In model driven software development, meta models define modelling formalisms. Numerous modelling formalisms are available, each formalism is suited to a particular purpose in representing software. Using meta models, software tools are created to support model creation, model interchange between tools and conversion of models between modelling formalisms. Complex modelling formalism are possible by the creation of constraints, to arbitrarily restrict the models that are valid in a modelling formalism. Meta models exist in a hierarchy of modelling and are defined by meta meta models, which terminate the modelling hierarchy for model driven software development. Meta models are an essential component of model driven software development.

2.5 Conversion of Software Models by Model Transformation

In model driven software development, model transformation convert models [130]. Models conform to a modelling formalism, defined by the meta model and each modelling formalism is suited to a specific purpose. A given transformation converts a model conforming to a source modelling formalism and create a model in a target modelling formalism. By conversion a model can be applied to several purposes towards software creation. Several model transformation approaches, notations and tools are proposed in literature. Rule-based mechanisms are a common means to define model transformations. In advanced application of model transformation, the transformation execution can be recorded by a trace, meta transformations can convert meta models, higher order transformations are used to create model transformations and chains of transformation can perform conversion via intermediate modelling formalisms.

In the Model Driven Architecture, model transformations are used to convert models between different stages of development. A model transformation is created between a pair of software modelling formalisms and used in the conversion of many models. An outline of rule-based model transformations is given in figure 2.5. By transformation, models of software can be refined, elaborated, re-represented, abstracted, reverse engineered and analysed by model transformation (adapted from [90]). For example, model transformation can convert an abstract design model to a corresponding implementation. Model transformation can also be used to several purposes with in a given stage of development. For example at the design stage of software development, a design model can be used to create an implementation model, analysis model and a documentary model of the same software system, by model transformation. Model transformations are an important concept in model driven software development, allowing multiple applications of a single model.

A model transformation tool takes a source model as input and applies the transformation to, produce a target model. Models transformation can be defined and applied by a diverse range of techniques and tools for the conversion of models. Special purpose programming

formalisms are also proposed for the definition of model transformation; applying some underlying model of computation such as graph transformation (VIATRA [145], GREaT [23]), relational (QVT relational [115], TefKat [94]), imperative (QVT operational [115], SiTra [2]) or hybrid approaches (ATLAS [72]). Model to model transformation, such as [115, 94, 32], and model to text transformation, such as (OMG M2T [112]), respectively produce models or textual software artefacts by transformation. Model transformations are classified as imperative, defining the steps required to convert models, or declarative, defining only the specification of the conversion. As well as special purpose model transformation notations, general purpose programming languages have also been applied to the creation of model to model transformation, such as Java in SiTra [2] and AndroMDA [80]. AndroMDA [80] also uses the general purpose Apache Velocity template language and tools [66] for model to text transformations. A comprehensive review of model transformation techniques is found in [47], a taxonomy of model transformation approaches in [100] and comparison of graph transformation techniques in [138].

In the Model Driven Architecture initiative, the Queries, Views and Transformations "QVT" [115] standard is proposed for model transformation. The standard consists of several model-to-model transformation formalisms, extending and based on the Object Constraint Language. QVT relational and core specify the formalisms for creating rule-based, declarative conversion of models. QVT operational specifies a formalism for creating rule-based imperative model transformations. The standard specifies that relational model transformations may be converted to operational transformations, to execute the transformation. Several implementations are available, SmartQVT [6] and Eclipse M2M [137] implement QVT operational and MediniQVT [99] implements QVT relational as plug-in projects in the Eclipse Modelling Framework [137]. Further details on QVT can be found in the standard [115] and in a review of recent developments in [81].

Model transformations frameworks are created to support the definition and application of model transformations. The Simple Transformer "SiTra" [2] model transformation frameworks consists is a library defined in the general purpose Java programming language. SiTra

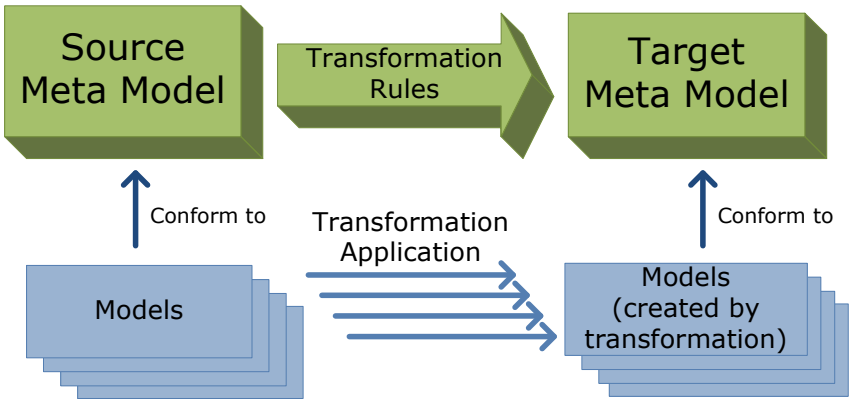
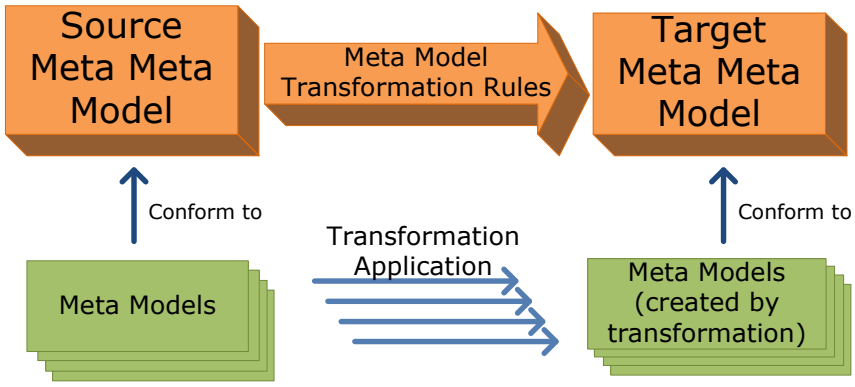


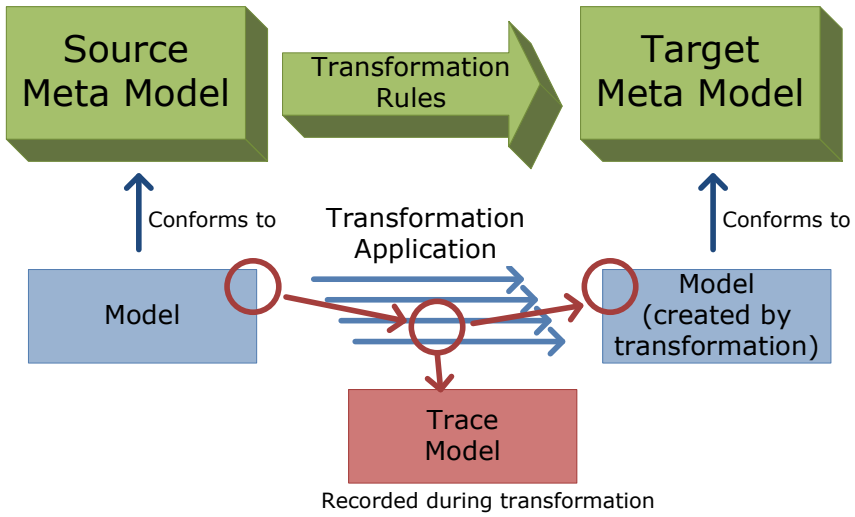
Figure 2.5: Overview of rule-based model-to-model transformation.

is a rule-based imperative-only transformation framework. SiTra can exploit Java libraries to perform model transformations using standard modelling notations such as EMF [137] or UML [110]. Rules in SiTra are defined as standard Java classes, implementing the provided "Rule" interface. The Rule interface consists of three methods, to define the conversion of a model element. A group of SiTra rules are applied to convert a model by the SitraTransformer implementation. SiTra has been applied to the complex transformation of UML State Machines to the VHDL integrated circuit language [3] and the challenging transformation of UML Class Diagrams with OCL to the Alloy software analysis language [11]. Further details of SiTra can be found in [2].

The ATLAS model transformation language [72] and supporting framework is implemented as a project in the Eclipse Modelling Framework [137]. ATLAS is rule-based, model-to-model transformation framework supporting both imperative and declarative model transformations. The ATLAS model transformation notation is a hybrid of relational and operational transformation approaches, is based on the Object Constraint Language. Several model transformations have been implemented in ATLAS, a comprehensive list is available in [20, 149]. Model transformations in the ATLAS language are applied in the ATLAS project to import models from different modelling formalisms. A model transformation definition in ATL is complied and applied to a model by the ATLAS model transformation virtual machine. ATLAS is a language and framework specifically created for the definition of model to model transformations. Further details of ATLAS can be found in [72, 83, 7].



(a) Meta model transformation.

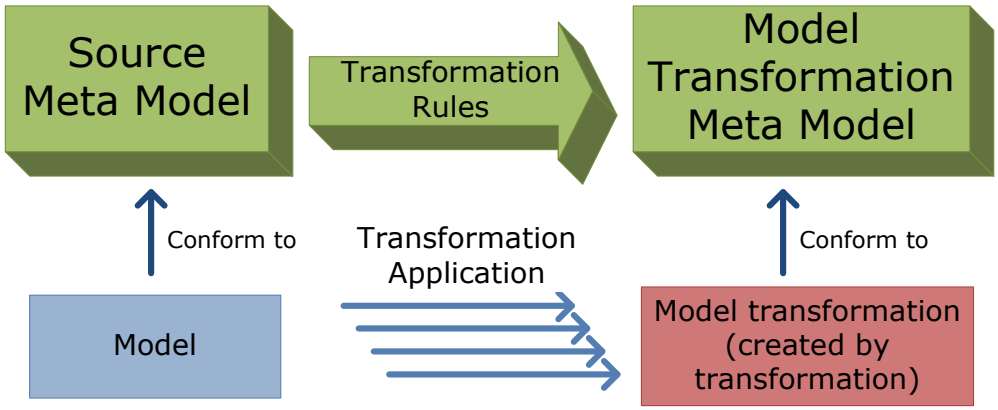


(b) Transformation execution tracing.

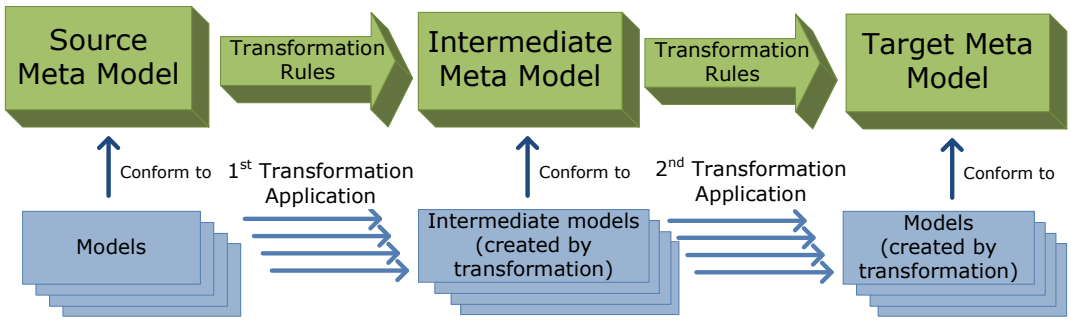
Figure 2.6: Model transformation: advanced applications.

Model transformation convert models form a source to a destination modelling formalism. Rule based model transformations are defined at the meta model level. That is, rules are defined for the elements of the modelling formalism. A rule is a specification of how a source meta model element is used to create a destination meta model element. A collection of rules between two modelling formalism form a transformation, to convert models from one formalism to another, as shown in figure 2.5. The architecture of model transformation relies on both input and output models conforming to the input meta model. When given model conforms to the source meta model, then the model can be converted by applying the model transformations rules to that model.

Several advanced applications of model transformations are proposed. As meta models are also models, model transformation can be created to convert meta models. Meta model



(c) Higher order model transformation.



(d) Chains of model transformation.

Figure 2.6: Model transformation: advanced applications.

transformations convert meta models between distinct meta modelling formalisms, as outlined in figure 2.6a. Meta model transformations create a bridge between meta modelling formalisms and between tools that support different meta modelling formalisms. Meta model transformations allow interoperability between meta modelling tools, as in the conversion of meta models from MOF to EMF [64], Coral to EMF [4] and MetaEdit+ [76] to EMF [77]. Also in the Eclipse Modelling Framework, meta model transformation are used to create software modelling tools automatically; where model editors are created automatically from a meta model [137]. Meta model transformation convert meta models and are used create tool support for model driven software development.

Model transformation tools can record the process of a model transformation [115, 70, 47]. When a particular model is applied to a model transformation, elements of the source model are converted by rules to create elements of the target model. Tracing a model transformation records the process of model transformation execution, so is specific to the model

and transformation used, as shown in figure 2.6b. The trace records which model element(s) of a source model are used to create which model element(s) of the created target model. Trace information is stored in model of the trace, either implicitly by the model transformation framework, or explicitly, by the transformation developer. The Queries, Views and Transformations [113] specification for model transformations defines how tracing a number of methods that query a tracing model. Model transformation tracing is a means to record the application of a given model to a model transformation.

In model driven software development, higher order model transformations are used to support the creation of software development tools [147, 47, 140]. Higher-order model transformations are defined for creating and converting model transformations, an outline is shown in figure 2.6c. A higher-order model transformation takes a model transformation as input or output (or both). Higher-order transformations treat model transformation definitions as a model. Several meta models for model transformation formalisms are available QVT [115], ATLAS [71]. SmartQVT [6] uses higher order transformations to convert an operational QVT transformation to a corresponding Java representation which performs model transformation. In the ATLAS [71] transformation framework, a higher-order model transformation creates a transformation artefact executable on the ATLAS virtual machine, from a human-readable ATLAS transformation definition.

Model transformation chains support the progressive conversion of models between diverse modelling formalisms [116]. Transformation chains exploit intermediate modelling formalisms toward the creation of final target model, an outline is shown in figure 2.6d. A chain of model transformations occurs when the output of a model transformation is used as the input of a following model transformation. A chain of model transformations can consist of several intermediate model transformations. By chains of transformations a model is converted by modular transformation steps to the required target formalism, via one re several intermediate transformations. Such transformation steps convert models progressively, rather than in a single complex transformation. In the Model Driven Architecture [134] initiative, chains of model transformation are used to progressively create software systems via

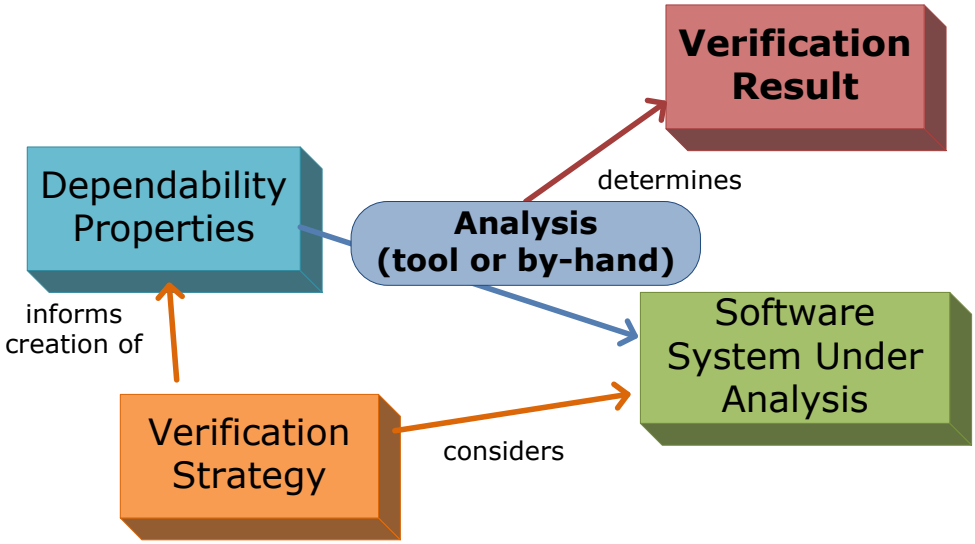


Figure 2.7: General overview of software verification against dependability properties.

conversion of Platform Independent Models to Platform Specific Models, and following to executable implementations [78].

Model transformations are the mechanism whereby models are converted from one modelling formalism to another. Model to model and model to text transformations allow modellers to simultaneously exploit the advantages of diverse software representation notations. Model transformations are applied to the automated refinement, elaboration, re-representation, abstraction and analysis of models, in model driven software development. Higher-order transformation convert model transformation, chains of transformations simplify and modularise complex model transformations, meta model transformation allow modelling formalisms to be converted and tracing records the application of transformation in advanced scenarios of model transformations. General purpose programming languages are applied to converting models, as well as special-purpose standards and implementations.

2.6 Software Dependability: Validation and Verification for Improving Reliability

A software systems is said to be dependable where is works as anticipated. Evaluating the dependability of software is a key challenge in computer science. Particularly for ensuring

that software tools used in the development of new software will work as anticipated. Dependability properties of software have be defined and classified as availability, reliability, safety, integrity, maintainability [22]. This work is concerned with one aspect of dependability: reliability; specifically, dealing with pre-emptive detection of errors in model transformation.

Software is analysed to search for defects (validation) and analysed with regards to some set of properties that must hold (verification). This section provides a high-level introduction to dependability evaluation, by verification and validation. The aim of the current discussion is not to be comprehensive or complete, instead a brief overview is given as a basis for later discussions. As with any complex man-made systems, software is prone to defects that affect the normal operation.

To evaluate the dependability of software systems, the two generally disparate techniques of validation and verification are applied to software. Software validation evaluates a software system by demonstrating situations with regards to the required functionality [30], an outline is shown in figure 2.8. Software verification evaluates the dependability of a software system by analysing the software system is consistent with regards to some specified dependability properties [52], an outline is shown in figure 2.7. By detecting and correcting faults found by dependability evaluation, software systems can be improved.

In software verification, specific properties of the quality of software are analysed without execution. A diverse range of verification techniques are available, each suited to a specific class of software and properties verified. Techniques can employ expert interaction to create proofs of certain properties using interactive theorem proving tools such as Isabelle/HOL [109], Coq [139]. Static analysis tools can be fully automated to analyse program code to determine certain properties [45]. Symbolic model checking [119, 43] and bounded model checking [33, 42] inspect either a complete or bounded model of the state-space of a program code, for conformance to manually specified logical properties. Bounded model finding [68] is a relatively novel approach to inspect a manually created static abstraction of the software system, within a bounds, for consistency against manually specified logical properties. A range of automated, manual and interactive verification techniques are available.

In verification, the software artefacts created during development are analysed with regards to certain properties and without execution. The result of analysis by verification technique can be used to improve dependability of software, by detecting and correcting the cause of errors. In model checking [33] and model finding techniques [68], a counterexample can be produced by the tool, to demonstrate that a verified property is violated. The counter example informs the developer as to how a potential error is triggered in the system. In static analysis tools, the statements that violate a property are used by a developer to determine erroneous statements [45]. Software verification is applied to analyse, evaluate and possibly to improve software dependability.

Each verification technique has advantages relative to other techniques. An appropriate technique must be selected considering the software and properties to be verified [52]. In proof checking, a great deal of manual interaction by experts in the technique is required [109, 139]. Interaction defines the proof goals and guides proof finding. In model checking, properties of very large or complex software artefacts can not be verified due to the large state-space [43]. In bounded model checking and model finding based verification, property violations not in the bounds can not be detected [33, 68]. An appropriate technique for software verification must be selected and applied by an expert in the formalism. A recent review of automated verification techniques can be found in [52].

In software validation, demonstration is used to evaluate the dependability of software [30]. Particular input data is applied to the artefacts created in software development to demonstrate a software system possesses the required functionality and features. The software validation process consists of the creation input data and the setting of an expected outcome when that data is applied to the software. When the created data is applied to the validation of a software, the outcome is observed and compared to the pre-set expected outcome. Validation demonstrates that a software works as expected by execution. Validation can uncover errors, where a given input fails to meet the pre-set expectation. Validation can be performed on various artefacts of a software system, from the atomic units of development to entire systems. Once errors are detected, the cause can be determined and corrected by

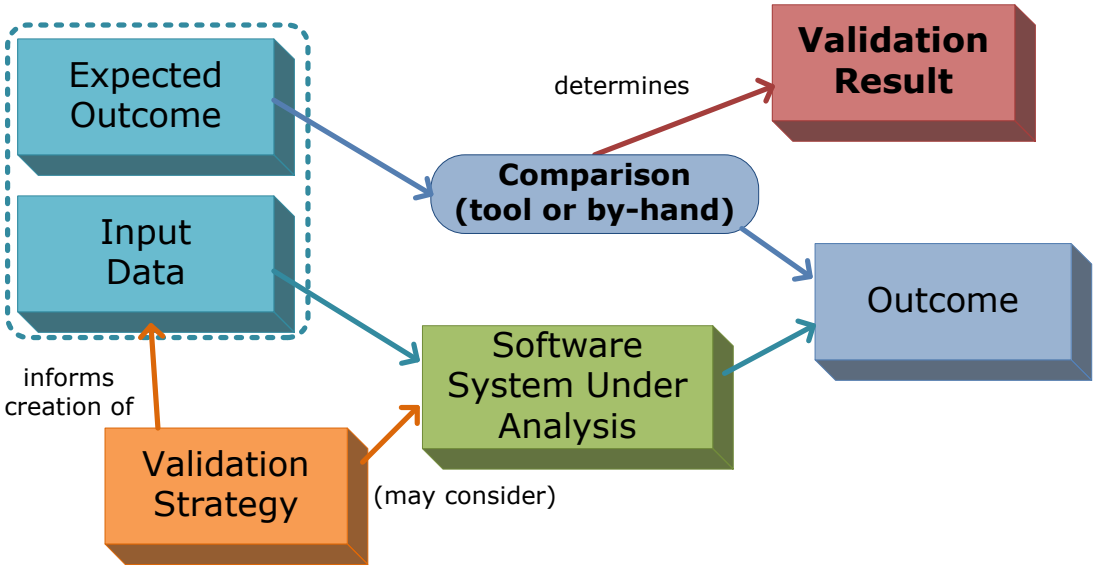


Figure 2.8: General overview of software validation by demonstration of software features.

expert interaction and debugging tools. Validation evaluates the dependability of software by demonstrating situations that work; and can improve dependability by error detection and correction.

The process of selecting input for use software validation is informed by the test selection strategy [104, 1, 34]. Test selection is performed by manual expert interaction. Several strategies are available, each suited to diverse validation aims. Functional validation treats the system validated as a "black-box" [29], creating input data without regard to the internal workings of the software. In contrast, structural or "white-box" validation designs the input data using knowledge of the internal workings of the software [104]. Data flow testing creates input data to exercise the flow of data in the software. Validation requires the creation of input data, informed by a test creation strategy, to evaluate the dependability of a software. For further details on selection test creation techniques, refer to the works of [104, 1, 34].

A key challenge in validation is the creation of test cases for use in the validation. Test cases are typically created by expert interaction. The process of creating test cases is highly time consuming and error prone in itself. The ideal is to generate the input data for validation automatically. This is problematic for as precise specifications of the complex input for software is not normally available [30]. Furthermore, creating test data automatically from an

input specification is difficult to automate, due to the large input space. Randomly generated test cases do not consider the objectives of the validation [104]. Advanced techniques to aid test data creation are either specific to a particular software system or only guide the manual creation of input data [30]. The creation of expected outcomes for validation is also challenging, as it requires accurate prediction of the outcome of a given test case. Creating test data takes time and expertise, and is difficult to automate, for further details on recent results in automated software validation refer to [30].

This section has introduced validation and verification to evaluate the dependability of software. Validation does not determine the absence of errors instead demonstrates the behaviour of a software system is as expected given some input. Validation applies input data to the execution of software, validation can detect errors where a system exhibits unexpected behaviour for some input. Several techniques are available to guide manual test data creation. Automated creation of input data is a key challenge in software validation. Verification establishes, by analysis, that a software system is consistent with regards to dependability properties. Several verification techniques are available, with varied levels of interaction and kinds properties verified. Validation and verification represent a diverse group of techniques used to evaluate the dependability of software.

2.7 Summary

This chapter has introduced the concepts of modelling, meta modelling and model transformation that constitute the model driven software development methodology. In model driven software development, models are used to abstractly represent the allowed scenarios in a software system. Models are created to conform to a modelling formalism that is appropriate for the software represented. Meta models are used to abstractly define modelling formalisms. Model transformations use meta models to define the conversion of models from one modelling formalism and are an essential component in model driven software development. Transformations automatically convert abstract model of software between modelling

formalisms, so models can be apply for several purposes towards the creation of software.

To ensure a software will work as anticipated, various validation and verification techniques are applied to evaluate software dependability. Verification analyses the conformance of a software to certain properties that indicate the software is dependable. Verification demonstrates particular situations of a software by creation and application of input data. Failure in verification and validation can indicate an error in the software system evaluated. After the cause of validation and verification failures is determined, the software can be corrected and the dependability of improved.

Model driven software development is considered further in the next chapter, where existing techniques for model transformation dependability are discussed.

CHAPTER 3

EXISTENT TECHNIQUES TO EVALUATE THE DEPENDABILITY OF MODEL TRANSFORMATIONS

What Descartes did was a good step. You have added much several ways, and especially in taking the colours of thin plates into philosophical consideration. If I have seen a little further it is by standing on the shoulders of Giants.
— Isaac Newton to Robert Hooke

3.1 Synopsis

Several techniques have been proposed for the evaluation of the dependability of model transformations. This chapter gives the motivation for creating the current thesis on model transformation dependability evaluation. Motivation is found by a discussion of the importance, unique issues of transformation evaluation; and the features and deficiencies of existent techniques. The conclusion is the definition of a unique combination of features for a promising model transformation dependability evaluation technique.

Two generally disparate approaches to evaluation of the dependability of model transformations are considered; that of verification and validation. Analysis of such methodologies, as currently applied, finds issues with practicality and assurance of dependability. By comparison and analysis of the existing methods, a unifying classification of the techniques is found. Discussion of the approaches shows an intersection of features that defines a promising technique for validation of model transformation dependability.

3.2 Dependability: Validation and Verification of Model Transformation Correctness to Improve Reliability

Evaluating the dependability of developed software is a key challenge in computer science. Particularly for ensuring that software tools used in the development of new software will work as anticipated. This work is concerned with one aspect of dependability: reliability; specifically, dealing with pre-emptive detection of errors in model transformation. Other dependability properties of software have be defined and classified as availability, reliability, safety, integrity, maintainability [22].

Model transformations are software systems where there must be a high degree of confidence in the correctness of the implementation. Correctness can be determined from two different perspectives: validation where the implementation is inspected to detect defects or verification where the implementation is compared against a specification of properties that must hold. In either case, the aim is to perform an evaluation of the correctness of the trans-

formation. However, the term correctness has a much stronger mathematical definition and so is not used. Dependability may have different meaning depending on the context, however it is taken to mean evaluation of the correctness of model transformations in the current work. This section compares and contrasts the two approaches of validation and verification, justifying the use of the generic term dependability.

There are several harmonies between validation and verification. First and for-most validation and verification are evaluations of software dependability, particularly, concerning the assessment of transformations for the improvement of dependability. The processes of validation and verification are concerned with assuring quality; the need for such evaluation implies that the product is imperfect, so possibly has defects.

In both validation and verification, the transformation is analysed and a result provided about the dependability [144, 56]. There are several means to accomplish this task. For example the transformation evaluated is typically an "operational" artefact: executable code is evaluated against a "declarative" artefact: a specification of the system. Also, the transformation specification can be the artefact that is evaluated, either with regards to a separate declarative specification or evaluated for consistency against a generic set of properties [91]. Furthermore, the "declarative" specification may describe the transformation as logical formulae [38] or the definition of the input of the system under evaluation [58].

One important differentiator between verification and validation is the scope of the process. Validation aims to show that the transformation possesses an intended feature, by one or several examples [89, 58, 75]. Validation of transformation can not universally guarantee the property [51], unless every case is considered. This is in contrast to verification, which analyses a software artefact (without execution) to show that some property holds for that software. Verification aims to prove a chosen property holds, for the entire transformation [97] or a with a pre-specified bounds [10].

This work is on the definition of a technique to increase confidence in model transformation correctness. There are benefits to both validation and verification techniques. However, model transformation correctness literature use the terms in a range independent and inter-

changeable ways [38, 89]. To avoid confusion, the following discussion is made without a referring to either verification or validation. Instead, the discussion is on evaluation of the dependability of model transformation.

3.3 Importance of Transformation Dependability Evaluation

Several studies have noted that model transformation dependability assurance is a similar problem to that of program complier dependability [47, 75, 59, 136]. As with compliers, model driven software development techniques intend to raise the level of abstraction at which software is created. For the acceptance of modelling as a development technique, there must be a confidence that the transformations will work as expected [59, 103, 38]. By ensuring the dependability of a transformation, a developer will be more willing to invest time and effort in developing models which rely on transformation. Dependability of model transformations is vital to acceptance of model driven software development methods.

Model transformations are used in software tools used to support model driven software development. For example, the Eclipse Modelling Framework uses model transformations for the import and export of models [137, 64]. A given transformation may also define model transformation as the input or output, such transformations are termed *higher-order* [147, 140]. For example, as part of the implementation SmartQVT defines a model transformation from the QVT standard to a Java transformation artefact [6]. When dependability of a model transformation can not be evaluated, high quality tool support for model driven software development is a difficult and time consuming task.

It has been noted that model transformations without assurance of dependability implies that the results may be invalidated by errors in the transformation [118, 97, 60]. Model transformations may be used in various scenarios towards creating software; Lano [90] notes models may be refined, elaborated, re-represented, abstracted and analysed for dependability by model transformation. If model transformations with errors perform these activities, the

benefits of automating the processes is negated, as the quality of the produced model is put into question.

In chains of model transformation, the output of one transformation is used as the input for other transformations [116]. If a transformation in the chain of transformation introduces a defect, any proceeding transformation may transmit the defect into the resulting software [88]. The defects of any single transformation in the chain can effect the validity of a chain of transformations, regardless of whether any defects exist in any other transformation in the chain. Therefore, it is imperative that model transformation dependability is evaluated, to support the development and dependable of model transformation chains.

3.4 Difficulties of Model Transformation Dependability Evaluation

Model transformations are software artefacts, ensuring dependability of transformation has similar goals to ensuring the dependability of software created using non-model driven development techniques. However, there are several fundamental differences between evaluating the dependability of model transformations and that done for existing software. This section discusses the specific difficulties of ensuring model transformation dependability.

As was previously explained in chapter 2, a defining feature of model transformation is the inputs and outputs are in the form of models [47]. Models are more complex than the input of common software systems [59, 56, 36, 151]. Models contain data, but also classes of data and relationships between classes which often mean complex structure of the input. Because of the complexity, the input of any model transformation is potentially infinite [27], making dependability assurance for transformations in general a difficult proposition. With such a large input space, an automated technique would be ideal for aiding dependability assurance. That is, an automated technique should reduce the effort of manually assuring dependability of transformations.

The input complexity of model transformation is not only structural, the input of model

transformations is also typically constrained [10, 58, 56] by sentences in the Object Constraint Language [114]. Such constraints define part of the assumptions of a given transformation. Any technique must take the constraints that define the input of a transformation into account when providing an assurance of dependability. This is particularly relevant to automated validation techniques. With a complex input space for transformations, any dependability assurance technique should be practical: computable using finite resources.

The size of the model transformation artefact under evaluation must also be considered. Model transformation definitions in either form, "operational" or "declarative", can be large unwieldy [67, 144]. Techniques for dependability assurance must take into account the potentially size of the transformation, allowing for dependability assurance in large and small transformation (scalability). Furthermore, phasing and rule [83, 46] mechanisms can be used to define model transformations in a modular, systematic way. The size of transformations is a further reason for automation of dependability assurance. A dependability evaluation technique should allow for the systematic assurance of dependability of not only the whole transformation but also the constituent parts of a transformation.

Another consideration in dependability is the heterogeneity of available transformation systems [47] used in the definition and execution of model transformation. There are dozens of transformation systems, from those based on general purpose languages [2, 87], to those specified by graph transformation [50, 38] and transformation systems based [83, 136, 79] on the OCL standard [114].

Ideally, techniques to evaluate to the dependability of model transformation, without regards to the language used. This is problematic, particularly in dependability techniques that interpret the internal working of a transformation ("white-box" approaches); no single transformation formalism is widely accepted [89, 47]. Techniques can exploit the modelling formalisms used as input to the model transformation, without regards to the transformation implementation ("black-box" approaches). Such "black-box" approaches have the advantage that model transformations that are unsupported by the technique because the modelling formalism is different, can become supported via modelling formalism "bridges", such as [64, 77, 4].

Bridges are converters that can be used to convert models as-is, between different modelling formalisms.

Conceivably, model transformation dependability can be evaluated for a single model transformation application, when applied to certain models. For example, in [107, 144] the dependability assurance is specific to a single transformation and the input and output models used in the analysis. This is useful as certain properties such as preservation of model properties by the transformation can be analysed. However, it has been noted that this type of analysis may not be easily applied to the analysis of other model transformations, [15] as model properties may not be persevered in all model transformations (also noted in [91]). Furthermore, properties of models can not be stated generically as the diverse models in a modelling formalism can have diverse properties. So the properties must defined in advance for each pair of models used such analysis [15]. Ideally, a technique for ensuring model transformation dependability should be applicable to a wide range of transformations. This can be achieved by assuring dependability of a wide range of models, instead of specific models.

Model transformations are subject to change. For example, transformation may have errors or be lacking features that require development of the transformation. Modelling languages are software too, that is they can change [108, 123]; as can model transformations. Changes to either model transformation definition or modelling languages used would invalidate any previously conducted dependability evaluation. Thus, any validation technique for model transformations must be adaptable to changes in the transformation and modelling formalism(s) analysed. The technique should not require a great amount of effort for adaptation and re-application.

3.5 Comparison of Model Transformation Dependability Techniques

Table 3.1 presents a classification by features, of available model transformation dependability evaluation techniques. The majority of the techniques require semi-automated analysis in the application. Advanced of the semi-automated techniques, for example those using mechanised proof finders [91, 118, 95, 40], necessarily require training and expert interaction in producing results of analyses. Several advanced techniques [10, 38, 24, 27] rely on bounding the state-space for the verification of a given property; less interactivity at the expense of the breadth of analyses. Either concession, bounding or manual interaction, are undesirable because such analyses would still require specialist training or would only apply to very simple model transformations, respectively.

Clearly a fully automated dependability analysis technique is desirable. For example, [97, 144] make use of model checking tools to automate verification of model transformation specification. The technique of [144] is notable as it is purely "black-box", requiring no knowledge of the inner-workings of the transformation and using model checking. However the technique in [144] verifies only properties of only specific pairs of models used in transformation. In [97], the technique is "white-box" and so relies on assumptions about the form of the transformation specification, reducing the applicability to the given formalism. Model checking based techniques such as [97, 144] are also limited in that either the transformation state space must be finite [97] or the analyses may never complete, due to the space complexity [144].

Another method used in fully automated dependability evaluation techniques is that of [55], to derive a complete and correct application, given a transformation specification and a particular model. The technique is focused on synthesis of "operational" steps towards the implementation of a transformation from the specification. The technique depends on a correct specification in-advance, along with a correct instance graph. The technique in [55] is "white-box", also relying on assumptions about the form of the transformation specification. It is not clear how this technique may be applied to already existing transformations.

In several of the automated [97, 144, 55] and semi-automated [10, 91, 118, 95, 24] techniques,

the artefact used in analysis is a declarative specification of the system under evaluation. In some cases, the aim is to verify universally that certain properties of the specification of transformation hold [97, 91, 118, 95, 24]. In other cases, the focus is to verify that the property holds in some subset of cases [10, 144, 55]. The difficulty in either kind of verification is the reliance on the existence of a transformation specification, and further, on the existence of the property verified. A transformation specification does not directly convert models, instead is interpreted or converted. This is problematic as the specifications may not reflect how the specification is interpreted. For example if not all pertinent properties are verified or the properties verified are erroneous, the utility of these approaches can be diminished.

In several approaches, the artefact used in analysis is the operational artefact of the system, such as an executable formalism [58, 56, 27, 59]. In effect, the system is validated by analysis based on specific cases. The difficulty in these techniques is deciding how many and which cases to use, because the results of the validation is only applicable to the cases attempted. The advantage is the practicality of the approach, permitting some analysis over normally difficult-to-verify executable code. A further advantage is the ability to detect faults and misapplied specification, by producing specific cases that violate the specification [58].

In the techniques that perform code-based analysis of model transformation, there are some fully automated techniques, requiring little intervention [27, 58, 56]. The philosophy in these techniques is broadly similar: generate models from a precise definition of the input (the input meta model) and use the generated models as test cases to exercise the transformation. A defining feature of model transformations is the reliance on input and output meta models [47]. In the such techniques, the meta model is treated as specification of the input and the techniques are naturally "black-box"; without interpretation of the inner workings of the transformation.

In [56], the automated creation of instances is done by encoding meta-models as generative graph grammars. This draws an interesting parallel with string grammars, used for creating program sentences that can be used to test program compilers [47, 75, 59, 136]. The approach is limited in that constraints of the meta model and complex associations between meta

elements are not considered in the creation of instances, which may lead to very many useless test cases. A further concern in this proposal is that the creation of instances is exhaustive, that is for large meta models, many instances may be created not allowing for systematic creation of instances [56].

In [58], instance generation for testing is done by the use of a logical and constructive encoding of meta-models and constraints, that can generate instances automatically. An interesting feature of the technique is the ability to request, in-advanced, certain formally specified models, allowing for systematic testing. The technique is notable in that a specific example is given of how a generated model might find an error in a transformation.

However, in [58] the process of converting the constructive representation of a given meta-model, constraints and generation-requests is done by hand, in Prolog. This may be problematic as the process of the conversion (manual or automated) to the constructive system may also be erroneous, as any transformation. Furthermore, the technique in [58] requires some manual "rule-of-thumb" developer intervention when encoding certain kinds of constraints, to avoid state-space-explosion.

Table 3.1 shows several conceptual techniques are presented in literature. The techniques are varied and not discussed in depth because of obvious limitations to transferability to other model transformations. Furthermore, the majority of the techniques are "white-box", relying on assumptions about the form of the transformation that is analysed. However, such techniques may still inform the development of automated techniques.

For example, several conceptual techniques propose frameworks for supporting dependability evaluation techniques [96, 79, 88, 87, 89, 41]. Of note amongst these are [79] and [88] as they present actual flaws found by the technique. Another notable computable technique in this class is presented in [103]. The work is on the "black-box" selection of a set useful test cases from a larger set based on known faults, by synthetically injecting faults into transformations. Other conceptual techniques such as [54, 85, 41] are useful as they define sets of particular properties that may be used to manually assess the dependability of a transformation.

In summary conceptual, mostly manual techniques provide insights for the processes and research direction of semi-automated techniques. The semi-automated and fully automated techniques discussed are highly desirable but some concessions must be made to allow for practical dependability evaluation. For example, requiring expert interaction or application to only simple transformations or limiting the scope of the analysis. A notable and common trend is automated and semi-automated techniques is the conversion to a different formalism to perform the analysis. This is both desirable, exploiting powerful known analysis methods, and simultaneously undesirable requiring a manual or automated conversion, which as with any transformation, may be erroneous.

3.6 Classification of Model Transformation Dependability Evaluation Techniques

This section gives a summary of the literature on model transformation dependability, presented as a feature table. Note that the feature comparison in the table is done using results in the respective published articles; empirical evaluation is not possible due to unavailability of implementations of each technique. For the comparison of techniques, the following features are identified:

- Analysis Automation
  - **Automated** : Major components of the analysis are automated, requiring minimal analyst intervention.
  - **Semi-Automated** : The analysis has some automated component(s) but requires some analyst intervention.
  - **Conceptual**/Manual : The analysis requires mostly manual application.
- Analysis Artefact
  - **Specification** : The technique analyses specification of the transformation.
  - **Code** : The technique analyses the executable code of the transformation.
- Other Artefact of Analysis
  - **Meta-model** : The technique analyses meta model(s) of the transformation.
  - **Instance** : The technique analyses model instances produced by the transformation.
  - **Annotations** : The technique requires annotations to augment artefacts of the transformation.

- Mechanism
  - **Conversion (auto.)** : The technique requires an automated conversion.
  - **Conversion (man.)** : The technique requires a non-automated conversion.
  - **In-place** : The technique analyses artefacts directly.
  - **Static** : The analysis is performed by only inspection of the software artefact, in an external interpretation.
  - **Dynamic** : The analysis is performed using the application of a software artefact, using the system under inspections' own interpretation.
- Level
  - **White-box** : The technique uses interprets the internal structure of the artefacts to perform analysis.
  - **Black-box** : The technique only uses the externally published knowledge of the artefacts to perform analysis.
- Oracle
  - **Models** : The outcome of analysis must be compared with oracle model-instances.
  - **Meta Model Conformance** : The conformance to meta models or external tools are used as oracles.
  - **Models' Properties** : Specific properties of the outcome of analysis are given as oracles.
  - **Generic Criteria** : Success is measured by simple criteria such as termination of analysis.
  - **Manual** : Success is measured manually by the analyst.
- Computability
  - **Bounded** : The analysis computable but analysis is bounded.
  - **Guidance** : The analysis computable but analysis requires some manual intervention to complete.
  - **Complete** : The results of the analysis are computable for every transformation.
- Validity
  - **Formal** : The analysis technique is validated by proof.
  - **Semi-formal** : The analysis technique is validated.
  - **Example** : The analysis technique is demonstrated by an example.
- Error Detection : The technique has been shown to detect errors in model transformations.

Technique	*Analysis Automation*			*Analysis Artefact*		*Other Analysed Artefact(s)*			*Mechanism*					*Level*		*Oracle*					*Computability*			*Validity*			
	Automated	Semi-automated	Conceptual	Specification	Code	Meta-model	Instances	Annotations	Conversion (auto.)	Conversion (man.)	In-place Interpret.	Static	Dynamic	White-box	Black-box	Model	MM Conformance	Model Properties	Generic Criteria	Manual	Bounded	Manual guidance	Complete	Proof	Semi-formal	Example	Errors Detected
Lúcio et al. [97]	●	○	○	●	○	○	○	●	○	○	●	○	●	●	○	○	○	○	●	○	○	○	●[1]	●	○	●	●
Varró and Pataricza [146], [144]	●	○	○	●	○	○	●	○	○	●	○	○	●	○	●	○	○	○	●	●	○	○	●[2]	○	○	●	○
Ehrig et al. [55]	●	○	○	●	○	○	●	●	○	○	●	●	○	●	○	○	○	○	●	○	○	●	○	○	○	●	○
Fiorentini et al. [58]	●	○	○	○	●	●	○	●	○	●	○	○	●	○	●	○	●	○	●	○	●	●	○	○	○	●	●
Ehrig et al. [56]	●	○	○	○	●	●	○	○	●	○	○	○	●	○	●	○	○	○	○	○	○	○	●	●	○	○	○
Poernomo and Terrell [118], [117]	○	●	○	●	●	●	○	○	○	●	○	●	○	●	○	○	○	○	●	○	○	●	○	●	○	●	○
Anastasakis et al. [10]	○	●	○	●	○	●	○	●	●	○	●	○	●	●	○	○	●	○	●	○	●	●	○	○	○	●	●
Ledang and Dubois [95]	○	●	○	●	○	○	○	●	○	●	○	●	○	●	○	○	○	○	●	○	○	●	○	○	○	●	●
Cabot et al. [38]	○	●	○	●	○	○	●	●	●	●	○	○	●	●	○	○	●	○	○	●	●	●	○	○	○	●	○
Baresi and Spoletini [24]	○	●	○	●	○	○	●	●	○	●	○	○	●	●	○	○	○	●	○	●	●	●	○	○	○	●	○
Calegari et al. [40], [39]	○	●	○	●	○	●	○	●	○	●	○	●	○	●	○	○	○	○	●	○	●	●	○	○	○	●	○
Rafe and Rahmani [120], [26]	○	●	○	●	○	●	○	●	●	○	○	●	○	●	○	○	○	○	○	●	○	○	●	○	○	●	○
Lano [91], [92, 93, 90]	○	●	○	●	○	●	○	●	●	●	○	●	○	●	○	○	○	○	●	○	○	●	○	○	○	●	○
Sani et al. [124]	○	●	○	●	○	●	○	○	●	●	○	○	●	●	○	○	○	○	○	○	●	○	○	○	○	○	○
Baresi et al. [25]	○	●	○	●	○	○	●	○	○	●	○	●	○	●	○	○	○	○	○	○	○	○	●	○	○	●	○
Asztalos et al. [18], [15, 16, 17]	○	●	○	●	○	○	○	●	●	●	○	○	●	○	●	○	○	○	●	○	○	●	○	○	○	●	○

●: the feature is present. ○: the feature is not present. Continued on next page...

[1] model transformation state-space must be finite

[2] subject to state space explosion

Technique	*Analysis Automation*			*Analysis Artefact*		*Other Analysed Artefact(s)*			*Mechanism*					*Level*		*Oracle*					*Computability*			*Validity*			Error Detection
	Automated	Semi-automated	Conceptual	Specification	Code	Meta-model	Instance	Annotations	Conversion (auto.)	Conversion (man.)	In-place Interpret.	Static	Dynamic	White-box	Black-box	Model	MM Conformance	Model Properties	Generic Criteria	Manual	Bounded	Manual guidance	Complete	Proof	Semi-formal	Example	
Baudry [27], [129, 128]	○	●	○	○	●	●	○	●	●	●	○	○	●	●	○	○	○	○	○	○	●	●	○	○	○	●	○
Wang et al. [151], [150]	○	●	○	○	●	●	○	○	●	○	○	○	●	●	○	○	○	○	○	●	●	●	○	○	○	○	○
Fleurey et al. [61] [59, 36]	○	●	○	○	●	●	○	●	○	○	●	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○
Narayanan and Karsai [106], [1]	○	●	○	○	●	○	●	●	●	●	○	○	●	○	●	○	○	●	○	○	○	●	○	○	○	●	○
Darabos et al. [49], [48]	○	●	○	○	●	●	○	○	○	○	●	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○
Kolovos et al. [79]	○	○	●	○	●	●	○	●	○	○	○	○	●	●	○	○	○	●	○	●	○	○	○	○	○	●	●
Küster et al. [88]	○	○	●	○	●	○	○	○	○	○	○	○	●	●	○	○	○	○	●	●	○	○	○	○	●	○	●
Lin et al. [96]	○	○	●	●	○	○	○	○	○	●	○	○	●	●	○	○	○	○	○	●	○	○	○	○	○	●	○
Küster [85], [86, 84]	○	○	●	●	○	○	○	○	○	○	●	●	○	●	○	○	○	○	●	○	○	○	○	●	○	●	○
Cariou et al. [41]	○	○	●	○	●	●	●	○	●	●	○	○	●	●	○	○	○	●	○	●	○	●	○	○	○	●	○
Mottu et al. [103], [13]	○	○	●	○	●	●	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	●	○	○	○	○
Ehrig et al. [54], [50]	○	○	●	○	●	●	○	○	●	●	○	●	○	●	○	○	○	○	●	○	○	○	○	○	○	●	○
Küster and Abd-El-Razik [87]	○	○	●	○	●	●	○	○	○	○	○	○	●	●	○	○	●	○	●	○	○	○	○	○	○	●	○
Lamari [89]	○	○	●	○	●	○	○	○	○	○	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○

Table 3.1: Features of model transformation quality evaluation techniques.

●: the feature is present. ○: the feature is not present.

[1] [105, 107, 74, 75]

3.7 Basis for Model Transformation Validation Technique

Validation and verification have similar aims but work in opposing ways. The objective of validation is to demonstrate the dependability of a transformation by exercising the scenarios where a transformation work as expected. So by validation, scenarios may be found that cause error in the transformation. In verification, the transformation is compared for consistency against a specification. By verification, the conformance to certain pre-set dependability properties can be found. Both validation and verification are applied to the evaluation of transformation dependability. This section justifies the selection validation, whilst exploiting verification tools, as a basis for the current work; due to the automation, apparent scalability and wide-applicability.

In the advanced techniques for transformation dependability evaluation, several methods automate the analysis involved. The analysis is automated or semi-automated by some analysis tools: the Coq proof checker in [118], analysis in Alloy in [24, 27, 10], or CSP [41]. Model transformation verification is problematic due to the amount of manual interaction required by experts in the formalisms either to create and abstraction or to guide the analysis [118]. Manual interaction is also required to specify the properties of the transformation verified [118]. Advanced validation techniques aim to automate the generation of models. Validation techniques can exploit the meta model as a specification of the input of transformation, to generate models [56, 27, 58]. Analysis is still required to generate models, where verification techniques can be applied.

In the existing techniques for transformation dependability evaluation, analysis is carried out in a separate formalism. In several advanced methods, the conversion to analysis formalism is either fully or partially-automated [10, 56, 27, 58, 120, 38]. However, the evaluation and assurance of dependability of the conversions is not often considered. In several techniques, some property of the conversion using in the technique is analysed separately and manually, for example as in [56] and [120]. In several techniques, model are generated from meta models [56, 27, 58]. However these techniques are unable to generate meta models for self-validation. The conversions used by quality evaluation techniques can be consider as

or are actual model transformations; for example [10] utilises a transformation from UML to Alloy and [41] uses UML to CSP. In a given technique, if the transformations used in dependability evaluation are not evaluated, the outcome of the evaluation of the technique may be flawed.

In verification techniques for dependability analysis, the results of analysis do not adapt to changes in the verified transformation. Verification techniques analyse consistency with regards to a specification [118, 10, 120, 38]. However, if the transformation is found be inconsistent against the specification, changes are required to correct the inconsistencies. Changes to the verified transformation mean that any of the verification already carried out is invalidated, and must be repeated. As several verification techniques [118, 10, 120, 38] require some expert interaction to guide the analysis, repeating verification is problematic. After changes to a transformation, any of the validation already carried out is similarly invalidated. However, to repeat the analysis done in the validation of a transformation simply requires the re-application of the previously generated models used for validation [56, 27, 58]. In contrast, when verification is repeated, the time-consuming and costly expert interaction required to complete analysis must also be repeated. Validation is suitable for application where transformation is likely to change, for example after errors are detected and corrected.

The techniques for verification are also problematic due to difficulty of application to large and complex model transformations [97, 118, 120, 38]. In [97] the technique only applies to simple transformations. Due to the complexity and size of model transformations, verification is applied to the abstract specification of the transformation. Simplifications are required as the analysis formalisms used in the techniques are unable to scale to large or complex software systems. Conversely, in validation, the executable artefacts are evaluated by the application of models [56, 27, 58]. Validation of transformation by applying input data can be applied to large and complex transformation, as the operational artefacts of the transformation are analysed.

In verification, a transformation definition is analysed for consistency against a specification [97, 118, 120, 38]. This requires interpretation of the statements used in the trans-

formation. By analysis, the statements that constitute a transformation are shown to either be consistent or not with regards to the specification. Verification analyses and interprets the statements of the transformation formalism, making each verification technique specific to the transformation formalism analysed. Similarly, validation techniques that inspect the inner workings of the transformation to create models are specific to the particular transformation formalism. Transformation validation applies specific models with an expected outcome, to the demonstrate transformation [56, 27, 58]. If the expected outcome is violated, the case that was applied to the transformation can be analysed using debugging support in the transformation framework. Validation techniques not considering the internal working of the transformation can create models for application to the dependability of a wide range of transformations. Validation can be performed on a diverse range of transformation formalisms, as the internal transformation definition is not analysed.

In this section, a discussion of techniques of validation and verification has been presented. Model transformation verification is problematic in existing methods as a deal of expertise and interaction in an analysis formalism is required. For validation, models may be created automatically by exploiting tools for software verification. Existing techniques are also problematic as conversions are used, but the techniques can not be apply to those conversions used in the technique. Verification of a given transformation must be repeated if the transformation changes, requiring a repeat of the interaction necessary in verification. Models generated for validation of a transformation can be simply re-applied to repeat the validation. Verification techniques are limited in terms of the size of transformations analysed, due to the software analysis formalisms used. Validation is done on executable artefacts of transformation, so validation scales to large and complex transformation. Validation can also ignore the transformation definition, so being independent of the model transformation formalism used. For the propose of this work, validation is selected as a basis for transformation dependability evaluation.

3.8 Outline of Properties for Proposed Validation Technique

Based on the discussion of existing techniques, the following properties inform the creation of a promising technique to support automated model transformation dependability evaluation.

- Automated- due to the number and complexity of model transformations and the diverse model transformation formalisms, the technique for model generation should be automated. By exploiting software verification formalisms and tools, models can be automatically generated from the input specification of transformations.
- Executable artefacts- the "operational" code of a transformation should be analysed so that a wide range of transformations can be evaluated. That is, the validation will be done by executing the transformation.
- Valid- where conversions are used in the technique, it should be possible to evaluate the dependability of those conversions by application of the technique (self-applicability).
- Adaptable- model transformations definitions can change when errors are detected. The technique should be easily re-applied to a model transformation if it the transformation is changed.
- Scalable- model transformations are large and complex software, the technique should apply to evaluate large model transformations.
- "Black-box"- the internal structure of a transformation should not be analysed as this allows for the evaluation of a wide-range of transformation, without regards to the formalism used.

3.9 Summary

This chapter has introduced the importance and difficulty of evaluating the dependability of transformations. By analysis of existing techniques, a classification is found. By review of

the existing techniques a unique selection of properties for are discovered for an automated model transformation dependability technique.

In summary, the contributions of this chapter are as follows:

- A comparative discussion and classification of existing techniques to perform transformation validation and verification.
- An outline of a particular set of desirable properties towards model transformation validation.

The next chapter will discuss a particular proposed technique for model transformation dependability evaluation, based on the discussion in this chapter. The created technique is evaluated with regards the above properties in chapter 6.

CHAPTER 4

A TECHNIQUE FOR THE AUTOMATIC CREATION OF MODEL GENERATORS TO ASSIST MODEL TRANSFORMATION VALIDATION

It makes no difference whether a work is naturalistic or abstract;
every visual expression follows the same fundamental laws.
— Hans Hofmann

4.1 Synopsis

This chapter proposes a practical, self-validating technique to address the problem of creating models for use in model transformation validation. The core of the method is the conversion of a given meta model to a software analysis formalism. The analysis formalism is normally used for the verification of software, here it is applied to generate models. Meta models specify the input of model transformation, and a generative representation can create valid instances of the specification. The technique presented in this work creates a model generator based on a meta model, to support validation of model transformation.

The technique uses model transformations as a central scheme for creating model generators. By using model transformation in all conversions, the technique can also be self-applied, to support validation of the transformations involved in the technique. That is, an implementation of the method can be used to automatically create models that self-validate the implementation and other model transformations, by testing.

This chapter does not reference any particular implementation, to present the concepts of the technique without the complex details of an implementation. The discussion is made on the complexity of model generation from meta models, SAT based analysis formalisms, the use of such analysis for model generation and the conversion of meta models to such analysis formalisms. An abstract discussion clarifies the difficulty of model generation and the essential features of the presented solution.

4.2 Meta Models: Complex, Non-Generative Input Specification

By the review of the previous chapter, set of features are proposed towards a technique for model generation. The proposal involves the automatic generation of models from meta models. The meta model is the basis from which the valid models in a modelling formalism are created. To manually develop test models for use in validation requires a great deal of interaction by an expert in the modelling formalism. Manually creating models validation

depletes valuable developer resources. Designing models for testing also places burden on the creator, as manually created models in such large and complex state-space may not uncover errors, only expending significant time. Furthermore, the number of models created for use in validation must fulfil the adequacy criteria of the testing [59], to form a significant sample for the validation. Instead, it is proposed that valid models are created automatically, based on the meta model, to aid in the validation of model transformation.

Many modelling formalism have been created and new modelling formalisms can be defined for repressing different software problem. Each modelling formalism is defined in the meta model of the formalism. Model transformations allow the models from one modelling formalism to be converted to another formalism. Model transformations have been described as a key concept in model driven software development and any combination of meta models can be used in a given model transformation. Each model transformation can convert between a distinct combination of meta models and meta models can develop change over time. Generating instances for validation must not be specific to any given meta model, as done previous model generation techniques [58], as only a few transformations may be validated. The generation of meta models must apply to different meta models, so a range of distinct model transformations may be validated.

Abstractly, a meta model are made up of two parts, structural elements and logical constraints on the structural elements. Structurally, meta models consist of the allowed elements (meta classes) and element properties (meta class attributes) and relationships (between meta elements) [73]. Logical sentences constrain the properties of the allowed elements, by application of the Object Constraint Language [121]. A meta model may consist of any number or combination of these logical constraints and structural elements to define a modelling formalism. A model is said to conform a given meta model when using only instances of meta elements, attributes and relationships of the meta model, and the structural elements are valid according to the logical constraints.

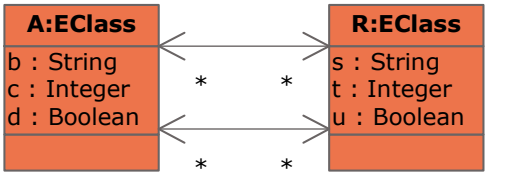
Meta models are beneficial to define expressive modelling formalisms, allowing for a wide range of complex but valid models in the modelling formalism. For this reason, meta

models are also problematic. Deriving valid models or specific models from meta models automatically, without expert intervention, is not a straight forward-task. This is in alignment with the intended purpose of modelling formalisms. Model developers select and instantiate elements from the meta model and within the constraints of the formalism, using model elements to describe the concepts of a system in a model.

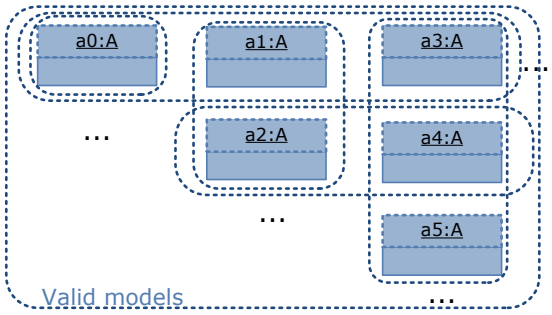
Automated, algorithmic creation of models from meta models is problematic for two opposing reasons. Firstly, the structural features of a meta model typically allow for an infinite number and combination of elements in models. In a simple meta model with only one meta element, there are an infinite number of conforming models. Considering it is possible to define a meta model with any number of meta elements, combinations of instance elements can be used to create valid models. Opposing the number and combination of possible models are the complex logical constraints and relationship cardinalities. Constraints and relationship cardinalities arbitrarily constrain which subset of the models are valid in the modelling formalism. Although models can be automatically created from the structural elements of a meta model, only few of those may conform to the constraints and relationship cardinalities of the modelling language. Furthermore, the constraints of a meta model can also over-constrain the meta model, meaning no instances are possible. Algorithmic creation of valid models that conform to the structure and constraints of a meta model is a difficult undertaking

To illustrate how a meta model defines an infinitely large number and combination of models, consider the example meta model defined in figure 4.1a. The number of valid model of figure 4.1a is infinite because any number or combination of the meta classes may be used to create a valid instance model, as shown in figure 4.1b. Further, in each of the valid instances any combination of the pairs of meta classes may have a relationship between them, as in figure 4.1c. Finally, in any of the valid instances may have any combination of values assigned to each labelled attribute, as in figure 4.1d.

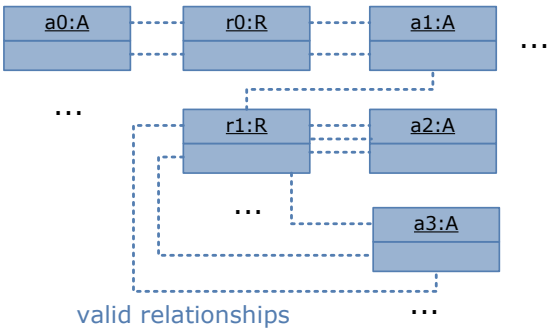
For a moment, not considering the complex constraints of a meta model, algorithms to generate all possible instances of a structural meta model, have been previously proposed.



(a) Example meta model.



(b) Combinations of valid models with multiple meta elements.



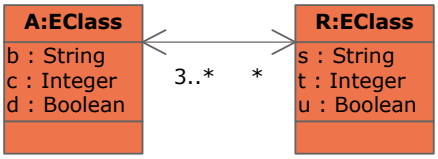
(c) Relationships between models with multiple meta elements.

(d) Attributes of meta-elements allow for more valid instance of the element.

Figure 4.1: Meta models define an infinite number of structural models.

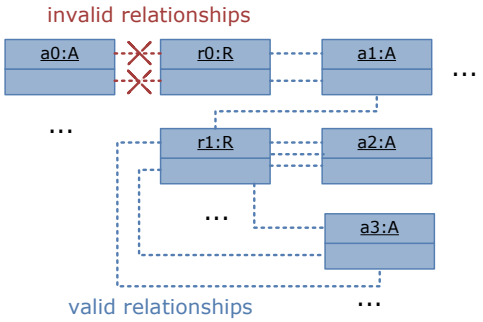
Most advance of these is the work of [56], proposing a graph grammar for representing meta models and algorithm for generating instances from the graph grammar. By the technique, a meta model is converted to a grammar upon which the algorithm acts to generate instances. The grammar created from a meta model must be terminal, so may not contain loops (as the meta model in figure 4.1a), as this mean the algorithm does not terminate. The grammar and algorithm of [56] does not consider the cardinality of relationships or logical constraints of the meta model, generating invalid models from the meta model.

To illustrate how constraints and relationship cardinalities complicate automated model generation, consider the updated meta model in figure 4.2a, based on figure 4.1a. Relationship cardinalities define the numeric range of the possible elements allowed between related elements in a valid model. Applying the sample cardinalities, only some of the structural models of the meta modelling formalism are now valid models, as shown in figure 4.2b. The logical constraints of the sample meta model further complicate which of the structural models are

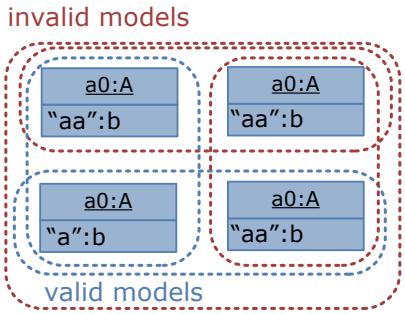



```
context A
inv: A.allInstances() ->
forAll(a:A | a <> self implies a.b <> self.b and
                               a <> self implies a.c <> self.c)
```

(a) Example meta model with relationship cardinality and constraints.



(b) Relationship cardinalities restrict the valid models.



(c) Logical constraints further restrict the valid models.

Figure 4.2: Meta models constraints arbitrarily restrict the valid models in a modelling formalism.

valid, as shown in figure4.2c. Logical constraints and relationship cardinalities arbitrarily restrict which of the possible models are valid models in a modelling formalism.

The logical constraints of a given meta model are specific to the meta model and the elements of the meta model. Relationship cardinality can be represented as logical constraints. In practice, the Object Constraint Language [114, 121] is used to specify meta model constraints. The Object Constraint Language is similar to first order logic with the addition of navigability of relationships, some collections, arithmetic operations and the concept of undefined values (null) [121]. The constraint shown in figure 4.2a is defined in the Object Constraint Language notation. Constraints are used to restrict the allowed models in meta models, to define complex modelling notions.

Typically, valid models of a modelling formalism are created by hand, using expert knowledge to create models for transformation validation. Creation of valid models of a modelling formalism by an algorithm is possible, but would result in invalid models and does not consider logical constraints, looped relationships or relationship cardinality [56]. It is difficult to

automate the generation models directly from meta model, for use in the validation of model transformation. According to only structure, there are an infinite number of valid models in a typical modelling formalism. Only a subset of the possible structural models are valid when considering the logical constraints and relationship cardinalities that define a modelling formalism.

Towards automated model transformation validation, a selected number of the valid models of a meta model are created and applied to the transformation. To generate models is problematic for model transformation validation, due to the structural and logical constraints in the meta model. Several software verification formalisms and tools are available to analyse software specifications. Such formalisms and tools are also applied to the analysis of software models [8] and model transformations in model driven software development. The formalisms and tools perform analysis, whilst considering the complex constraints.

4.3 Generation of Models from Meta Models - Architecture for Model Generation

4.3.1 Bounded "Model Finding" Software Verification Tools Exploiting SAT Solvers

Boolean satisfiability "SAT" and software abstraction formalism and tools exploit SAT for software verification. Satisfying any given large boolean formula is a difficult problem, and involves selecting variables which satisfy that formula [102]. Computationally difficult problems have been shown as equivalent to SAT [44]. Existing techniques propose and apply advances in boolean satisfiability solving for creating formalisms and tools for the analysis of complex software systems [141, 69, 125]. The analysis techniques typically have a notation and tools to allow automated reasoning, with in a bounds, about systems represented in the notation. This is beneficial as properties of the given system may be verified. The analysis techniques of [141, 69, 125] require the notation and tools be applied to the system analysed manually, using the expertise of a trained analyst.

The formulae SAT attempts to solve consist of terms (variables) and the logical connectives (operators) between the terms. An instance of a formula consists of an assignment of values to each term of the formula; a formula with values assigned to the terms may evaluate to true or false. The aim of SAT is to uncover a particular assignment of values in the formula which results in the formula evaluating to true. A selection of values that a boolean equation that evaluates to true is said to solve or to be a satisfying solution of that equation. All possible solutions, both satisfying and not, of a boolean formula becomes exponentially large as the number of terms increases. Only some of all of possible solutions may satisfy the formula. For a typical number of terms in the formula, the number of possible instances is too large for all cases of the formula to be considered in practice[1]. Instead, bounds are set within which analysis may be practical and the analysis is only valid within those bounds.

Software tools "SAT-solvers" (see [102]) are available to automate the process of attempting to solving a given boolean formula. There are several parallels between solving a SAT formula and model generation from a meta model. A defining feature of SAT-solvers is the ability solve boolean formulae, constraints, within a bounds. The solvers search for an instance of a given boolean formulae where the instance yields true. However, reasoning about numbers or strings, as commonly used in meta models is not possible.

SAT solving does not directly enable reasoning about software systems [102]. Instead, software abstractions [141, 69, 125] are used to encode software systems an exploit SAT-solvers to perform the analysis. The formalisms and tools of [141, 69, 125] are used to reason about the properties of the systems. In Alloy [69], a system may be modelled textually and constrained by a form of first order relational logic. In the related technique of the Kodkod [141] reasoning system, models and constraints are created using first order relational logic via a Java library. Kodkod also allows for the specification of partial instance models, upon which analysis may be based. Both Alloy and Kodkod support reasoning, within a bounds, about strings and integers. Software verification formalisms allow reasoning about an abstraction of software; indirectly using a SAT-solving tools [102] for the analysis.

[1]For example, a boolean formula with 50 terms has $2^{50} = 1.12589991{\times}10^{15}$ possible solutions.

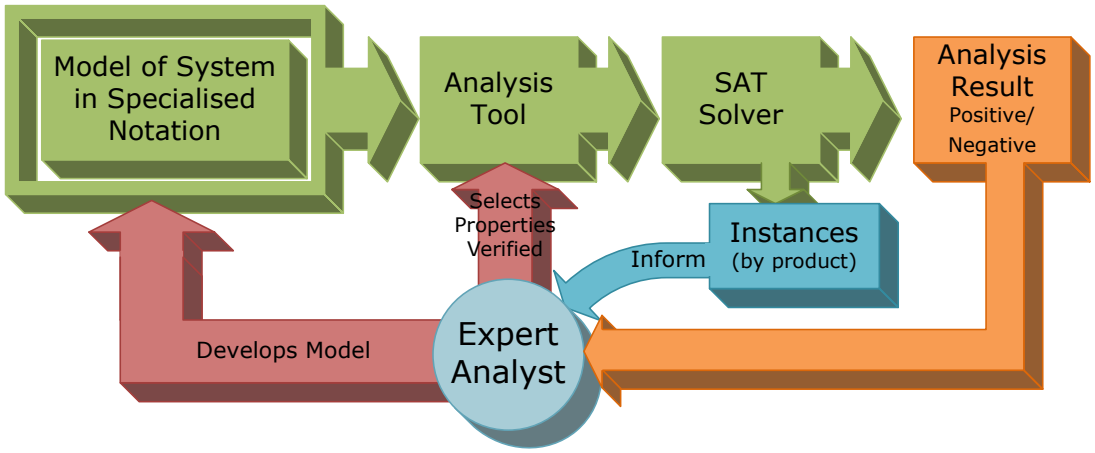


Figure 4.3: SAT based analysis of complex software abstractions.

The techniques of [141, 69] are referred to as "model finders" [69]. The techniques and analysis tools provided by those techniques can be exploited to generate models form a meta model specification.

4.3.2 Model Generation using SAT based "Model Finders"

The aim of model finding software verification techniques is not to generate instances from a specification. Instead certain properties of the system evaluated are specified and analysed. Automated SAT solving is used in the techniques to search for an instance of the specification that violates the specification. If a violation occurs, then the property is invalidated for the system. In a similar way, a property that must never hold may be specified, and is checked in all cases within the bounds. An overview of the process of software verification using SAT is shown in figure 4.3. A by-product of such verification are the instances satisfying the specification.

The current work proposes the generation models for application to models transformation validation. This is in contrast to software analysis tools, where all cases within the bounds are checked against the property analysed. As part of the model finder-based analysis, instances of the given software specification are created by exploiting a SAT solver. A meta model is a complex software artefact, and can be re-represented in the notation provided by the analysis techniques. However, instead of analysing the properties of the meta model, the

techniques can be exploited to create some valid instances of the meta model.

The models produced by a model finding software analysis technique are meaningless instances that conform to the specification. The analysis techniques allow the definition of additional "synthetic" constraints. The synthetically constrained abstraction of the system allows the reasoning and analysis to be directed by an expert [141, 69]. By placing additional constraints on the model, specific required models (as specified by the constraints) can be generated. Specific generated models when applied to the transformation validated, will show the range of scenarios for which the transformation is applicable. For example to cover the entire meta model (further discussed in section 4.5).

Model finding software analysis formalism that exploit SAT for software verification proves expressive software abstraction notations. In the proposed technique, the generated instances will be used for model transformation validation. A meta model must be converted to the notation of the analysis formalism. However manual conversion would require a great deal of expert intervention and is not ideal due to the number and susceptibility to change of meta models. Instead, and to generate models from any given meta model using the model finding formalism, it is proposed that the meta model is automatically converted.

This section has discussed model finding software analysis systems. Such systems are found to be ideal for the generation of models from structural meta models with constraints, as the analysis takes the constraints into account. Generated models can be applied to the validation of model transformations. The conversion of meta models to an analysis formalism is necessary create model generators.

4.3.3 Meta Model Re-Representation towards Model Generation

As previously explored in chapter 2, models are defined by a multi-level modelling hierarchy [73]. Meta models are also models, as such, are defined by a meta model, known as a "meta meta model", used to define meta models. Meta meta models are self-defining and terminate the modelling hierarchy. That is, a meta meta model conforms to only the elements defined by it and exists as a terminal of the model definition hierarchy. To convert a meta

models, a model transformation must be defined at the meta meta level. In the multi-level modelling hierarchy, model transformations can be defined at the meta meta level, to convert meta models.

Model transformations convert models between different modelling formalisms. To convert meta models to a generative form, a meta model transformation is required. Meta model transformation is used in the proposed technique, to convert meta models to a corresponding analysis representation. Meta model transformation is not a novel concept, essentially similar to model transformations. The input of a meta model transformation is a meta model instead of a model. An overview of meta model conversion to a model finding formalism is given in figure 4.4. By utilising a meta model transformations, the technique can benefit from the automation afforded by model transformation, to convert a given meta model.

Meta models may be composed of several smaller meta models and reference elements of other meta models. Towards this goal, meta models can define a name space for the current elements and can reference elements of separate meta models' name spaces. In effect this allows for reuse of previously created meta models. For the purpose of re-representing such meta model referenced elements from external meta models must also be converted to the generative form. This involves treating the meta model and each referenced meta models as a single name space ("flattening") to a single meta model before conversion. This is also relevant for model transformations where multiple meta models are used as the source meta model.

Several meta meta models are available, such as the Meta Object Facility "MOF" [113] (and variants Essential-MOF "EMOF", Complete-MOF "CMOF") standards, the Eclipse Modelling Framework Core "ECore" [137] and the Kernel Meta Meta Model "KM3" [71]. Meta meta models are intentionally expressive enough to define a wide range of modelling languages. For example, MOF is used to define the UML [110] family of modelling languages, including both behavioural and structural formalisms. Using MOF, KM3 and ECore, it is also possible to create meta models for other, non-modelling formalisms. In this way, models may be converted to non-modelling formalisms. For example the Java meta model defined in ECore [137]

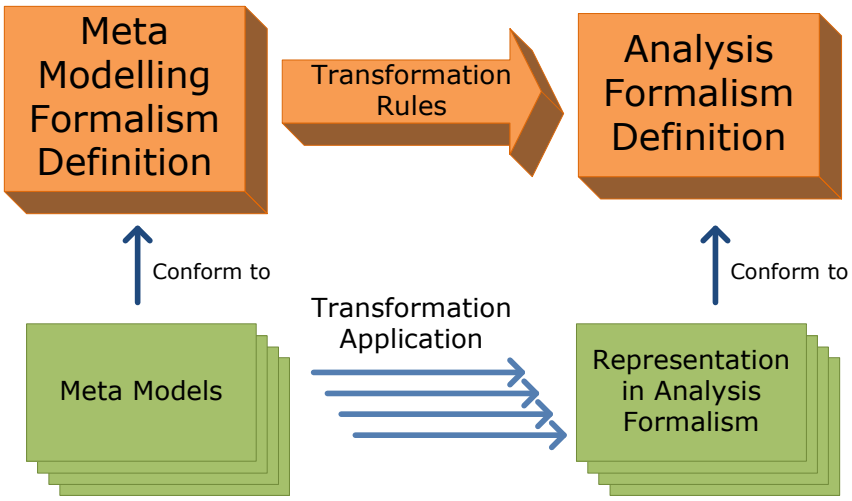


Figure 4.4: Overview of meta model transformation : conversion of meta models to model finding formalism.

or the OCL meta model defined in MOF [122]. Meta meta model are used to both define the concepts of and by tools to allow modelling of diverse formalisms.

A defining feature of meta meta model is the ability of each meta meta model to create a representation of itself [137, 71, 113]. Meta meta models are of equivalent expressiveness, it is possible to represent and one meta meta model using another. For example, KM3 may be used to re-represent ECore or MOF, as presented in [72, 71, 64], and vice versa. It is also possible to automatically convert the meta models in one meta model formalism to another, by a meta model transformation. Thus, the selection of a particular meta meta model formalism is minimally consequential and left as an implementation issue.

Software analysis formalisms are defined by textual languages, not using modelling notations. Textual formalism are not naturally viewed as having meta models or models, as required by model transformation. However, it is possible to create a pseudo-modelling hierarchy for converting between modelling and textual notations, as shown in figure 4.5 and figure 4.4. Model finding techniques have an instance level, and instances are produced by via tools that exploit a SAT solver to analyse the system. In the current technique, the model finding instance level is the equivalent of a model and the abstraction of the system is equivalent to a meta model. The string grammar of the abstraction formalism is equivalently the meta meta model. Thus, an equivalence can be made between the modelling hierarchy and

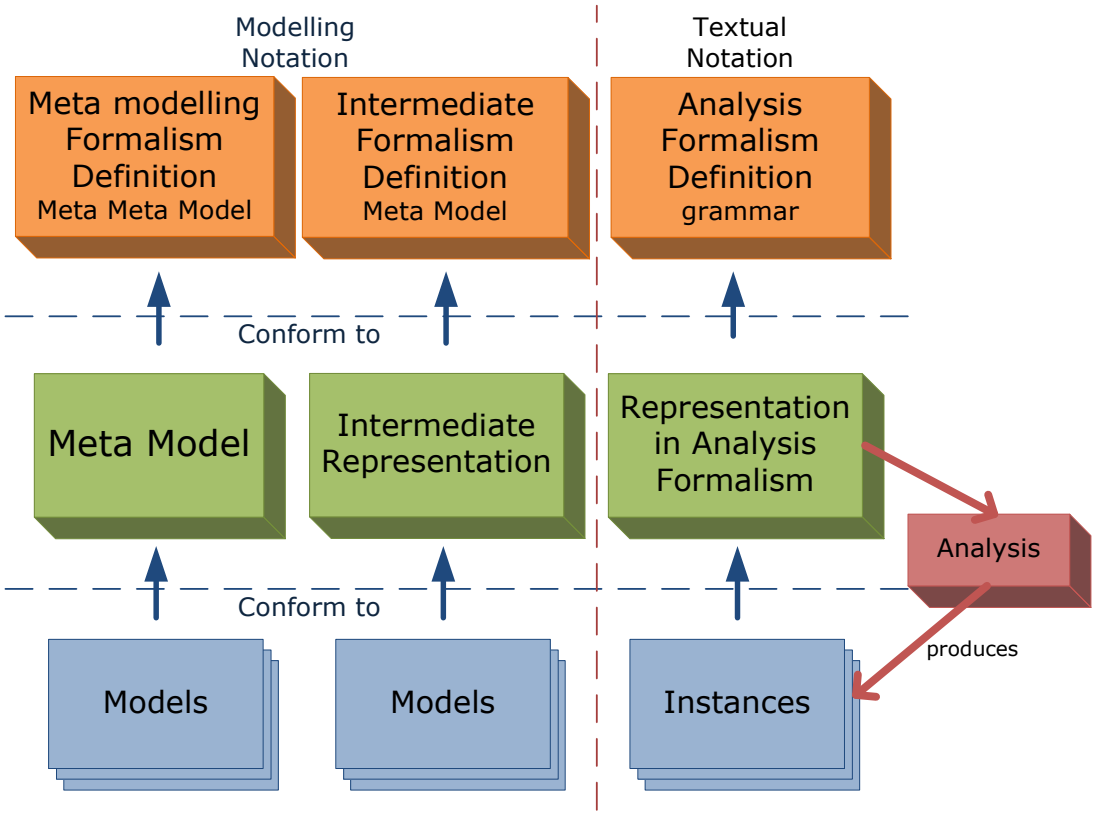


Figure 4.5: Modelling hierarchy : models, meta models and meta meta models.

the language hierarchy of model finding techniques.

The textual analysis formalisms are fundamentally different to modelling techniques. It may be possible to interpret the formalism hierarchy, but it is still necessary to re-represent meta models in the model finding formalism. It is proposed that the conversion of meta models is done by a model transformation, converting a meta model to the abstraction formalism. Model finding analysis formalisms are text based, defined by a string grammar. The transformation will require some form of model to text transformation, as a bridge between the modelling and text based formalisms. By model to text transformation, the analysis of a meta model can done in a model finding formalism.

The form of the meta model will naturally change, as there are differences between meta modelling formalism and the model finding analysis notation. The instances of the model finding formalism will not directly apply to the original meta model. Model finding abstraction formalisms do not have the same structural elements of a modelling language. By a conversion of a meta model to a model finding form, the instances produced from the model

finding form are not meta model instances and the instances must be converted before they can be used as an instance of the original meta model.

The conversion of model finding generated instances is problematic. An instance produced in the model finding formalism is based on the abstraction, not the original meta model upon which the abstraction is based. The model finding representation uses the elements of abstraction to generate instances. Those elements of the modelling finding notation are in a different arrangement the elements of the original meta model. It follows that instances generated by the model-finder notation have a different arrangement than elements than a corresponding meta-model model. To create the corresponding meta model instance from a generated instance requires information on how the elements of the meta model and corresponding elements of the model finding notation are related. Once the correspondence is known, the elements of the generated instance can be used to create elements in a meta model instance.

Because the original meta model and corresponding model finding representations are different, the generated instances are different to the corresponding meta model instances. So it becomes necessary to convert instances to the meta model form as required by the validation technique. The next section elaborates the automated conversion of generated instances to meta model form, using the trace of the meta model transformation.

4.3.4 Meta Transformation of Traces to Automate to Conversion of Generate Instances

To convert generated instances, it is necessary to determine the relationship between elements of the original meta model and the representation of the meta model in the analysis formalism. Once the relationship is determined, elements of the model finding formalism can be converted to models in the meta model formalism. To automate this process requires that a record is made of exactly how each meta model element was used to create each model-finding element in the analysis notation, for a given meta model conversion. The record of the relationship of source and destination elements in a model transformation is known as

the trace of the model transformation [47].

In general, model transformations define a relationship of how two meta models are related. The trace of a model transformation is specific to the model transformation and the input meta model and the created model-finding form. The trace of the transformation is a record of elements converted. In effect, the trace of the meta model transformation is the specification of how meta model elements are related to the corresponding model finding representation.

To automate the conversion of generated instances to meta-model models, it is proposed that a model transformation is created. The trace of the conversion of a given meta model transformation can be used to create the instance transformation. As the trace is specific to the meta model used as input to the meta model transformation, so the created transformation is specific to the original meta model. The model transformation created from trace is used to convert the instances generated in the model finding notation to meta model instances. The model transformations created to convert generated instances is at a different level than the original meta model transformation. Section 4.3.5 discusses the transformation of generated instances.

Traceability of model transformations has previously been studied for aiding the development and supporting chains of model transformations [115, 70, 47]. The trace of a model transformation can also be recorded as a model [70, 47]. Trace meta models are used during transformation, by the transformation framework, to record the trace of a transformation as a trace model. To convert a transformation trace to an instance model transformation can be done by a higher order model transformation [147, 47, 140]. Higher order transformations have various uses particularly where in model transformation tool support and measurement of model transformation [147, 47, 140]. In the higher-order transformation of traces, the trace model is treated a specification for a model transformation; the trace is interpreted to create a model transformation for instances. Figure 4.6 gives an overview of the conversion of trace to model transformation.

The trace is treated as the specification of how instances are converted from the model

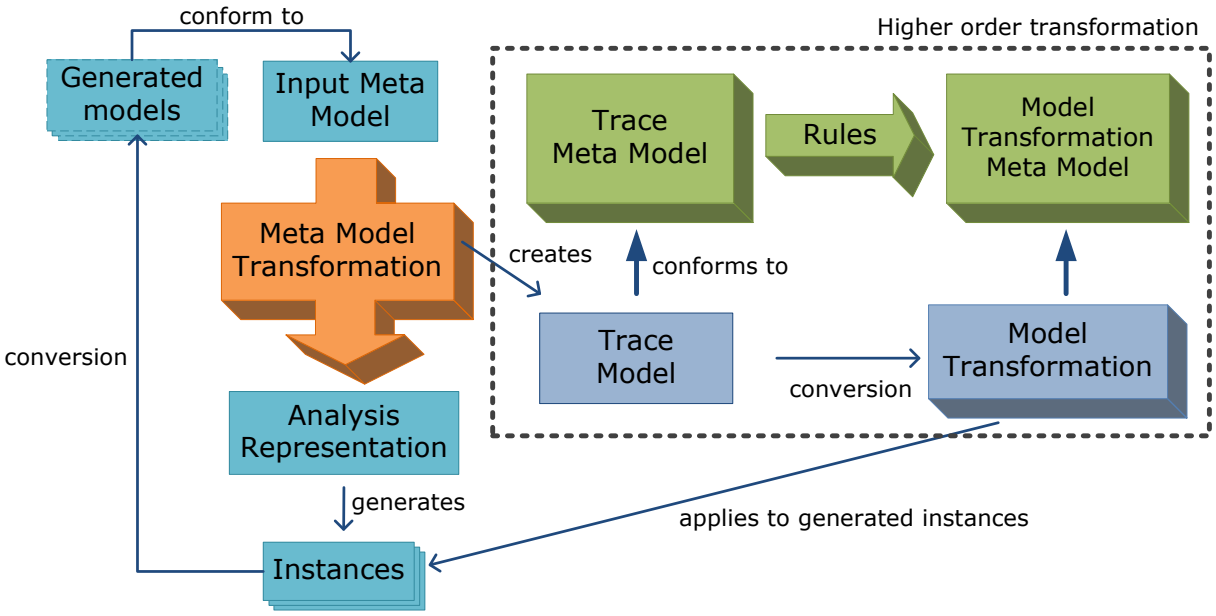


Figure 4.6: Higher order transformation to convert trace to model transformation.

finding formalism to the meta model formalism. The conversion of instances is required as model instances in the corresponding model finding representation conform to the model finding abstraction and the abstraction uses a different arrangement of elements than the original meta model.

4.3.5 Transformation of Generated Instances

The conversion of a meta model to model finding abstraction is unique for each unique meta model. The trace of that transformation is therefore specific to each unique meta model used. A transformation created from a given trace is therefore also unique to the meta model used in that traced conversion. That is, the trace is converted to an instance model transformation only once per unique meta model. The created instance converter also only applies between instances the unique meta model and the corresponding model finding representation. So the instance transformation is created once per unique meta model transformation and only applies to the meta model used.

In the generated model transformation, the created converter acts at a different meta level and direction than the original meta model conversion. The meta model transformations de-

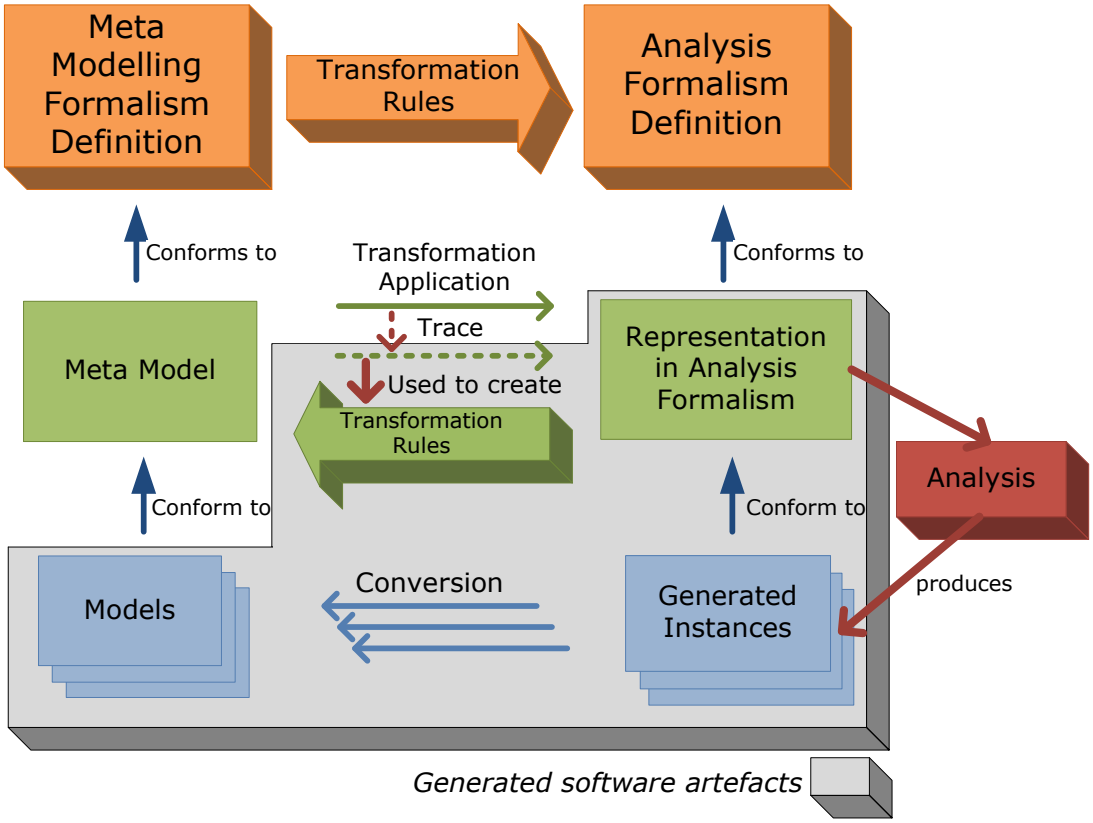


Figure 4.7: Conversion of generated instances to models of the original meta model.

fines the conversion of meta models, to a corresponding model finding representation. The instance model converter defines conversion of instances from a specific model finding abstraction to the original meta model. The instance model converter can be applied to convert instances created by the model finding tools, in the reverse direction; to create instances of the meta model, as shown in figure 4.7. The derived instance transformation is therefore a reverse model transformation, based on the original meta model transformation.

In this section a discussion was made on the features of the generated instance conversion. The instance transformation is derived from the trace and is specific to the given trace, and meta model used. The instance converter needs only to be created once per unique meta model conversion. The previous sections have presented how a given meta model is converted into a generative, model finding representation. The generated instances are then converted to models of the given meta model. Such model can be applied as test cases for model transformations which use the given meta model. Thus, instance models can be generated

automatically, using meta models.

The three transformations proposed are the meta model transformation, the higher-order trace-transformation and the generated instance model transformation. The trace and instance transformations follow from the original transformation. The technique presented in this work forms a chain of transformation, a conversion from meta model to model finding representation, the conversion of the trace of that transformation to a generated instance converter. One issue not so-far considered is the dependability of the model transformations used. That is, if any of those transformations involved contains errors then the results of the technique, the generated models, are also invalidated.

4.3.6 Evaluating the Dependability of the Model Transformations used in the Technique to Create Model Generators

To be applied in practice, the technique to create model generators must be implemented. As with any developed model transformation, the transformations created for and by the implementation are prone to errors. If the transformations in the implementation are erroneously defined, the created model generators can produce invalid models. Before the technique is applied to creating model generators, the dependability of the model transformation used in an implementation must be evaluated.

A man-made conversion of meta models to a model finding notation, as with any software can contain errors. For example, the produced model finding representation of the meta model artefact may not be an accurate or complete representation of the original meta model. An erroneous resulting representation in the model fining notation may be used to generate instances but the instances would not correspond to the original meta model. So any errors in the conversion invalidate the results of analysis - instance generation - in the model finding representation.

The conversion of trace to transformation must produce a model transformation to convert generated instances to meta model instances. The conversion of trace to model transformation is manually developed, The conversion of trace to model transformation is also

developed manually and so also prone to error. The transformations produced by the trace converter may also be erroneous. The trace conversion produces a transformation would also reflect any errors in the trace, for example if the trace is incorrectly recorded. Errors in the transformation to convert instance means the generated instances can be incorrectly converted.

In review of advanced model transformation dependability evaluation, it is found that model transformation verification is not practical. In several techniques, it is necessary that a simplified specification is used in the evaluation transformations, with finite transformation and without inheritance or complex relationships in meta models [97]. This is an issue in the current as the meta models used in the technique presented in this work are not finite and use complex inheritance and relationships. In yet other techniques, only pairs input and output model are used to validate a specific application of the transformation [38]. This technique proposed that each execution of the transformation is verified, not the transformation in general. So existing verification techniques do not apply to evaluate the dependability of the transformations used in the current technique.

In other advance model transformation verification techniques [118], it is necessary to set the property proved and guide the verification of the transformation, using proof finding tools. This is problematic for the model transformation is created from the trace, each time a meta model is applied. The created transformation is used for the conversion of instances generated by the model finder. The instance transformation is created once for each unique meta model. The verification of the instance transformation have to be repeated for every meta model used, which is not practical. The verification techniques from literature have issues that preclude them from verifying an implementation of the technique presented in this work.

Instead, it is proposed to exploit two properties of the presented technique. First, the technique uses model transformation. Secondly, the model transformation uses a meta meta model to specify the input. This allows the application of the presented technique to itself. For self-validation, the self defining meta meta model can be applied as the input to and

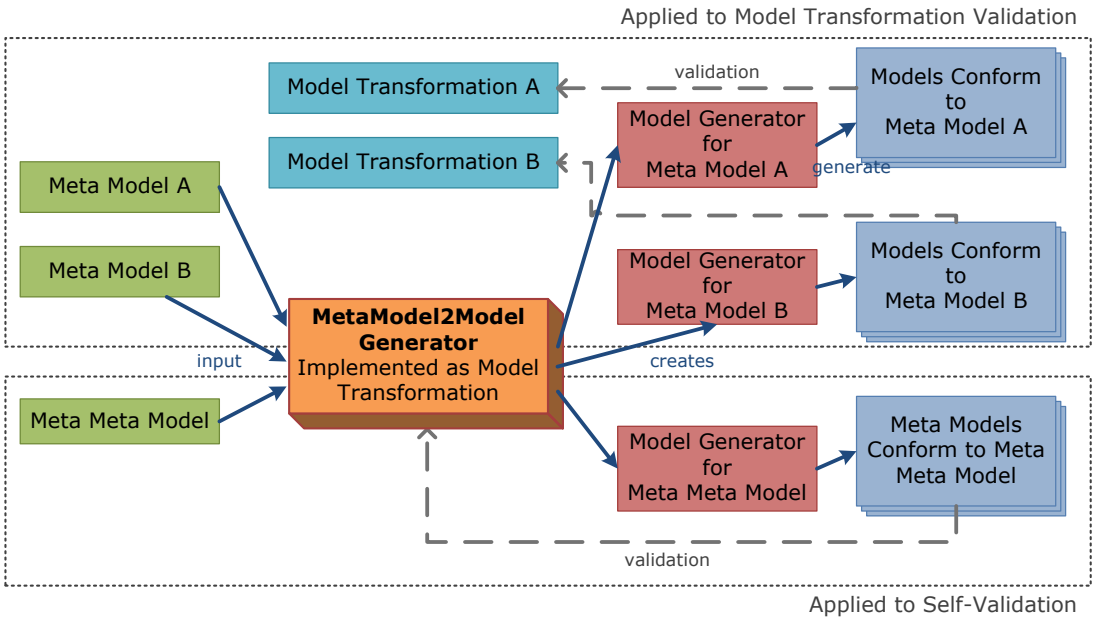


Figure 4.8: Self validation by generating meta models.

implementation of the current technique. In this case, the implementation would create a meta model generator. The meta model generator consists of a model finding representation of the meta meta model and a model transformation, to convert the generated meta models. Generated meta models can then be applied as the input to the original implementation transformation, for self-validation. The self validation property is illustrated in figure 4.8. In this application of the technique, the generated meta models apply as input to the validation of the implementation.

Because model transformations are generated by the technique manual verification of those created transformations is infeasible. Furthermore, some verification techniques would only apply to particular applications of the transformations involved. Instead, it is proposed that the dependability is evaluated by applying the proposed technique to generate meta models. By generating meta models that apply to the validation of the technique, an implementation of the technique can be used to self-validate.

4.4 Model Transformation Validation using Generated Instances

4.4.1 Automating Application of the Technique

Using the proposed method, it is possible for to produce models to validate a model transformation. As the meta model is considered the specification of the input, then testing using models derived from the meta model, without considering the code of the transformation is considered "black box" [34] validation. The instance models produced by the technique may be meaning-less, but are within the specification of the meta model. That is, models will be produced not intentionally represent meaningful models of a system, but will be useful for validation of a model transformation.

A model generator created by the technique is applicable such that multiple test cases are automatically generated and repeatedly applied for validation of a transformation. The technique can be applied to create a model generator for the input meta model of a given model transformation. The generated test models may be applied manually to validate the transformation. The application of a set of manually created models to model transformation is discussed in [136]. A test driver or test harness [104] is a tool that given a set of input data, will automatically apply each one in turn to the system validated. In the current technique, test driver for a given transformation can take models generated by the model generator and apply the generated models to a transformation, automatically. Figure 4.9 gives an overview of how the created model generators may be automatically and repeatedly applied to validate a given model transformation. A model generator for an entire meta model can produce a sequence models, so the validation can be repeated indefinitely, applying the sequence of instances to the model transformation for validation.

An optional method for management and direction of validation is also possible by exploiting expert knowledge in the technique. The meta model may optionally be enhanced by an expert to include synthetic constraints. Synthetic constraints specify those models that may be interesting for validation of a given transformation. As the constraints are specified

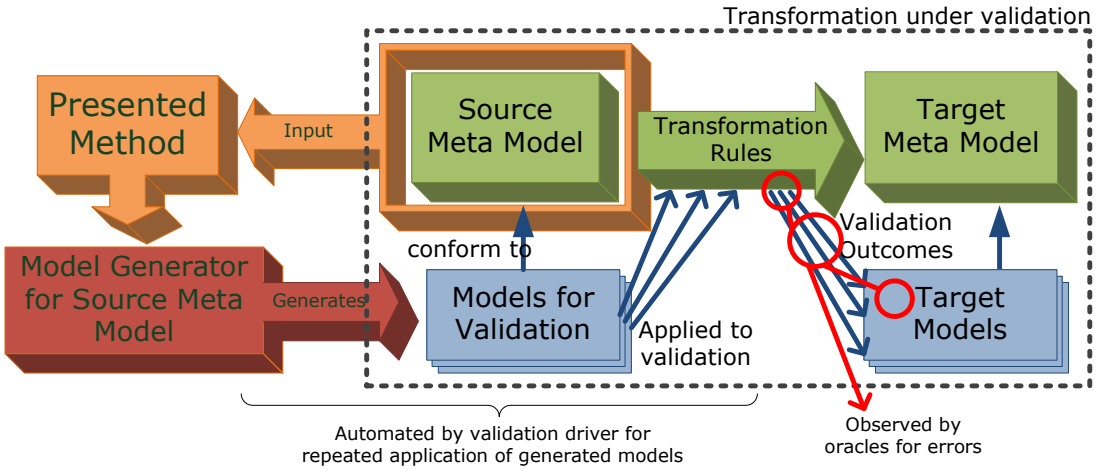


Figure 4.9: Validation using model generators created by the technique presented in this work.

by expert interaction, the validation by this method can be either using knowledge of the internal structure and algorithms of a transformation ("white box") or not ("black box"). Using this additional mechanism, an expert can apply testing techniques from the literature to develop test models to maximise error detection in validation [104, 34].

Model generators can be created for specific validation aims, by defining optional synthetic constraints on the meta model. The model generators will then create test data that conform to the specified synthetic constraints. The specification of the optional contains to guide the model generation must be done with strategy of the validation in mind. There are several strategies and techniques for designing test data [34, 62, 104], which data is created for validation depends both on the software validated and the aims of the validation. The selection of validation strategy for systematic model transformation validation using generated models requires careful consideration from the validation expert [148]. In practice, the validation expert can specify the properties models required for use in validation.

As well as "black box" validation, the presented model generation technique may also be used to perform "white box" validation [104]. "White box" validation uses information of the internal structure and algorithms of the transformation. For example, the effective meta model can be established by inspecting the model transformation use of the meta model. The effective meta model is a subset of elements from the meta model such that only the elements used in transformation are present. The concept of and automated techniques for finding the

effective meta model have been established in the literature. However, as this is a "white box" technique, inspecting the code of the transformation, it should done in conjunction with "black box" validation. "Black box" validation does not consider the internal structure of the transformation and may detect distinct errors than "white box" validation. Both "white box" and "black box" validation may be applied for validation of a given transformation.

The presented validation technique may be used to validate model transformation by exploiting pre- and post- conditions of the model transformation. Such a method has previously been applied in model generation for model transformation validation [58]. Logical conditions assert expected properties of the result (post-condition) when a specified property exists in the input (pre-condition). In validation, models may automatically be generated conforming to the pre-condition of the transformation. The development of pre- and post- condition annotations is a prerequisite for this type of validation.

For a given meta model, a model may be generated by the technique that causes an error in the transformation. As well as detecting errors, the cause of errors in the model transformation must also be sought. This is so that the error may be identified and rectified. Detecting the cause of a given error in any software may not be an automated process. Several process may be used towards detection of the cause of an error. Ideally, debugging should be done without modification to transformation under test. Several tools that support model driven software development support debugging the transformation by inspecting the state of internal variables [7, 145]. Information from the debugger may then be interpreted to uncover the cause of the error found by a generated model.

When a generated model has caused an error, the transformation may change. Depending on the result of debugging the change may be necessary in the model transformation or in the meta model. Where only the model transformation is changed in rectifying the error, the originally used model generator still applies to create models for use in validation, as the meta model is unchanged. This feature is useful in regression testing as the model generator, when re-applied will produce models using the same specification. Re-using the same model generator in validation demonstrates that the error is corrected and that previously working

test models still apply to the transformation. A change in the meta model requires only the reapplication of the technique presented in this work, creating a new model generator for the changed meta model.

In this section a discussion was made of the application of the previously presented model generation technique to the validation of model transformation. It is possible to use the technique to create a model generator to that can automatically generate the instances of a given meta model, repeatedly. The meta model may optionally be restricted by the effective meta model or by synthetic constraints, to direct the creation of models towards finding errors. When errors are detected in a transformation, a debugger may be used to inspect the inner workings of the transformation. An issue so far not discussed is how to determine success or otherwise of the large number and variety of generated models applied to transformation.

4.4.2 Oracles to Support Evaluating Results of Validation

When a model is applied to transformation for validation, it is necessary to evaluate the success or otherwise of the validation. A pre-set expectation of validation is known as the oracle of the validation [34], and can be set manually. The technique presented in this work allows for a large number of models and a wide variety of models for validation of a model transformation. Setting the oracle is particularly important in the current technique as there may be a large number of models and wide variety of models generated. The oracles of the validation must be set in advance, before application of generated test cases.

The models automatically generated by the technique are meaning-less, without any pre-determined form besides meta model conformance. To predict the outcome of the validation using generated models would require that the generated model be interpreted to accurately guess the outcome of the model transformation. That is, without applying the generated model to the transformation, predict the expected outcome. This is a known difficult undertaking in computer science [142], and if possible in general, would nullify the need for validation. Instead, setting specific oracles can only be done manually.

Model property preservation may be used as a basis for manually defining the oracles for

Oracle Name	Description	Indicated Transformation Error
Production of target model	A target model must be created when a generated model is applied to model transformation validation	Unexpected termination; Infinite execution
Relative size of target model	The target model created in transformation must be of a similar or relative size to the generated model used in the validation.	Errors/omission in translation
Meta-model conformance of target model	The target model created must conform to the target meta modelling formalism when a generated model is applied to the validated transformation.	Errors/omission in translation; Unexpected termination
By-product inspection	The messages produced by a transformation can be inspected for indication that an error occurred in the transformation.	Errors/omission in translation; Unexpected termination
Non-termination	The transformation may not terminate when a generated model is applied to validation.	Infinite execution.
Non-confluence	When a unique generated model is repeatedly applied to the validation of a transformation, the same unique target model must be created in each application of the generated model.	Errors/omission in translation; Unexpected termination.

Table 4.1: Six generic oracles for validation of model transformation.

judging the success of validation. In this case, certain known properties of the source model are expected to be preserved by the model transformation, as proposed in [41]. Such a property may be very specific to the transformation and models involved. For example, the property that must be preserved could be the name of an element (from [41]). In such techniques, models with the outcome of each model transformation is checked for preservation of that property.

It is noted that property preservation oracles apply to only specific kinds of transformations, where the aim is to preserve properties and the destination language is some-what similar to the source [41, 38]. It is also noted that the checking of such properties may not be possible, in general in certain destination languages [38]. Model properties may be used as manually defined oracles, but only for specific kinds of model transformation.

As the validation technique may generate a large number and wide variety of models, general oracles must be set and can automatically judge the success or otherwise of validation.

General oracles are applicable to any given transformation and where generated models are applied to validation. The model or model transformation properties do not need to be analysed or inspected in advance of validation. Generic oracles are particularly relevant in the current technique as meaningless models are generated and the outcome of the validation can be automatically determined by such oracles. Six generic oracles are proposed for evaluating the outcome of model transformation validation. Table 4.1 outlines the proposed general oracles for supporting model transformation validation, using the proposed technique.

A generic oracle for validation is the presence of an output model. This intuitive oracle exploits the property that a model transformation should produce an output model, when applied to an input model. In the technique presented in this work, when a generated model is applied to a model transformation under test, where no output model is produced may indicate that there is an error in the transformation.

A general oracle is the size of the model produced by validation, relative to the test model applied to validation. When an output model is produced from a large input model, a property of the transformation validated may be the preservation of elements of the model. In such transformations, the size of the model created by transformation is expected to be approximately corresponding size relative to the model used for validation. The presented model generation technique may be used to produce a large number of model. The size-preservation oracle is useful to determine the success of model transformation validation, by measuring the size of output models. Model measurement techniques are available for this purpose [98]. The size of models produced by validation of a transformation can be used to automatically evaluate the success of a transformation.

An oracle that may be applied to any given transformation is the conformance of the output of the transformation validated with the output meta model. This oracle exploits the property that, given a valid model, the transformation should produce an output models that conform to the output meta model. Several techniques from the literature, meta model conformance is presented as a general model transformation oracle [61, 27]. In the technique presented in this work, this is significant as the models automatically produced for validation

may be applied to the transformation and the output of transformation automatically checked for conformance to the output meta model [1].

Modelling and transformation frameworks do not normally allow for the creation or storage of invalid models. Models created by a transformation may not be a valid instance in the destination modelling formalism due to errors in the transformation. Any model created by model transformation must be a valid instance of the meta model of model transformation. The output model created may only use meta elements, from the destination modelling formalism. The created model must also conform to the constraints of the destination meta model. During transformation, conformance to the destination meta model is relaxed so models can be created incrementally by the transformation rules. An erroneous transformation can create models that are not valid instances of the destination meta model, for example if the transformation unexpectedly halts before execution is incomplete due to an error. When generated model are applied to validation of a transformation the conformance of the created models to the meta model can be used as an oracle for validation.

A general oracle for the validation of model transformations is the inspection of by-products of the model transformation, when applied to generated models. During model transformation, errors may occur but the transformation framework is able to continue and producing some output. For example, the transformation may report that a certain element could not be converted. The transformation framework can also record the process of transformation in a log, for example in a file or terminal output. The trace of the transformation still further records how the elements of a model are related to the elements of the model created by transformation. The trace and log, when observed, can be used as oracles in transformation. For example, if a critical element of the transformation is not present in the trace or if errors are reported in the log, the transformation may have errors. Analysis of by-products of the transformation when applied during model transformation are oracles to evaluate the outcome of validation.

The termination of the model transformation is a further oracle to evaluate the success

[1] Model to meta model conformance can be checked using a tools from the modelling framework, for example, using the ECore validation framework as described in section 5.7.

of the validation. The termination can not be determined in general by inspection of a program definition [142]. However, a generated model may cause an erroneous transformation to infinitely execute, due to an error in the transformation definition. In [88], termination is presented as a criterion for white-box model transformation validation using manually created models. In the current model generation technique, models may be produced that cause an infinite execution of the transformation under validation. The complexity of a transformation, the expectation of the performance for a given input, may be used to approximately calculate the amount of time taken for a given input. By setting a pre-determined time limit for the completion of the transformation of a model, the violation of the time constraint is an oracle to evaluate the outcome validation.

A given model transformation should produce a single unique output model, for each unique input model. This property is known as transformation confluence [88]. That is, when a model transformation is applied to the same input more than once, it should produce the same output in each application. In [88], confluence is presented as a criterion for white-box model transformation validation when using manually created models. In the current model generation technique, this oracle may be used to apply generated models more than once and inspect that the outcome is the same in each case. Optionally, in subsequent applications of the same model, the model may be non-critically modified. For example, re-arranging the order in which elements appear, where order of elements in the modelling language is unimportant. The of outcome of applying a unique model repeatedly to model transformation must be the same. In the current technique, a given unique generated model, in each repeated application to the transformation must result in the same model. Thus, confluence is a generally applicable oracle.

In summary, there are six oracles to evaluate the outcome of validation using the presented model generation technique. Oracles of model transformation validation can be set manually, with regards to pre- and post- conditions or by property preservation between input and output models. Generic oracles are those that require minimal expert intervention. This section presented six of the available generic oracles that can potentially be automated in

an implementation; including confluence, termination, transformation-by-products, presence of output, conformance of output to meta-model and relative size of output. Manual and automated oracles are used evaluate the outcome of validation when using models generated by the technique.

4.5 Integrating Coverage for Generating Models

Creating valid models from a meta model automatically is one part of the more general approach of validating correctness of model transformations. Another aspect of validation related to model creation is selecting and setting the properties of the models used in validation. For validation, coverage is used to determine which models are essential when carrying out the validation.

There are two main aspects of coverage that can be considered. In the literature, coverage for model transformation is categorised as input meta model coverage [61] or code coverage [98]. Input meta model coverage from [61] is adapts earlier work on coverage of UML design models [12] to the coverage of meta models. In contrast, code coverage [98] inspects the paths of the transformation under validation. The two approaches are completely disparate: high meta model coverage does not imply high code (and vice-versa) [98]. Because meta model coverage [61] considers only the input, it is purely "black-box"; code coverage is inherently "white box" as it inspects the code paths of the transformation when creating data.

The current work does not directly deal with the meta model coverage of the generated models. However, it is possible to generate models that conform to meta model coverage criteria, with some expert interaction. Meta model coverage criteria (e.g. "all class coverage" [12, 61]) can be achieved by adding an appropriate constraint to the meta model (that each class must be instantiated at least once) before creating the model generator from that meta model. The created generator would then produce models that conform to the coverage criteria specified by the added constraints. A new model generator would be required for each meta model coverage criteria. Furthermore, the validation expert must select the appro-

priate coverage measures suitable for the transformation under validation. Further research is required on how more complex coverage criteria can be represented using OCL and how this process could be automated. Further study is also required on the effectiveness of code and meta model coverage with regards to error detection ability using the current approach.

4.6 Summary

This chapter has presented the use of bounded model finding formalisms and tools for the generation of models. The generated models may be used to support the validation of model transformations.

In the technique presented in this work, meta models are treated as the input specification. To generate models, a meta model must be converted to a corresponding representation in a model finding analysis formalism. The conversion is done automatically by meta model transformation. Due to natural difference between the formalisms, it is necessary to convert the instances generated by the analysis formalism to the modelling formalism. An instance-converting model transformation is created from the trace of the meta model conversion to the analysis formalism. The created transformation converts instances into models of the original meta model.

"Black box" validation– where validation is done without consideration of the internal structure or algorithms of the transformation– is partly automated by the technique. The internal structure and expert knowledge may optionally be applied to specify desirable properties– for coverage– of the models generated for validation. In either case, generated models can be automatically and continuously re-applied. Six oracles are available to determine the success or otherwise of generated models when applied to transformation validation. Once errors are found, transformation development environments may be used to locate the cause of the error.

The proposed technique employs meta model transformation, model transformation tracing, chains of transformations and higher-order transformation to automate creation of a

model generator. The technique may also be used to generate meta models. That is, an implementation of the proposed technique can be applied to validate implementations of itself and other model transformations.

In summary, the contributions of this chapter are as follows:

- A consideration of how meta models may be automatically re-represented by model transformation in a model finding formalism; as a specification used to synthesize instances. A trace of the meta model transformation can be used to create a model transformation, to convert generated instances to the required form automatically.
- The implementation of the technique to can be applied to generate meta models to assist the validation of that implementation.
- The potential application of the proposed model generation towards assisting model transformation validation. A review how oracles can be used to assist validation using technique.

In the following chapter an implementation of the technique to create model generators from a meta model is presented.

CHAPTER 5

IMPLEMENTING SELF-VALIDATING AND REPEATABLE CREATION OF MODEL GENERATORS FROM META MODELS

I have been impressed with the urgency of doing.
Knowing is not enough; we must apply.
Being willing is not enough; we must do.
Leonardo da Vinci

5.1 Synopsis

This chapter describes proposed technique to create model generators from meta models, as a proof-of-concept implementation. The presented implementation is created using a combination of existing tools. The implementation exploits the Eclipse Modelling Framework, the Java-based Simple Transformer framework, the Apache Velocity template engine and Kodkod SAT-based model finding library also in the Java programming language.

Given a meta model, the presented implementation produces two artefacts, a Kodkod instance generator and an instance converter, both specific to the original meta model. Together, the created artefacts form a model generator which creates models of the original meta model.

The implementation utilises advanced model transformation concepts including meta model transformation, higher order transformation, chains of transformation, tracing; also utilising model-to-model and model-to-text transformations. The complete Object Constraint Language is not converted to Kodkod, instead tabl 5.2 outlines the conversion of a subset of OCL. Using the model transformations a Kodkod instance generator and a model transformation to convert instances is created from an ECore meta model. The presentation describes the key technologies and alternatives for future implementations of the techniques.

The proof-of-concept model transformation implementations described in chapter are made available online [1].

5.2 Outline: Creating a Model Generator

Two artefacts are created when a meta model is applied to the presented implementation- a Kodkod re-representation of the given meta model and an instance-converting model transformation. The Kodkod tool support generates instances for a representation of the meta model defined using the Kodkod library, by exploiting SAT solvers. The created instance-converting transformation takes instances generated by the Kodkod representation and creates models

[1] http://cs.bham.ac.uk/~szs/thesis-software-demo/

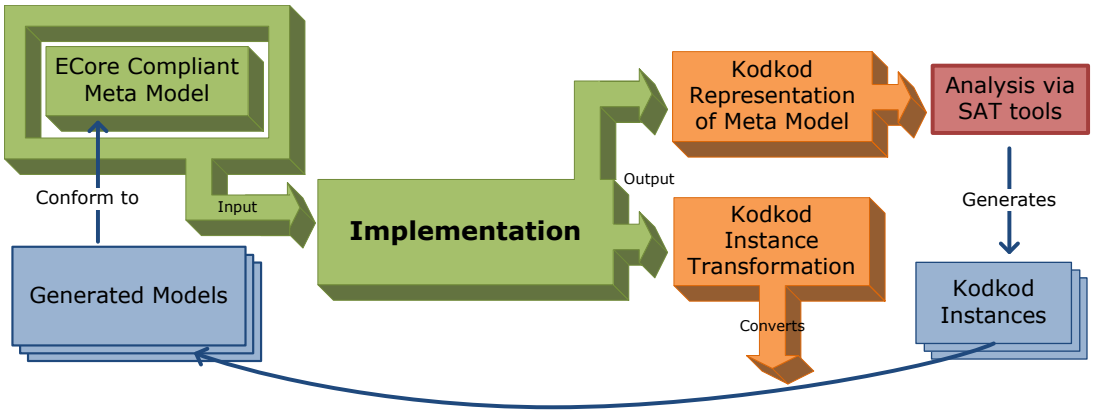


Figure 5.1: Overview of the implementation and the artefacts produced by the implementation.

that conform to the original meta model. The transformation of instances is required as the Kodkod instances are not models that conform to the original meta model, as described in section 5.4. Section 5.4 also describes the difficulties of creating a general algorithm for converting instances and the rational for exploiting the trace to create an instance converter. Together, the two artefacts created by the implementation form a model generator for the given meta model. An overview of the implementation is shown in figure 5.1.

The first stage in the implementation involves is the conversion of a given Eclipse Modelling Framework [137] ECore meta model to Kodkod [141], a SAT-based model finding library in Java. This transformation is a meta model transformation composed of two parts, firstly the conversion to an intermediate, *Fur* model-based representation, followed by the creation of the actual textual representation of the model in Kodkod. The Fur meta meta model is a wrapper for Kodkod allowing for the creation textual Kodkod models by meta model transformation.

EMF2Fur uses the SiTra [2] model transformation framework in the conversion of ECore meta models, as detailed in section 5.3.1. When the EMF2Fur transformation is applied to a particular ECore meta model, a trace model records the element creation. Also in this stage, the Fur2Kodkod transformation converts a given Fur meta model into a Kodkod-Java textual form, by a Velocity model-to-text transformation, as detailed in section 5.3.2. The outcome of this stage is a Kodkod representation of the input meta model. The Kodkod representation can generate instances, but only in the Kodkod notation.

The second stage of the implementation uses the trace of an EMF2Fur transformation to create another transformation. The produced transformation is required to convert instances generated by the Kodkod representation that is produced in the first stage. The input of the second stage transformation is the trace, from which a transformation is produced, making the second stage a higher-order model transformation. As in the first stage, the second stage involves a conversion to an intermediate model-based form, followed by a model to text transformation.

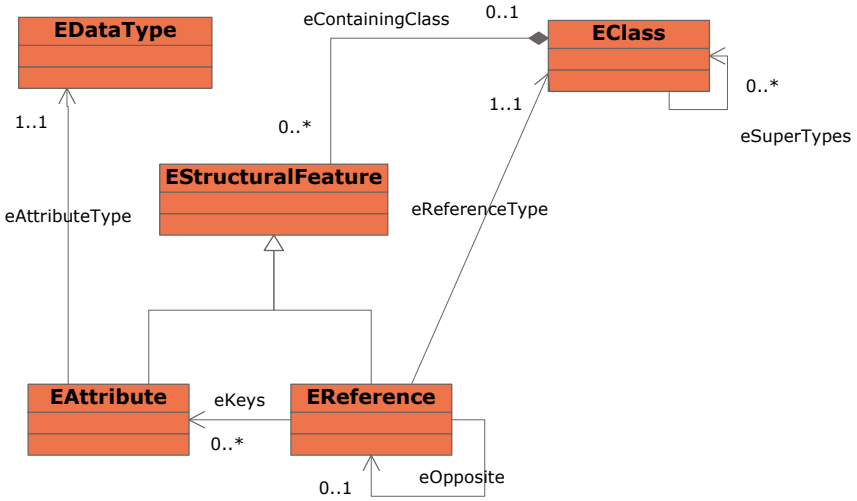
An intermediate form is used in the second stage as an abstract representation of an instance transformation. The concrete SiTra model transformations is derived from the intermediate from by model to text transformation. The creation of a transformation from a trace is detailed in section 5.5. The model to text transformation creates the Java-textual representation of the concrete instance transformer, based on an abstract model, details are given in section 5.5. The created transformation is specific to the original meta model and the Kodkod representation created by the EMF2Fur transformation.

Together, the transformation in first stage of the implementation, followed by the transformation in the second stage form a chain of transformations. The implementation creates model generators described in section 5.6. When applied to the ECore meta meta model, the implementation creates an ECore meta model generator, generating meta models for application to the implementation, for self-validation, briefly described in section 5.6. Future implementations of the technique may be applied in other contexts, including other modelling languages, as described in section 5.7.

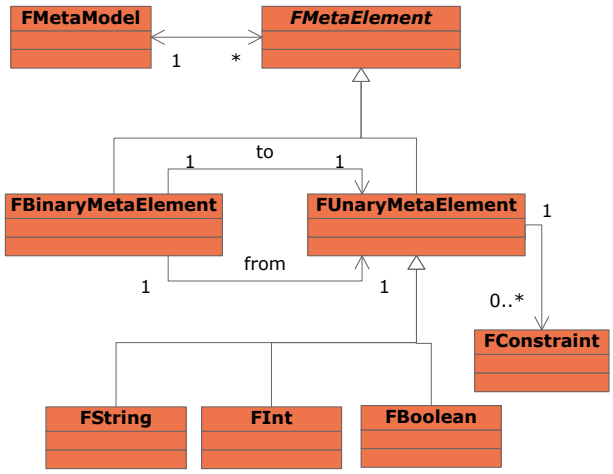
5.3 Stage One: Conversion to Kodkod Re-Representation

5.3.1 Meta Model Transformation : EMF2Fur

In this stage, a meta model conforming the ECore meta meta model is converted to Fur, which is an intermediate, model-based wrapper for Kodkod. Fur is created for this work, to



(a) ECore kernel meta model.



(b) Fur meta model.

Figure 5.2: Meta models used in EMF2Fur model transformation.

bridge the wide gap between the ECore modelling and Kodkod text-based notations. The conversion of ECore meta models to Fur is performed by SiTra transformation rules. This section describes the ECore and Fur meta meta models. The SiTra rules used to convert ECore meta models to Fur are described. The presented transformation takes an ECore meta model and converts it to the corresponding Fur model. The Fur representation is converted to textual Kodkod notation in a second conversion, discussed in section 5.3.2.

The Eclipse Modelling Framework: ECore Meta Meta Model

The Eclipse Modelling Framework "EMF" uses the ECore meta meta model in the modelling hierarchy. In the framework, ECore is used for specifying modelling formalisms, such as UML Class Diagrams or UML State machines, so such formalisms may be used in the EMF. Also the ECore meta meta model is also directly used to represent models of software systems. The current implementation applies to any ECore meta model, and will create an instance generator.

The ECore meta meta model is available in two forms: the kernel and extended versions. The kernel ECore meta model is the self defining subset of the extended ECore meta meta model [137]. That is, ECore kernel can be used to define itself and the extended ECore meta meta model. The ECore meta meta model can be represented using elements from the ECore meta meta model, as follows. EClass, EReference and EAttribute are created a EClasses. Relationships between the EClasses of EClass, EReference and EAttributee are EReferences. The attributes of the EClasses of EClass, EReference and EAttribute are EAttributes.

In this implementation the kernel variant is used, as shown in figure 5.2a. The ECore kernel meta meta model has four main elements: EClass, EReference, EAttribute and EPrimitiveTypes. An EClass may have zero or more EAttributes or EReferences. An EClass may also be related to another EClass, as a sub-class, from which structural features such as attributes and elements are inherited.

An EAttribute represents the attributes of an EClass. Every EAttribute belongs to an EClass and has an EPrimitiveType, which may be one of EBoolean, EInteger or EString. Every EReference also belongs to an EClass and has an EClass as the type. EReferences are used to represent the relationship between EClasses. An EReference may also have an optional opposite EReference, used to represent the same relationship, in the opposite direction. EAttributes and EReferences are structural features. EClasses and EPrimitiveDataTypes are data types. Each of the four elements may have a name and EClasses may also have EAnnotation.

Elements of a meta model can have subtle constraints on the allowed elements and relationship values. Constraints on meta model are created by adding EAnnotations to elements

of the meta model. The body of an EAnnotation is not defined by ECore itself. Instead, a separate textual notation is used to represent constraints. In the current implementation, the Object Constraint Language is used.

The Fur Meta Meta Model

The Kodkod formalism is a Java and text-based notation, in contrast to element-relationships of modelling notations. Despite this, the constructs of Kodkod notation can be expressed in a meta meta model. To simplify the conversion of ECore meta models to textual Kodkod, the intermediate Kodkod meta model: Fur is defined in this section. Obvious differences between the ECore and Kodkod notations are the lack of attributes and textual form of Kodkod representations. The Fur meta model is a wrapper for the structural elements of the Kodkod formalism, to allow representation of such elements in model-based manner.

With the definition of the Fur meta model wrapper for Kodkod, a hierarchy is created for Kodkod models, comparable to that of ECore. The Fur meta model is used to describe a Kodkod model as shown in figure 5.2b. A Kodkod model has instances, as created by the analysis of a SAT solver, Fur is also used as a wrapper for those instances. By the created Fur wrapper, Kodkod can be used in a modelling context.

The Fur meta meta model consists of the FMetaModel class, related to zero or more FMetaElements. An FMetaElement may either be an FUnaryMetaElement or an FBinaryMetaElement. The FBinaryMetaElement represents a relation between two FUnaryMetaElements. An FUnaryMetaElement has a label and is related to zero or more other FUnaryMetaElements, from which the FUnaryMetaElement inherits relationships. An FUnaryMetaElement may also have textual constraints, represented by FConstraints.

SiTra Rules: EMF2Fur

The Simple Transformer "SiTra" framework can be used to define rules to convert meta elements of a source meta model to another, target meta model. Conversion in the current implementation is done by creating SiTra rules. A rule defines how an element of the source

meta model is used to create an element of the target meta model. Collectively, a group of implemented SiTra rules form a model transformation. SiTra provides an environment to execute and apply a group of rules to a given model, by the SimpleTransformer.

The implemented EMF2Fur transformation is a meta model transformation using SiTra. Meta models in the ECore modelling formalism conform to the ECore meta meta model. Each rule defines how an ECore element is used to create Fur element(s). Each element in an ECore meta model conforms to an element of the ECore meta meta model. Application of EMF2Fur rules to elements in a ECore meta model results in a corresponding Kodkod representation of the ECore meta model.

The SiTra transformer can be used to apply a transformation to a given model. A rule in SiTra is defined between S, a source element and T a target element. In a particular rule, S is a selected meta element of the source meta model and T a selected meta element of the destination meta model. A rule in SiTra is defined as a Java class that implements the SiTra rule interface. A SiTra rule must denote S, the kind of element of the source meta model and T, the corresponding kind of destination meta model element created when the rule is applied.

The SiTra Rule interface defines three Java methods that each SiTra rule must implement: *check*, *build* and *setProperties*. In a SiTra rule, the check method ensures the rule applies to a given instance of S. If the check is positive for a given element S, the rule (build and setProperties methods) apply to that S. The applicable build method creates a corresponding T element, based on a give S. The T created by build is minimal, properties of T are not set in the build method [1]. Additional properties of the instance T is set in the setProperties method of the rule. SiTra transformation rules are Java classes that implement the Rule interface, with methods defining the conversion of a given source element.

In the current SiTra meta model transformation, the source meta meta model is ECore and the target is the Fur meta meta model. The rule EMetaModel2FMetaModel is the entry point of the meta model transformation. Note that an EResource may be composed of multiple

[1]The properties of T may require the application of another rule and this may lead to infinite and recursive rule application

Rule	ECore Element	Fur Element
EMetaModel2FMetaModel	EResource	FMetaModel
EClass2FUnaryMetaElement	EClass	FUnaryMetaElement
EReference2FBinaryMetaElement	EReference	FBinaryMetaElement
EAttribute2FMetaElements	EAttribute	FUnaryMetaElement
EAttribute2FMetaElements	EAttribute	FBinaryMetaElement
EBoolean2FBooleanMetaElement	EBoolean	FBooleanMetaElement
EInt2FIntMetaElement	EInt	FIntMetaElement
EString2FStringMetaElement	EString	FStringMetaElement
EAnnotation2FConstraint	EAnnotation	FConstraint

Table 5.1: EMF2Fur: rules to convert elements of ECore meta models to elements of Fur meta model.

EPackages (groupings of meta elements) and can also reference external packages. As Kodkod does not have meta elements to represent packages directly, the elements from each package are treated as a single grouping in the resulting FMetaModel. In practice, the rule collects all the elements of the given EResource, applying the rules of the transformation to the elements to build a FMetaModel equivalent.

An outline of the meta model transformation rule is shown in table 5.1. Each EClass of the ECore meta model is converted by the EClass2FUnaryMetaElement rule, producing a corresponding FUnaryMetaElement element. EReferences are converted to FBinaryMetaElements. The conversion rule EReference2FBinaryMetaElement uses the source and type of EClass of an EReference to create the 'from' and 'to' parts of a FBinaryMetaElement.

As Kodkod has no corresponding meta element to represent EAttributes, the rule to convert EAttributes: EAttribute2FMetaElements rule is complex compared to previous rules. The rule is not a one-to-one mapping, one EAttribute is used to create three elements by the rule EAttribute2FMetaElements; one FUnaryMetaElement and two FBinaryMetaElements. The FUnaryMetaElement represents the attribute. The rule also create an FBinaryMetaElement to relate the FUnaryMetaElement of the EAttribute to the FUnaryMetaElement of the EClass to which the EAttribute belongs. The second FBinaryMetaElement created by the rule relates the FUnaryMetaElement of the EAttribute to the relevant data type in the FMetaModel.

The primitive data types used in an ECore meta model are converted as simulated data

types in the FMetaModel. Creating primitive data types, such as integer boolean and string in Fur requires a specific arrangement of elements in the FMetaModel. Data types are converted by the rules only once per data type, as a given data type represented in Fur may be re-used. The EString data type is simulated by an FStringUnaryMetaElement. The EInt data type is simulated by the FIntUnaryMetaElement and the EBoolean data type by the FBooleanMetaElement. Each created data type element in the resulting Fur meta model is re-used for each converted EAttribute of the equivalent type in the original meta model.

The textual constraints in the ECore meta model are encapsulated by EAnnotations. The contrivances are, without any modification, copied over into FConstraints elements and later converted to Kodkod logical constraints (described in section 5.3.2). The invariants of EClass elements are converted using by a conversion specified in table 5.2. The conversion excludes any constraints not defined in OCL.

The EMF2Fur model transformation is a conversion of an ECore meta model such that the produced Fur representation uses only elements allowed in the Kodkod formalism. The transformation of a specific ECore meta model to Fur is also traced. Each rule application is recorded in the trace by the SiTra transformation framework, as a trace instance. Each trace instance records an element of the given ECore meta model, the rule applied to it and the created element resulting Fur representation. The trace of such a conversion is used as a basis for the second stage, to create an instance transformation and described further in section 5.5.

5.3.2 Fur Meta Model to Textual Representation : Fur2Kodkod

In this part of the implementation, a Fur representation of an ECore meta model is converted to the textual Kodkod notation. The conversion of Fur a meta model to Kodkod is done by a model to text transformation, Fur2Kodkod. The Fur representation is a re-arrangement of elements of the ECore model using only the elements available in the Kodkod formalism. The Fur2Kodkod conversion, converts the Fur meta model which has both structural elements and textual constraints, into the textual-only Kodkod-Java notation. The created Kodkod

representation generates instances when analysed via Kodkod and SAT solver.

The Fur2Kodkod conversion is implemented as a template-based model to text transformation. The Apache Velocity template language and tool [66] is employed in the implementation, via the available Java library. A template outlines the essential form of the textual notation, without the elements required to represent a model in the notation. Instead, the velocity template includes constructs in the Velocity notation. The Velocity constructs affect how a given model is used to textual elements. The Velocity engine takes as input a Velocity template and a model, as a Java object. The template and model are merged by the Velocity engine to create a textual representation of the model.

In the current scenario, the Velocity template holds the generic outline of a Kodkod model. The generic Kodkod model in the template follows the pattern of known Kodkod implementation models [141]. The template is merged with a meta model in the previously presented Fur formalism. The template defines how a given Fur model is interpreted to create a corresponding textual Kodkod representation. The Velocity template applies is used to crate a textual Kodkod representation, from a Fur model.

The Kodkod template used in the model to text transformation has three sections, as outlined in figure 5.3. Each section of the template is initialised by the Velocity engine, based on the input Fur meta model. In the first section of the template, the Kodkod relation declarations are created, based on the Fur meta model elements. In the second section, constraints in Kodkod logic are declared over the relation declarations made in the first section, also based on the Fur meta model. In the third section, bounds of the relation declarations of the first section are set, within which analysis is performed. Also in the last section of the template, means access to the Kodkod relations and generated instances are defined, based on the Fur meta model elements.

In the first section of the Velocity template for Kodkod, the elements of the Fur model are used to create Kodkod-relation declarations. Each FUnaryMetaElement of the Fur meta model is used to create a textual Kodkod relation with an arity of one. Similarly, each FBinaryMetaElement of the Fur meta model is used to create a textual Kodkod relation declaration

```
package $FMetaModel.getPackage();

//import statements ...

public class $FMetaModel.getName()
                implements Iterator<FInstance>, Iterable<FInstance>{

#foreach( $element in $MetaElements )
        private final Relation $element.getName();
#end

        //constructor to initialise (meta) models objects
        public $FMetaModel.getName()(){
#foreach( $element in $MetaElements )
                $element.getName() =
                        Relation.nary("$element.getName()",
                                $element.getArity());
#end
                solver = new Solver();
                solver.options().setSolver(SATFactory.MiniSat);
                solutions = solver.solveAll(this.declaration(),
                                                        this.bounds());
        }

        //builds the Formula that constrains the elements
        private Formula declaration(){
                final List<Formula> decls = new ArrayList<Formula>();
                //declare the model
#foreach( $element in $MetaElements )
...
#emd
                return Formula.and(decls);
        }

        //sets the bounds for simulation of the model
        private final Bounds bounds(){
                ...
                return b;
        }

        //accessor methods, used by instance convertor
#foreach( $element in $MetaElements )
...
#end
        //iterator method...
}
```

Figure 5.3: Velocity template for Fur2Kodkod model-to-text transformation.

```
--OCL fragment
context Shop
 inv: Shop.allInstances() ->
 forAll(s:Shop | s <> self implies s.manager <> self.manager)

//corresponding Kodkod expression
Variable s = Variable.unary("s");
Variable self = Variable.unary("self");
s.eq(self).not()
        .implies( ( s.join(_Manager).eq(self.join(_Manger)) ).not() )
                .forAll( s.oneOf(_Shop).and(self.oneOf(_Shop)) );
```

Figure 5.4: Example conversion of OCL constraints to Kodkod logic.

with an arity of two. The names of Fur elements are used as labels in the Kodkod declarations. Where no name is available for a Fur element, a unique but arbitrary label is used in the Kodkod declaration.

In the second section of the Velocity template for Kodkod, the textual Kodkod constraints are defined over the relation declarations in the first section. Additional Kodkod constraints are necessary to represent certain relationships. For example, a given FUnaryMetaElement may inherit properties from another element. As modelling-inheritance is subtly different from Kodkod relational inheritance. Additional Kodkod constructs are required to make sure that any instance of a sub class is also an instance of the superclass. Similarly, for FBinaryMetaElements require specific textual Kodkod definition of multiplicity constraints.

Also this section, the textual constraints of the original meta model defined in the Object Constraint Language "OCL" are converted to corresponding textual Kodkod logic. Only the OCL constraints of EClass elements, that is, EClass invariants are converted in the current implementation. The meta model constraints are parsed into a model using the EMF OCL libraries, and converted in a program that employs a visitor pattern. Table 5.2 outlines the conversion of a subset of OCL constraints to Kodkod logic. The objective of the current implementation is not to provide a complete conversion of the OCL to Kodkod logic; only a subset of the OCL is converted to Kodkod logic. Further details of a more complete transformation of OCL to the similar Alloy relational logic can be found in [8, 11, 9]. Optionally, an expert in the Kodkod formalism can also manually define constraints in the created Kodkod

OCL Expressions	Corresponding Kodkod Expression(s)
`context C inv: expr1 inv: expr2`	`expr1.and(expr2)`
`expr1.expr2`	`expr1.join(expr2)`
`forAll ( expr1 | expr2 )`	`expr2.forAll(expr1)`
`select ( expr1 | expr2 )`	`expr2.forAll(expr1)`
`exists ( expr1 | expr2 )`	`expr2.oneOf(expr1)`
`var:Type`	`Variable var = Variable.unary("var")` `var.oneOf(Type)`
`self [Context C]`	`Variable var = Variable.unary("var")` `var.oneOf(C)`
`not expr1`	`expr1.not()`
`expr1.size()`	`expr1.count()`
`expr1 = expr2`	`expr1.eq(expr2)`
`expr1 <> expr2`	`(expr1.eq(expr2)).not()`
`expr1 and expr2`	`expr1.and(expr2)`
`expr1 or expr2`	`expr1.or(expr2)`
`expr1 xor expr2`	`expr1.xor(expr2)`
`expr1 implies expr2`	`expr1.implies(expr2)`
`if bool1 then expr2 else expr3 endfif`	`if(bool1).thenElse(expr2,expr3)`
`integerliteral`	`IntConstant.constant(integerliteral)`
`intexpr1 > intexpr2`	`intexpr1.gt(intexpr2)`
`intexpr1 < intexpr2`	`intexpr1.lt(intexpr2)`
`intexpr1 >= intexpr2`	`intexpr1.gte(intexpr2)`
`intexpr1 <= intexpr2`	`intexpr1.lte(intexpr2)`
`intexpr1 + intexpr2`	`intexpr1.plus(intexpr2)`
`intexpr1 - intexpr2`	`intexpr1.minus(intexpr2)`

Table 5.2: OCL and corresponding Kodkod expressions: outline of OCL2Kodkod transformation.

representation.

As noted, only a subset of the OCL can be converted to Kodkod logic. However, if necessary, the OCL in a given meta model can be manually re-factored to use only the OCL expressions that can be converted. The work by Cabot and Teniente [37] describes how there are syntactic alternatives when authoring OCL constraints. In that work, the aim is to uncover the optimal (efficient, comprehensible) means to denote the same constraint. The significance to the current work is that an unsupported OCL constraint can be re-written to use only the supported constructs from table 5.2. For example, (from [37]) some OCL invariants using the *allInstances()* construct could be converted to a simpler invariant that only references *self* [1]. This way, a constraint using *allInstances()* can become supported by the contructs in the table 5.2. Further research is required on how unsupported OCL statements could be re-factored to become supported and how the process could become automated.

In the third section of the Velocity template for Kodkod, the bounds are created of Kodkod relation declarations made in the first section. Each FUnaryMetaElement has a positive integer bounds is set in the textual Kodkod notation. The bounds define the limit within which analysis is performed by the Kodkod tool. Each FBinaryMetaElement must also have a bounds set; the bounds is the product the FUnaryMetaElements related by the FBinaryMetaElement. Finally, methods in the Kodkod notation are added to retrieve the instances produced by the analysis of a SAT solver. The additional methods also include access to the relation elements created in the first part of the template.

This stage of the implementation has presented the two-part model to model and model to text conversion of a given ECore meta model to Kodkod representation. The first model to model transformation, EMF2Fur creates an intermediate representation in Fur. Fur is created as a formalism with only analogous elements found in the target Kodkod form. A second model to text transformation creates a textual Kodkod representation from the intermediate Fur representation. The created Kodkod model, generates instances via Kodkod and a SAT solver. The instances created in Kodkod are not instances the original meta model, so require

[1] The complete example is given in [37]

conversion. In the next section, a discussion of the difficulties of converting Kodkod-produced instances into ECore meta model instances.

5.4 Kodkod Instance Conversion: Rational for Model Transformation based on Trace

The Kodkod representation is able to generate instances, in the Kodkod formalism. The instances generated in Kodkod do not conform to the original ECore meta model. Many instances can be generated by the Kodkod representation There are natural differences between the Kodkod and the ECore formalisms and many instance can be generated, so the instances must be converted mechanically to become usable model of the ECore meta model. The conversion of instances requires consideration of how elements of the orignial meta model are converted to Kodkod.

In the presented implementation, there are three artefacts toward generating valid models of a given ECore meta model. The original meta models in ECore form is a complex specification, denoting the allowable models in a modelling formalism. The model transformation EMF2Kodkod converts a given ECore meta model into a corresponding Kodkod representation. Each unique ECore meta model, when applied to the presented transformation produces a unique Kodkod representation. When the Kodkod representation is analysed by the minisat SAT solver [53], Kodkod instances are automatically created. The minisat SAT solver is automatically invoked by the Kodkod libraries at runtime. The instances produced by analysis are instances of the Kodkod representation, not of the ECore meta model. Instances generated by SAT-solver analysis of a Kodkod representation, are specific to that Kodkod representation, and must be converted to become ECore meta model instances.

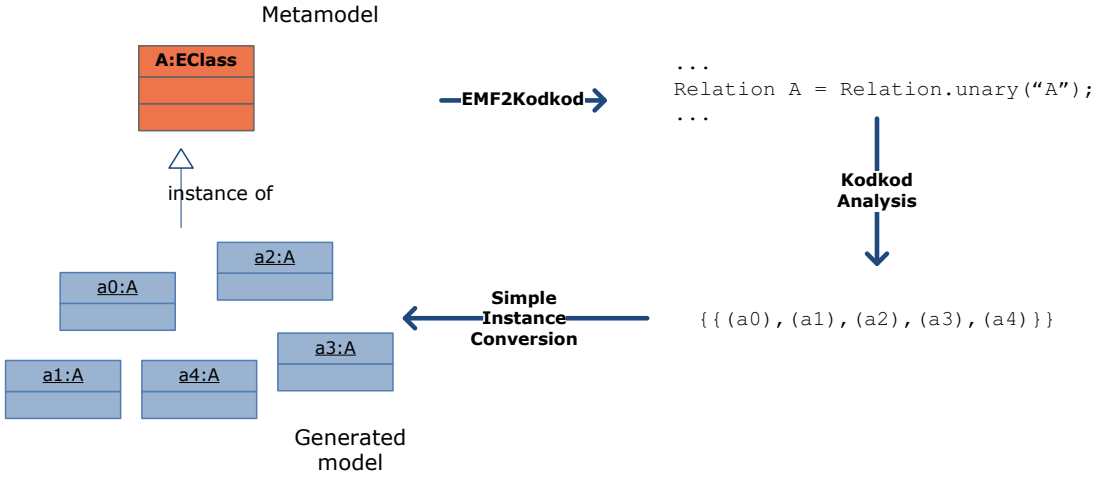
A Kodkod instance is produced from a Kodkod representation of a meta model by analysis. To convert a generated Kodkod instance, each element of the instance must be used to create a corresponding element in a model, to become a valid model of the original ECore meta model. The elements of a Kodkod instance are tuples, either binary or unary, that conform to the

binary and unary relations of the Kodkod representation. Each instance generated in analysis is specific to the Kodkod representation from which that Kodkod instance is generated. Each tuple of a Kodkod instance must be used to create a corresponding EObject of the ECore meta model.

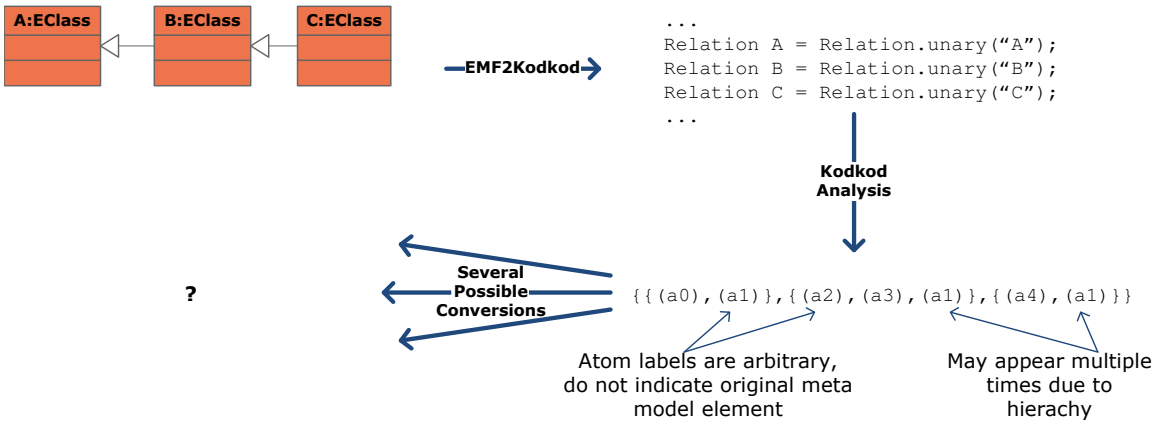
Given a tuple in a generated instance, a corresponding EObject for the tuple must be created in an ECore model. When creating an EObject, the EClass type of the EObject must be set. The EClass type is an element from the original meta model. By converting the tuples from a Kodkod instance, the created EObjects form a model of the original meta model. The corresponding EObject that must be created from a tuple depends on the relational arity of the tuple. Particularly, the element from the original meta model used to create the relation of a given tuple must be determined, as shown in figure 5.5a. Determination of the relation of a tuple is not always a straight-forward task.

To convert Kodkod generated instances, the relation of each tuple in an instance must be determined. This can be problematic when relational inheritance is represented in the Kodkod model as a unary tuple may appear multiple times in an instance. By inspection, it is not possible to determine the relation of a tuple, as atom labels are unreliable to determine the relation and can only be used to infer the most general relation of the tuples. The tuples of a generated instance may conform to the Kodkod relations with inheritance in a number of possible ways, as shown in figure 5.5b. Several distinct ECore models can be created from the tuples are possible given such a scenario. Instead, the Kodkod API and the accessors methods created in the first stage of the implementation must be used to determine the most specific relation of a tuple.

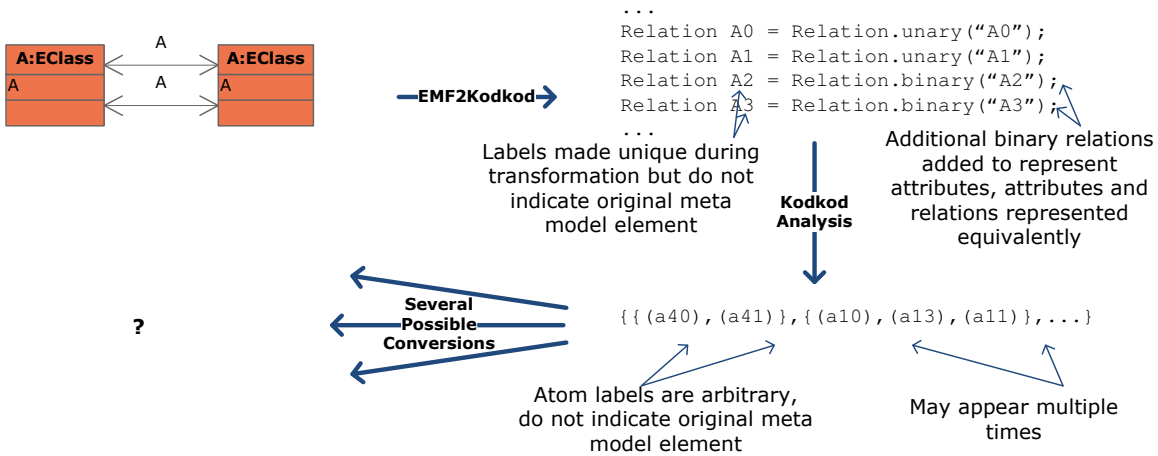
Once the relation of a unary tuple is determined, a corresponding EObject must be created in the ECore notation. The EClass (type) of the created ECore EObject is the EClass that was used to create the Kodkod relation of the tuple, in the EMF2Kodkod transformation (as shown in figure 5.5a). To create an ECore model from a Kodkod instance requires information on the correspondence between Kodkod relations and ECore meta elements The correspondence between ECore meta model elements and Kodkod relations can not be determined by



(a) Simple Kodkod to ECore instance conversion scenario.



(b) Kodkod to ECore instance conversion: problematic due to duplicate atoms and hierarchy.



(c) Kodkod to ECore instance conversion: problematic due to duplicate, arbitrary names and binary relations.

Figure 5.5: Kodkod to ECore instance conversion scenarios.

inspection alone. This is due to the possible lack of labels and possible duplicate labels in ECore, making labels unreliable to determine the correspondence between Kodkod relations and ECore meta elements. Furthermore, ECore EAttributes and EReferences are represented in the same way in a corresponding Kodkod representation. So extra relations are created when re-representing an ECore meta model so extra tuples can appear in a Kodkod generated instance. A general transformation to convert any given Kodkod generated instance to the corresponding ECore meta model instance is not possible due to unreliability of labels and the difference between formalisms.

To illustrate how it is not possible to determine the correspondence of ECore classes and the relations in a Kodkod representation, consider the example in figure 5.5c. Names are not required on all elements in the meta model and duplicate names are allowed in ECore meta model. The EMF2Kodkod transformation assigns arbitrary unique labels to relations in the meta model. Furthermore, the ECore meta model EAttributes and EReferences are represented using the same kind of element, binary relations in Kodkod. To determine the correspondence between an ECore meta model and corresponding Kodkod representation is not possible by inspection alone.

There are several possible mechanisms to determine the correspondence between an ECore meta model and the corresponding Kodkod representation, to enable the conversion of instances. Firstly, by manual expert interaction. It is possible to convert each Kodkod generated instance by an interactive conversion. An interactive conversion would require inspection of the original ECore meta model and the create Kodkod model to determine the correspondence between ECore meta elements and Kodkod relations. By manually determining the correspondence, instances can be converted. However, the inspection of artefacts is prone to errors and the conversion of models may be performed incorrectly. Furthermore, A large number of instances are generated by the Kodkod representation, so manual conversion of generated instances is time consuming.

A slightly more advanced scheme is possible using expert knowledge to create an instance transformation. Instead of converting each instance manually, an instance transformation can

be developed by using expert knowledge. By using expert knowledge of the correspondence between the ECore meta model and the Kodkod representation, to create model transformation. The manually created transformation can then used to automate the conversion of instances. However, such a scheme is not ideal as transformation must be manually created for each unique ECore meta model and corresponding Kodkod representation, using expert knowledge. Also, as any model transformation, a manually created model transformation for Kodkod instances is also prone to developer error. Thus, manual conversion of generated instances is time consuming and error prone, as is the manual creation of a model transformation.

A more advanced scheme may be possible to convert Kodkod generated instances. By manually modifying the ECore model to add identifiers an algorithm to convert instances may be possible. The original ECore meta model may be modified to ensure all names are unique in the meta model, by adding unique identifiers to each label. Unlabelled elements in the ECore meta model can be modified and labelled using arbitrary names, before applied to the EMF2Kodkod transformation. Further, EReferences and EAttribute element names in the ECore meta model may modified to add an identifier to indicate whether the element is an EReferences or EAttribute. When modified in this way the names an ECore meta model and the created Kodkod representation may be used to determine the correspondence between the elements created by EMF2Kodkod transformation. The correspondence can then be used to convert Kodkod instances, by an algorithm. In effect, the models would have been modified to add tracing information. However, such a scheme is unnecessary as the correspondence between ECore meta model and the created Kodkod representation is available in the EMF2Kodkod transformation trace.

It is proposed that the trace of an EMF2Kodkod transformation is used to automatically create an instance converter. The Kodkod generated instances are models, as are the ECore models that must be created from them. Automated conversion of instances is required as many instances are generated by a Kodkod representation of a meta model. So the trace is used as the basis for creating a model transformation. The process of creating a model

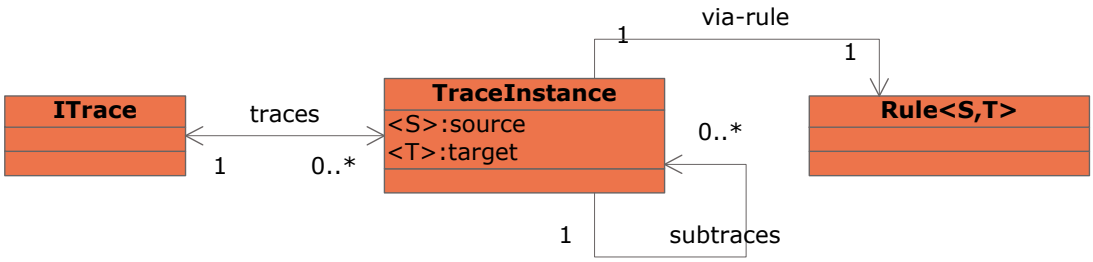
transformation from a trace can be automated by a higher-order model transformation. In the created instance transformation, Kodkod instance elements can automatically be used to create the corresponding ECore model elements.

Manual conversion or a manually created model transformation to convert Kodkod generated instances is not ideal. Instead, the trace of a particular application the EMF2Fur transformation can be used to derive, automatically an instance model transformation. In the proposed scheme, the conversion on Kodkod generated instances to ECore meta model instances would be done automatically. The instance converting model transformation created by this scheme is specific to the unique ECore meta model and the corresponding Kodkod representation. The created Kodkod instance transformation is based on the trace of the EMF2Kodkod conversion. The conversion of trace to instance model transformation is automated by a higher order model transformation, as described in the next section.

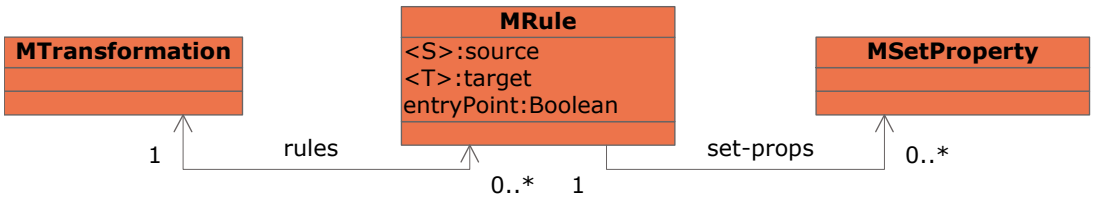
5.5 Stage Two : Creation of Kodkod Instance Converter from Trace

The previous stage of the implementation defines a two part meta model transformation. A given ECore meta model is applied to the transformation to create a corresponding representation in the Kodkod formalism. The Kodkod representation generates instances that satisfy the specification of the given ECore meta model. However, the instances produced by Kodkod are not instances of the original meta model, due to natural differences in the ECore and Kodkod notations.

The conversion of Kodkod instances is specific to the ECore meta model and Kodkod representation. Each unique ECore meta model produces a unique corresponding Kodkod representation. The difference between ECore and Kodkod formalisms requires that Kodkod generated instances are converted. The current stage of the implementation solves the problem by the creation of Kodkod to ECore instance transformations based on a trace of the EMF2Fur transformation. The instance transformations produced in this stage are specific to



(a) TraceMM: Trace Meta Model.



(b) MTransformation: Transformation Meta Model.

Figure 5.6: Meta models used in Trace2MT Model Transformation.

the unique ECore meta model and corresponding Kodkod representation.

The transformation of a trace to Kodkod instance transformation is automated by model transformation. The input to this higher-order model transformation is a trace. The outcome is a Kodkod instance model transformation. The transformation input is defined by the trace meta model and the output is defined by the meta model of model transformation. The higher-order transformation defines rules between those meta models so that a given trace of an EMF2Fur transformation is converted to a model transformation.

The trace of the EMF2Fur transformation is created by the SiTra transformation engine, when an ECore model is applied to the transformation. The trace meta model TraceMM, shown in figure 5.6a. The trace meta model is based on the meta model in [133] and used in the current implementation to record the trace as a trace model. A TraceInstance is created for each application of a rule in the model transformation. The trace meta model defines an ITrace element; all TraceInstances recorded in a model transformation are related to a single ITrace element. A TraceInstance stores the source element of the input model and target element created from the source element. The rule that was applied to the source element to create the target model element is also recorded in the TraceInstance. A trace model is automatically populated by the SiTra transformation engine when a model is applied to a model transformation.

Rule	Trace Element	MTransformation Element
ITrace2MTransformation	ITrace	MTransformation
TraceInstance2MRule	TraceInstace	MRule
TraceInstance2MSetProperty	TraceInstace	MSetProperty

Table 5.3: Trace2MT: rules to convert trace to model transformation.

In the current implementation, the higher order transformation requires the creation of a meta model transformation, shown of figure 5.6b. Meta models of model transformation for other model transformations are available in the literature [115, 71]. In the current implementation, the MTransformation meta model for SiTra is defined for representing a Kodkod instance transformation. The meta model consists of three meta elements: MTransformation, MRule and MSetProperty. MTransformation represents a model transformation by a relationship with several MRules. An MRule defines a meta source and a meta target element type, and may be an entryPoint rule. There may only be one entryPoint MRule in a given MTransformation. Each MRule may also be related to zero or more MSetProperty elements. A MSetProperty elements defines a property of the target element that is set based on some property of the source element of the rule.

In the current implementation, an EMF2Fur model transformation is traced. The trace of the transformation is converted to a Kodkod instance transformation by the Trace2MT transformation. The Trace2MT model transformation is defined by three rules, outlined in table 5.3. The rule ITrace2MTransformation creates a single MTransformation element, from a singleton ITrace element. TraceInstance2MRule creates an MRule element from a TraceInstance element and applies only to trace elements where a FUnaryMetaElement is recorded in the trace.

In the Trace2MT transformation the rule TraceInstance2MSetProperty creates an MSetProperty element, from a Trace instance. The rule Instance2MSetProperty applies to TraceInstances where a FBinaryMetaElement is recorded in the trace. The created MRule and MSetProperty elements are created using information from the trace elements. In creating an MRule, the recorded source and target of the TraceElement are used to determine the types

in an MRule. Collectively, the created MRules define the creation a model of an ECore meta model, from a given a generated Kodkod instance.

The model based representation of a transformation conforming to the MTransformation meta model must be converted to the textual notation of the SiTra transformation formalism. The conversion of the MTransformation model to a model transformation is done by a model to text transformation, using the Velocity template engine [66]. The application of Velocity for model to text transformation is described in section 5.3.2. The model to text conversion in this stage differs and involves the creation of a textual model transformation in SiTra. The current Velocity template outlines a SiTra rule and the template is applied to a MTransformation model. The result of the application is a number of SiTra rules, one for each MRule element in the MTransformation model. The produced SiTra rules represent a Kodkod instance transformation.

The Velocity template of this stage defines the essential features of a SiTra rule, as outlined in figure 5.7. The template produces a SiTra rule when applied to an MRule element of a MTransformation model. The template has three sections, with each section corresponding to the three parts of a SiTra rule: the check, build and setProperties methods. In the check section, the template uses the source and destination types of the MRule element to declare the SiTra rule signature and the check part of the rule. Similarly, the build part of the SiTra rule is created, also using the source and destination types of the MRule. Each MSetProperty element of an MRule is converted to create a part of the set properties definition of the created SiTra rule. The SiTra rules and transformation created are specific to the MTransformation model and is also specific to the given trace.

The check, build and set properties parts of the Velocity template for MRules is not generic; the template uses specific constructs from the Fur and ECore notations for creating and accessing model elements. The build part of the template employs specific elements from the Fur to query instance elements. The build and set properties sections of the rule also use notations specific to both ECore and Fur. The build and set properties sections use specific elements for creating ECore instance elements, querying Fur elements and setting values in

```
package $MRule.getPackage();

//import statements
import $MRule.getSource().getCanonicalName();
import $MRule.getTarget().getCanonicalName();

import sitra.Rule;
import sitra.Transformer;

public class $MRule.getName()
        implements Rule<$MRule.getSource().getSimpleName(),
                        $MRule.getTarget().getSimpleName()>{

        //check the element is an $MRule.getCheckMethodName() relation
        public boolean check($MRule.getSource().getSimpleName() source){
                return source.instanceOf(
                        (($MRule.getTransformation().getInstanceGenenerator()
                        .getSimpleName())source.getInstance().getMetaModel())
                        .$MRule.getCheckMethodName()() );
        }

        //create the result
        public $MRule.getTarget().getSimpleName() build(
                                $MRule.getSource().getSimpleName() source,
                                Transformer t){
                EClass metaClass =
                        (EClass) metaModel.getEObject(
                                "$MRule.getUriFragment()" );

                EFactory factory =
                        metaClass.getEPackage().getEFactoryInstance();

                return factory.create(metaClass);
        }

        //set the properties of the object
        public void setProperties(
                $MRule.getTarget().getSimpleName() target,
                $MRule.getSource().getSimpleName() source,
                Transformer t){
#foreach ( $setProperty in $MRule.getSetProperties() )
...
#end
        }

}
```

Figure 5.7: Velocity template for MTranformation2SiTra model-to-text transformation.

ECore elements. The model transformation is specific to converting instances of a Fur models and ECore meta models.

The trace model is a record of element creation in the application of the EMF2Fur model transformation to a specific ECore meta model. The first part of the implementation is the conversion of a trace model to a model of model transformation. In the second part of the implementation, the model of model transformation is used to create a working SiTra model transformation. The model transformation create by the implementation to converts Kodkod generated instances to ECore models.

5.6 Outcome: Model Generator by Model Transformation to Support Transformation Validation

A model generator is created in two stages, the first stage creates an instance generator in the Kodkod formalism. The instance generator is produced by a meta model transformation, converting an ECore meta model to a generative Kodkod representation. By automated SAT solver analysis, Kodkod instances are created from the Kodkod representation. The second stage of the implementation creates a model transformation to convert Kodkod-generated instances into models of the original ECore meta model. The implementation creates a generative representation and instance converter from a given meta model.

The presented implementation can be applied to create a model generated that is used for self-validation. The ECore meta meta model can also be applied to the presented implementation. In this application, the implementation will create a meta model generator for the ECore meta meta model. The meta model generator automatically generates meaningless but valid ECore meta models. However, when the generated meta models are applied to the implementation, then the implementation can be validated. The implementation creates a meta model generator for ECore, that applies to validation of the implementation.

The implementation creates a model generator. The implementation can create an instance generator for the ECore meta meta model, to generate meta models. The generated

meta models can be applied to the implementation for self-validation.

5.7 Direction for Future Implementation

5.7.1 Issues of Setting and Automating Oracles for Validation

In the current implementation, the input for a model transformation can be generated. These models can be used to aid the validation of a model transformation. However, to carry out validation some method to measure the success of the validation must be set. In the simplest case, this can be done by expert inspection of the models input and the output of the transformation. However, this is problematic as it would require a great deal of expert time to inspect each generated model and the outcome of applying that model to validation.

As discussed in chapter 4 oracles for model transformation can be either specific models or generic properties. Specific models that can be used as oracles may be generated by extending the current implementation. Instead of only converting the source meta model as is currently done, the entire transformation including source meta model, destination meta model and transformation rules could be automatically converted to a generative analysis form such as KodKod or Alloy.

By converting entire transformations, the simulation tools of the analysis formalism could be used to generate an instance of the model transformation. In this context, a transformation instance is a particular source model, a sequence of rule application and the outcome source model. The analysis done could then be converted back to the modelling formalism and used for the validation. In effect the generated source model would be the validation data and the generated destination model would be used as oracle. A similar proposal has bee made in the literature [10], however this requires a manual conversion of transformation artefacts. Further experiments are required to determine the practical feasibility and scalability of such a proposal.

Instead, and as suggested in chapter 4, generic oracles that are more readily automated

may be used. The generic oracles are summarised in the table 4.1 with detailed description of each in section 4.4.2. Each of the generic oracles can be automated by the test harness as follows:

- Production of target model - this oracle relies on a model being produced when a test case is applied. To check this oracle is accomplished, the test harness checks that an output model file has be produced for the given test case. A potential error is uncovered if no file is produced.

- Relative size of target model - this oracle relies on a model of the expected model-file size being produced when a test case is applied. To check this oracle is accomplished, the test harness would check that the model produced has the expected size. This oracle requires an expert to set the expected size of output models, given the input model size. This calculation can then be performed using a script for each test case. Where the size of the produced model is not as expected it indicates a potential error has been uncovered.

- Meta-model conformance of target model - this oracle relies on the model produced conforming to the target meta model. To implement this oracle, the test harness must invoke the EMF validation framework (Described in detail in chapter 18 of [137]). Essentially, the meta model is loaded and registered with the eclipse modelling framework, then the produced model is loaded against the meta model. Finally the *Diagnostician.validate()* method is called for each element of the model. Where the validate method returns false, suggesting a model not conforming to the meta model was created; indicating a potential error in the transformation.

- By-product inspection - this oracle relies on the transformation engine producing errors when an error occurs for a given test case. To implement this oracle, the test harness can request access to the transformation log of ATL. Each entry in the transformation log can be compared (using regular expressions) against known failure messages, (for

example “error”, “failed”, “could not match”). The messages are specific to the transformation engine used (for example ATL). Where there is match between any log entry and know failure messages it indicates an error in the transformation process.

- Non-termination - this oracles relies on the transformation terminating for every given test case. To implement this oracle, the test harness can use a time-out for each test application. The time-out value is set by the validation expert, based on the expected time complexity of the transformation for a given input size. When the time-out value is reach for a given input, it indicates a potential non-termination error has been uncovered.

- Non-confluence - this oracle relies on a unique model being produced per input. To implement this oracle, the test harness can apply each test model twice. Optionally the one of the test case could be modified in a non-critical way e.g. serialising model elements in a different order. Then the output from both model applications can be compared. When the two output model do not match each other for a given input, it indicates a potential non-confluence error has been uncovered.

In each case where a potential error has occurred, an expert must be alerted to the issue via, for example, a log file. The above automated oracles are not perfect and can only *indicate* that an error has occurred. An expert must intervene as false positives are possible. For example, in the non-termination oracle, the time-out may be reached due to external factors such as background processes using CPU time or incorrect calculation of time-out values. Such manual checking is normally done as part of the debugging process. As the generic oracles can be automated, all of the generic oracles can be applied to each test case to determine the success of validation. In practice, the validation expert may select which of the generic oracles are suitable for the transformation being validated. Integrating fully automated and selectable generic oracles is an enhancement for future implementations.

5.7.2 Tools and Modelling Technologies

The implementation employs a selection of software tools; including SiTra, Velocity, Kodkod and EMF. The technique is not reliant on any of the tools used, future implementation of the technique can be created exploiting distinct tools. This section presents the pertinent features of the software tools used. The features used are discussed with regards to possible alternatives for future implementations.

A key feature of the current implementation is the application of the SiTra model transformation framework to convert software artefacts. The advantage of the SiTra framework is the low-level, imperative nature of a Java-based framework. This allows for the rapid prototyping and development of the framework in parallel to implementing the technique. In the current implementation SiTra was modified to support tracing model transformation, higher order model transformation and model to text transformations via Velocity. SiTra is also ideal for integration purposes, as Kodkod and the EMF libraries are defined in Java. However, model transformation frameworks based on the Queries, Views and Transformations [115] standard, the ATLAS [72] or Epsilon transformation frameworks may be used instead of SiTra in future implementations. Existing model transformation frameworks other than SiTra may be applied in future implementations of the technique.

The presented implementation concerns the creation of a model generator from a given Eclipse Modelling Framework meta model. The current implementation creates a model generator from ECore meta models and as such, the implementation applies only to ECore meta models. The technique converts meta models to a generative form, by meta model transformation. The definition of such a meta model transformation requires a meta meta model, such as ECore in the EMF. The ECore meta model is applied to the current technique as the EMF modelling hierarchy has a three levels. Other modelling frameworks with three meta levels, such as the Meta Object Facility [113] or the Kernel Meta Meta Model [71] may be used in future implementations.

The presented implementation is able to create a model generator for the ECore meta meta model, for self-validation of the implementation. Self validation is possible due to the

self-defining nature of the ECore meta meta model and the use of model transformations in all conversions. The ECore meta meta model, when passed into the implementation, creates an ECore meta model generator that can be applied to validate that implementation. In future implementations, this property may be replicated by using a self defining meta meta model, such as the Meta Object Facility or the Kernel Meta Meta Model. By application of technique proposed in chapter 3 and self-defining meta meta models, future implementations can also produce meta-model generators for self-validation.

Another key feature of the presented implementation is the use of the Java-based Kodkod library for the generation of instance models. Kodkod [141] is a textual software abstraction formalism, used for the analysis of a specification in Kodkod via a SAT solver. The current implementation required the creation of Fur, a model driven wrapper for Kodkod. Fur allows the application of the textual Kodkod formalism in model transformation. Similar textual analysis formalisms, such as Alloy [69] may be used in future implementations. Kodkod has been found more efficient than using Alloy in previous studies [141].

The current technique is not reliant on any tool and alternatives may be used in future implementations. A three level modelling hierarchy such as MOF or KM3 may be used instead of ECore in future implementations. For future implementations to allow for self-validation as in the presented implementation, model transformation and a self-defining input meta meta model such as ECore, KM3 or MOF must be used. Kodkod may be replaced by Alloy but would require a model-driven wrapper, similar to Fur, to allow the application in model transformation. There are a number of options for possible future implementations than those presented.

5.7.3 Scalability and Tractability of Meta Model Analysis using Kodkod

Kodkod and Alloy are relation formalisms that can generate instances from a given relational specification. As with meta models, relational specifications can be intractable; having an infinite number of valid instances. As discussed in section 4.3, such "bounded model finders"

are backed by a SAT solver to discover instances that satisfy the specification. One of the reasons SAT solvers can solve seemingly infinite relational specifications is because a limit (or *scope*) of analysis is set [69]. Thus the analysis results from model finders are only valid within that scope.

In the current technique, when a meta model is converted to create a model generator, a scope is set. The scope of the analysis affects both the scalability- number of models generated- and tractability- ease with which instances can be calculated- of the model generation. Meta models are typically infinite in terms of the number of valid instances. By setting a scope, the analysis is restricted to a subset of valid models. This allows the technique to take into consideration the complex meta model constraints whilst generating valid models. The scalability is affected by the scope because the number of models generated when considering the scope becomes finite. That is, although the meta model is infinite, a scope of analysis means models valid with respect to constraints can be generated and the numbers models generated is finite.

The number of models generated via Kodkod from a meta model specification depends on the meta model constraints and the scope that has been set. Over-constrained meta models may have no valid instances and constraint-free meta models may have many valid instances. The scope allows early identification of over-constrained meta models and limits the number of valid instances created from constraint-free meta models. In the current implementation a default scope is chosen for any given meta model, similar to the implementation in Alloy [69]. In [69], the authors argue a default scope of ten instances per element are sufficient for initial analyses.

The current implementation follows the example of [69], allowing up to ten instances of each meta element[1] by default. This means the theoretical maximum number of generated models for an unconstrained meta models using the current technique is 10^n, where n is the number of meta elements. However, in future implementations, is may be possible to either automatically determine a scope or to allow for a custom scope to be set based on the

[1]Meta elements include meta classes and meta associations.

requirements for the generated test cases, as is possible in implementations of Alloy [69].

5.7.4 Practical Observations: Advantages and Limitations of the Proposal

The main advantage of the presented implementation is the ability to create a model generator from a meta model. The approach is useful in that it considers the constraints of the meta model. This means a sequence of useful valid models can be produced given only a meta model.

Because only the input meta model is used to create the model generator, the presented approach has several advantages when used to support transformation validation. Firstly, being black-box means that no analysis of the internal structure of the transformation is required. This is useful because the technique can easily be applied to different transformation languages; it is not specific to any one model transformation formalism. Furthermore, as the generated models are applied to evaluate the code of the model transformation as a black-box, then the technique is not affected if the transformation code is changed. That is, the same model generator can be used to validate a model transformation even after the transformation is modified or developed further.

A further advantage of the current approach is that model transformations are used to create model generators. This means implementation of the technique can be used to create a meta model generator that can be used to assist the validation of the same implementation. This is referred to as the ability to support self-validation, i.e. the ability to generate test data the can be applied to the tool that creates model generators. In previous studies, it has not been possible to detect defects in the tools used to support validation.

Apart from the apparent advantages, there are several limitations of the implementation, when applied to model transformation. The first disadvantage relates to the inability to fully automate validation. The technique employs a validation harness. In the current implementation the validation harness– the piece of software that passes generated models to the validated transformation and checks the result– is not created automatically. This requires an

amount of manual developer interaction to create the harness and the oracle scripts to check the outcome of validation. The reason behind this is the harness is highly transformation language dependent, it must load and execute a model transformation with the generated models. The process is amenable to be automated if the technique is integrated with a model transformation development environment e.g. as an Eclipse/ATL plug-in.

Another limitation of current implementation relates to the subset of OCL that can be converted to Kodkod and the lack of constraints in the ECore meta meta model. Firstly, only a subset of OCL is converted to Kodkod. However, as input meta models may use other parts of OCL that are unsupported, the tool will fail to take these into account in the created model generator. However, a meta model with unsupported constructs may be re-factored to only use supported parts of the OCL, as described in [37]. Secondly, the ECore meta meta model does not include all constraints in a machine readable form. This means that some models may be generated that do not conform the ECore meta model. When these constraints are made available, the tool can be updated to support the creation of only ECore compliant models.

A final limitation of the approach in the inability to direct model generation based on coverage criteria. Coverage is used to ensure the generated models are effective in validation of the transformation. However, models are not generated by taking coverage of the meta model or transformation code into account. Expert interaction can be used to create the a model generator which increases meta model coverage. A discussion of coverage issues in the current approach is found in section 4.5. Further research is required on how to integrate and automate code coverage into the current approach.

5.8 Summary

This chapter has presented the implementation of the previously presented technique to create model generators from a meta model.

The current implementation involves a model transformation of an ECore meta model to

the Kodkod formalism. The transformation consists of a model to model transformation to the intermediate Fur form, and a model to text transformation, to create a Kodkod representation of an ECore meta model. The conversion results in a representation of the meta model in Kodkod, to generate instances.

The instances created by the Kodkod model are not models of the original meta model. The Kodkod and ECore formalisms have natural differences; for example Kodkod has no direct corresponding element to ECore attributes. A general conversion algorithm of Kodkod instances to meta model instances is not possible. The instances must be converted using information from the trace of the original conversion from ECore meta model to Kodkod form. The current implementation automatically also creates a model transformation from a trace; the created transformation converts Kodkod generated instances to ECore meta model instances.

A Kodkod instance generator and instance converter are created by the presented implementation. Together, the Kodkod instance generator and converter form a model generator for the given meta model. A model generator created by the implementation automatically creates models for use in the validation of a model transformation. The implementation can also create a meta model generator, for the ECore meta meta model. The ECore meta model generator generates meta models that apply to the validation of the presented implementation.

A selection of specific software tools are employed in the presented implementation. SiTra is used for the model transformation framework, ECore for the modelling framework and Kodkod for the SAT-based model finder. Future implementations are possible using different software tools. Transformation frameworks such as QVT, ATLAS or Epsilon; modelling frameworks such as MOF or KM3; and Alloy may also be used in place of Kodkod in future implementations.

In summary, the contributions of this chapter are as follows:

- The definition of a two-stage model transformation of meta models in ECore to the Kodkod notation, via the intermediate Fur formalism. This includes the creation of model-

to-model transformation rules and a model-to-text template transformation. The definition of a two-stage model transformation of a trace to a SiTra model transformation. This also includes the creation of model-to-model transformation rules and a model-to-text template.

- A justification for creating a model transformation from a trace for the conversion of generated instances.
- A brief discussion of the model generating artefacts created by the implementation apply to model transformation validation and to self-validation.

The presented implementation automatically creates a model generator, from a given meta model. The model generator is applicable for use in transformation validation. The next chapter presents an evaluation of the technique, by application of the created model generators to model transformation validation.

CHAPTER 6

AN EVALUATION OF THE APPLICATION OF MODEL GENERATORS TO MODEL TRANSFORMATION VALIDATION

Example is the best precept.
— Aesop (The Two Crabs)

6.1 Synopsis

This chapter is an evaluation of the previously presented technique by case study. The aim of the technique is to support the validation of model transformations by automatically creating model generators, that apply to the validation of a given transformation. The case studies in this chapter present how validation is supported by model generators. The studies involve the creation of three model generators, and application to the validation of Families2Persons, Tree2List and EMF2Fur model transformations.

The case studies demonstrate the application of the technique and the desirable properties of the technique. Model generators treat the model transformations as a black-box and generate models. An implementation of the technique is able to validate itself, by creating a ECore meta model generator and applying it to the EMF2Fur model transformation. As well as demonstrating situations where a transformation works, validation also applies to the detection of errors. Examples are given of errors detected by application of the technique to the Families2Persons and Tree2List transformations. In the evaluation, a comparison is made between the current technique and existing state-of-the-art techniques for model generation.

A proof-of-concept tool to create model generators is made available online, along with the automatically created code generator used to carry out the Families2Persons case study [1].

6.2 Aim and Methodology of Evaluation

Model transformations are used in various scenarios towards creating software; Lano [90] notes models may be refined, elaborated, re-represented, abstracted and analysed by model transformation. In a transformation with errors, those errors can be transmitted to results of transformation. If a model transformation with errors performs these activities, the benefits of automating the processes is negated. Furthermore, the dependability of the produced models is also put into question. Validation of transformation is vital for both demonstrating situations that work and for possibly detecting errors.

[1] http://cs.bham.ac.uk/~szs/thesis-software-demo/

The principle goal of this evaluation is to determine the success of the presented technique to support model transformation validation. Evaluation involving application of the technique will also demonstrate the properties of technique, demonstrating how the technique is applied in practice.

The method of the evaluation is case studies applying the technique to validation. Each case study involves the creation of a model generator by the technique and the application of the generator to validate specific model transformations. The case study is done using previously unseen model transformations, from the ATLAS model transformation collection [149]. Such transformations are suitable for the evaluation as they are developed independently of the current work. The evaluation is carried out by case study application of the technique to existing, previously unseen model transformations.

Criteria for the success of the evaluation must be set in advance. The desirable properties of any validation technique are discussed in chapter 3. Evaluation will measure the extent to which the technique presented in this work possesses those desirable properties. The desirable properties include the automation, operation, application to large transformations, adaptability and validity of the technique. By demonstration and discussion, the current evaluation intends to show the extent to which the desirable properties are present in the current technique.

The presented technique intends to support the validation of model transformation by creation of model generators. A further measure of the success of evaluation is the extent to which the technique supports validation. When applied to model transformation, generated models can both demonstrate situations where the transformation works and possibly detects situations where the transformation is erroneous. Error detection using the technique may not be possible; model transformations may contain errors not detectable by the technique. However, the extent to which validation is enabled by the technique is set as one of the criterion to measure the success of the evaluation.

The current technique for creation of model generators must be evaluated to determine the utility and applicability to validation. The evaluation by case study will demonstrate how

the technique is used in practice. The desirable properties of techniques to support validation are known (from chapter 3) and are set as criteria to measure the success of the evaluation. The extent to which the technique supports in validation is also a criterion to determine the outcome evaluation. By evaluation, it is possible to determine the utility of the presented technique.

6.3 Rational for Chosen Case Studies

This section gives an overview of the demonstrative and comparative case study undertaken. The comparative case study discusses the relativity of the proposed technique when compared to the leading similar methods as found in chapter 3. The demonstrative case study involves creating a model generator for three separate model transformations- Families2Persons, Tree2List and EMF2Fur; and applying the model generator to validation. In these case studies, the generator is created by an implementation of the method that was described in chapter 4. No additional constraints are added to the meta models to guide the model generation and the default scope of up to 10 of each meta element per generated model is used.

The rational of the case study is two-fold: to demonstrate the mechanism of the technique in practice; to show what kinds of transformation and validation are accepted by the technique. To demonstrate the technique in practice the following are criteria for model transformations: three distinct model transformations are applied to the technique. To demonstrate what kinds of transformation and validation a range of model transformation and meta model complexities along with different algorithms used in translation and different transformation approaches.

The Families2Persons transformation (pages 133–141) is chosen due to the relatively simple transformation, with two rules. The Families meta model has two classes and four associations and the Persons meta model has three classes and no associations. Families2Persons uses no specific algorithm other than the rule scheduler of the transformation engine. The Tree2List

features a slightly more complex translation with only one rule, but two similar meta models each with three classes and one association. The Tree2List transformation implements a Depth First Search algorithm. The final transformation used in case study is EMF2Fur first described in chapter 5. The features that set EMF2Fur apart are the structurally complex meta model: ECore, with nine rules, and a visitor-pattern [63] based algorithm. The conversion is also at a different meta-level than in previous selected case studies; concerned with conversion of meta models. In EMF2Fur a fully imperative rule scheduler is used, along with SiTra, a different model transformation language than used by Tree2List and Families2Persons case studies.

Selecting three distinct transformations can demonstrate the mechanism of the technique in the context of three transformations. Furthermore, the simplest of the transformations in this case study, Families2Persons, is described in detail (on pages 133–141) to give the reader unfamiliar with model transformation a fully worked example. This transformation consists of two rules and is written in the ATLAS model transformation language: the rules of the transformation simply copy data from a source to a destination model, relying on the element matching algorithm of the ATLAS transformation engine. The second transformation in this case study, Trees2List, is slightly more complex in that it has only one rule but that rule implements the Depth First Search algorithm to traverse the input model. The time complexity of the Trees2List transformation is therefore $O(|R| + |E|)$, where R and E are the relationships and elements of the input model, respectively.

The final part of the evaluation is a comparison to state-of-the-art related techniques as discovered in chapter 3. The techniques used are those of Ehrig et al. [56], Baudry [27] and Fiorentini et al. [58]. Empirical evaluation against those techniques is not possible as implementations are not readily available. Instead, the features of each state-of-the-art technique is compared against the features of the current presented technique. Comparison aims to uncover the similarities between those techniques and the current one.

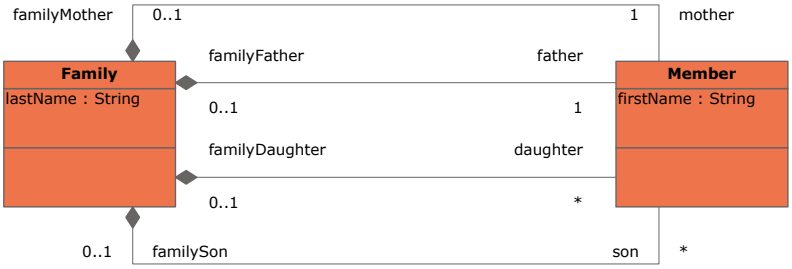
6.4 Case Study One: Families2Persons Transformation

Description of the Families2Persons Transformation

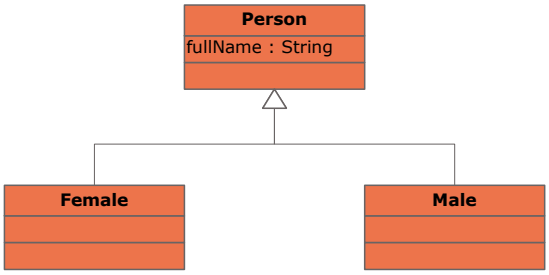
In the current case study, a model generator is created to support the validation of the Families2Persons model transformation, using the previously presented technique. The transformation is described as an introductory or "hello-world" [35] example in the ATLAS model transformation framework [72]. The transformation in the case study is created by the developers of the ATLAS framework, to teach the notation and application of the ATLAS model transformation language. The transformation is created to be purposefully simplistic, converting a model of related people (family) to a model of individuals. The primary aim of the current study is to demonstrate the application of the technique to support the validation of a previously unseen model transformation. The transformation can be found in [20, 21].

In the validation of this relatively "trivial" model transformation, interesting properties of the technique are demonstrated. By case study a demonstration is made of how the technique automatically creates a model generator for the families meta model, applied to the validation Families2Persons model transformation. The model generator automatically creates models to validate cases where the transformation works as expected. The operational and black-box nature of the presented validation is explained. Furthermore, the adaptability of the created model generator to changes in Families2Persons transformation is presented. Finally, a previously unknown error is detected in the Families2Persons model transformation, by models generated using the technique.

The same Families2Persons model transformation has recently been analysed in a previous, unrelated case study [98]. In case study, the Families2Persons transformation is used to illustrate the creation of white-box test cases for the validation of model transformation. White-box test-case creation uses the transformation definition as a basis for the test cases. In that work, criteria are proposed for the coverage of model transformations, specifically in the ATLAS model transformation language. No errors are detected in the Families2Persons model transformation by the work of [98].



(a) Families meta model.



(b) Persons meta model.

Figure 6.1: Families2Persons meta models.

The Families2Persons model transformation consists of three artefacts, two meta models Families, Persons and the Families2Persons transformation definition in ATLAS notation. The source meta model in ECore form Families, defines a simple modelling formalism for represented a group of relatives. In the family meta model, shown in figure 6.1a, a family has a name and possibly some members. Each family member is contained in a family and the member can be related to a family by the father, mother, sons or daughters relationships. By the multiplicity constraints, one family can only have up to one father and one mother and any number of either sons or daughters. Each Member must belong to only one Families, and cannot exist without that family. The target meta model Persons, shown in figure 6.1b, represents only individual persons, without any relationships and with only a full name. Each person in the Persons meta model can either be a Male or Female person.

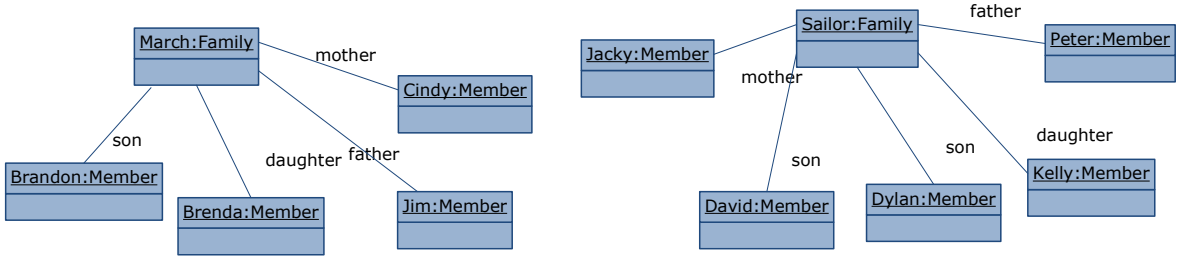
The model transformation Families2Persons, shown in figure 6.2, defines the conversion of any model conforming to the Families meta modelling formalism, to a model conforming to the Persons meta modelling formalism. That is, when a model of a family, with a father, mother, sons and daughters is given to the transformation, an equivalent person for each family member is created in the target formalism. In effect, the transformation creates a

```
module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: familyName : String = ...

helper context Families!Member def: isFemale() : Boolean = ...

rule Member2Male {
        from
                s : Families!Member (not s.isFemale())
        to
                t : Persons!Male (
                        fullName <- s.firstName + ' ' + s.familyName
                )
}

rule Member2Female {
        from
                s : Families!Member (s.isFemale())
        to
                t : Persons!Female (
                        fullName <- s.firstName + ' ' + s.familyName
                )
}
```

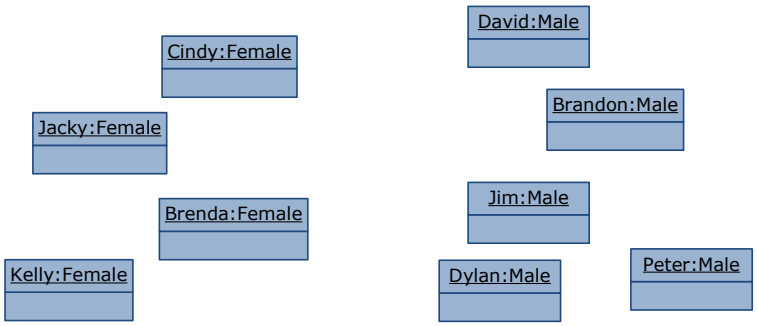
Figure 6.2: Families2Persons model transformation in ATLAS.

Male or Female person, based on whether the person is a Son/Father or Daughter/Mother. Each person is also created with a full name, based on the first and family name. The Families2Persons model transformation is defined by two rules and two helper methods.

In the Families2Persons transformation, the rule Member2Male defines how a son or father family member are used to create a Male person. The rule Member2Female similarly defines how a daughter or mother family member is converted to create a Female person. Both rules use the helper isFemale, to determine the gender of given Member. Gender is used to determine whether the Member2Female rule applies, in the case of mother or daughters, or the Member2Male rule applies, in the case of father or sons of a family. The Member2Male and Member2Female rules also use the helper familyName, to determine the family name of a given Member, in creating the full name of a Persons. The human created definition of the transformation, in a .atl file (shown in figure 6.2), is complied into an executable model



(a) Families sample model, provided with transformation.



(b) Persons model produced by application of transformation to the model in figure 6.3a.

Figure 6.3: Families2Persons provided sample family model and equivalent, created by transformation application.

transformation artefact, in a .asm file, that can be interpreted and applied by the ATLAS virtual machine.

The transformation comes with an example Families model, the example model contains two families; the family with the last name March and the family with the last name Sailor, shown in figure 6.3a. Each family has several members, and each member has a first name and a role in the family. When converted by the Families2Persons transformation, each member of both families is used to create an equivalent person. Female persons are created from each daughter or mother family member and Male persons for each son family-member. Applying the given example model to the transformation produces a list of persons, based on the input model, as shown in figure 6.3a. In such an application, the father named Jim in the March family model is used to create a male person, Jim March. The daughter Kelly of the Sailor family is used to create a Female person named Kelly Sailor, and so on. See figure 6.3a.

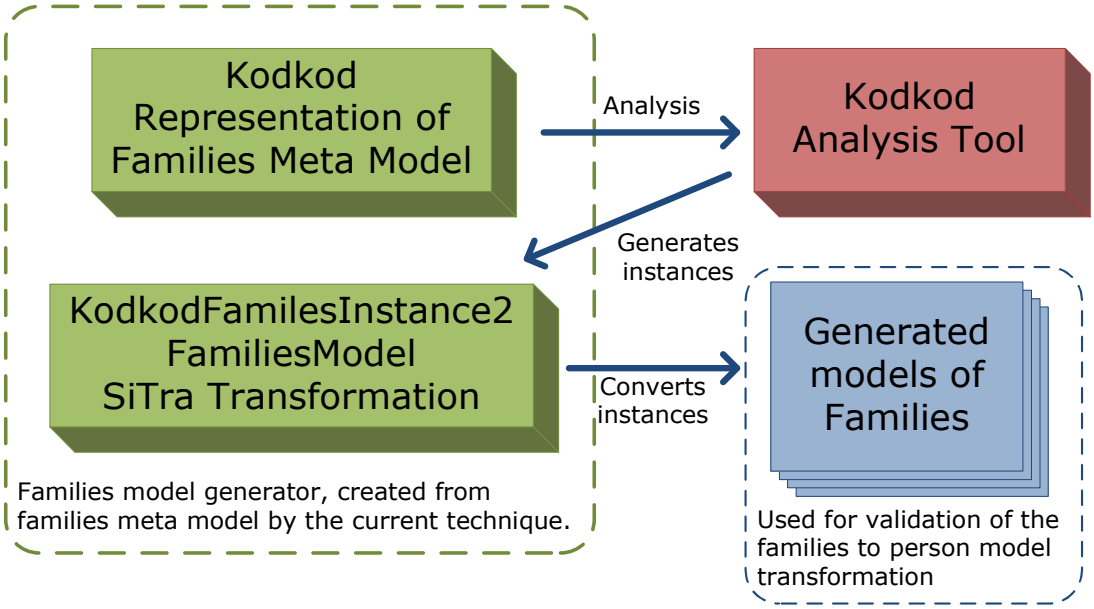
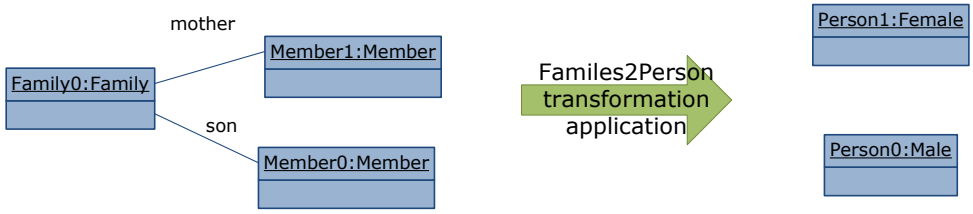


Figure 6.4: Model generator created for the Families meta model for use in validation.

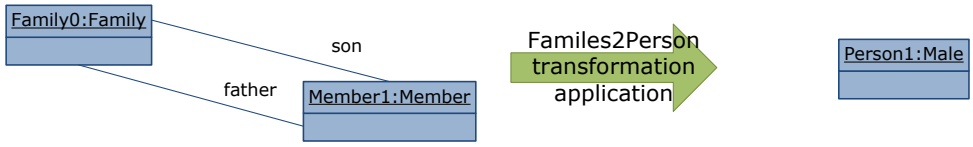
Validation of the Families2Persons Transformation

In the current evaluation, the technique for creating model generators is applied for validation of the Families2Persons model transformation. By the technique, a model generator is created for the Families meta model. The model generator created by the technique generates models of families, that conform to the Families meta model. The Families meta model is treated as the specification of the input to the Families2Persons model transformation. The family models generated are then applied to the Families2Persons model transformation, for validation. The model generator produces a sequence of models. A test harness is created to take generated models and apply them to the Families2Persons transformation, the creation of test harnesses is described in 5.7. By describing application of the technique to the Families2Persons transformation, the creation and application of model generators is demonstrated.

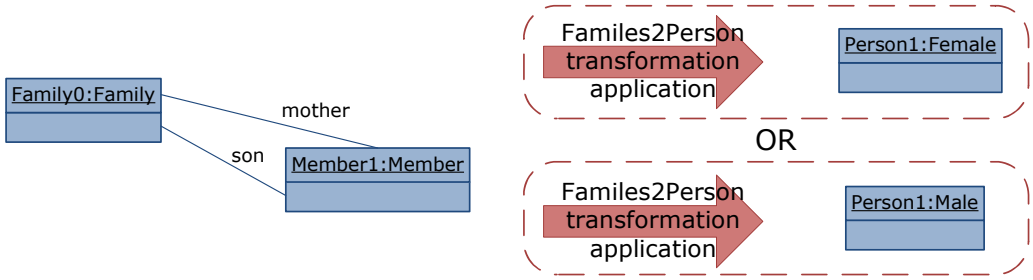
A model generator is created for the Families meta model, consisting of an equivalent Kodkod representation of the meta model and instance converter. The Kodkod representation of the Families meta model produces instances that conform to the Kodkod representation. A model transformation is also created, to convert Kodkod generated instances into models



(a) Generated model and equivalent: working situation.



(b) Generated model and equivalent: unexpected situation.



(c) Generated model and equivalent: error-causing situation.

Figure 6.5: Generated models and application to validation.

of the Families meta model. The created Kodkod representation of the Families meta model and the Kodkod families instance converter together form a model generator for the original Families meta model, as shown in figure 6.4.

The created model generator automatically creates models for validation of the Families2Persons transformation. The models created are valid instances of the Families meta model. A given generated model, when applied to the Families2Persons model transformation can demonstrate situations where transformation works as expected. For example, a model generated by the technique has families and members that are mothers, fathers, sons and daughters in each family, as shown in figure 6.5a. The model generator creates a sequence of valid models within the scope. Creating such models manually is time consuming and tedious, but also prone to error. The technique is able to automate the creation of a model generator; the created models demonstrate situations that work for the Families2Persons transformation.

The model generator for the Families meta model can create a sequence of models within the scope of analysis. The generated models are created using the meta model as a specifica-

tion, without any regard to the definition of the Families2Persons model transformation. The created model generator makes no assumptions about the workings of the transformation, the model transformation is treated as a black-box by the generator. The model generator creates models that apply to the operational code of the complied, executable Families2Persons model transformation.

In terms of complexity, the created families model generator consist of a Kodkod instance generator of approximately 450 lines of code in Java/Kodkod. The Kodkod instance generator takes approximately 600ms to produce the first instance and uses approximately 25 megabytes of memory. The model transformation converting the Kodkod instances to ECore models consists of 6 separate rules, with an average length of 80 lines of code each in Java/SiTra. The transformation takes 49ms on average to convert an instance created by the Kodkod instance generator, taking 185 megabytes of memory [1,2].

The model generator created to validate the Families2Persons model transformation purposely generates models without regard to the meaning. Automatically generated models can validate the applicability of a transformation to situations that may not have been considered by the transformation developer. Due to the families meta model relationships multiplicities, it is possible that a Member has a dual role in a Families. For example, a family model is generated where a Member is both a Father and Son, as shown in figure 6.5b. In the conversion of this model, a single Persons is created, as expected. When applied to the transformation, the models generated without regards to meaning are useful for validating the assumptions of a model transformation developer.

Automatically generated models can uncover errors in a model transformation. Another model is created by the generator, where a single Member is a mother and, a son in the same family, as shown in figure 6.5c. The developer of the Families2Persons model transformation did not consider this valid model in the Families modelling formalism. In the conversion of the error-causing model, a single Persons is created. However, the person is either Male or

[1]High memory utilisation is due to the I/O overhead of the Eclipse standalone environment.

[2]All experimental memory and time figures are averaged over three executions and rounded to the nearest unit. Experiments were conducted on an Intel Centrino Duo 7200, dual core 1800mhz with 1GB RAM, 32 bit Sun Java virtual machine version 1.6.0 and the 2.6.35 series of Linux kernels.

Female, depending on the order rule applications. The transformation is not confluent for the error-causing model. An error is detected in the transformation via a generated model, by the discovery of an unconsidered case in the model transformation.

For each generated model applied to validation, an oracle of the validation must be set. The oracle determines the success, or otherwise of the validation, particularly important in the current technique as meaningless models are generated. In the current case, four specific oracles are applied to judge the success of the transformation. When a generated Families model is applied to the Families2Persons transformation, the created Persons model must be a valid instance of the Persons meta model. The transformation is also communicative, models applied to Families2Persons transformation should produce a model approximately equivalent in size. The models applied to the transformation must be converted consistently, when applied multiple times (confluence). The confluence oracle is used to detect the error in the Families2Persons model transformation. The generic oracles used here were previously discussed in chapter 4.

The presented technique for creating model generators is adaptable to changes and developments of the Families2Persons model transformation. Once an error is detected in the Families2Persons model transformation by the generated models, the error must be corrected. In general, such a detected error may be corrected in several ways. Either the model transformation definition or the meta model is modified to correct the error in the Families2Persons transformation. If the model transformation definition is changed or the Persons meta model is changed, then the previously created model generator still applies to the validation of the changed Families2Persons transformation. In the case that the Families meta model is changed to rectify the error, the technique can be re-applied to create a new model generator. The technique for creating model generators for validation of the Families2Persons model transformation, is adaptable to changes in the model transformation or meta models.

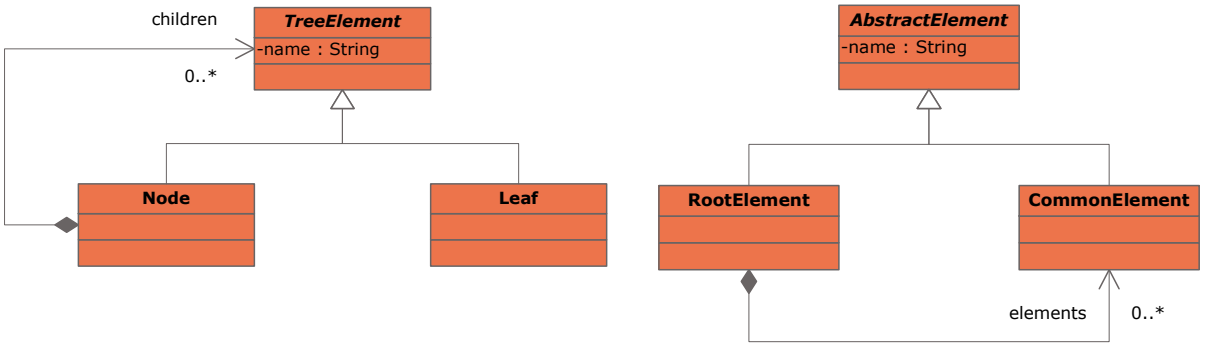
In this section, a case study of the presented technique is carried out, by the application to validation of the Families2Persons model transformation. A model generator is automatically created for the Families meta model and the models generated are applied to the validation of

the Families2Persons model transformation. The models generated demonstrate the situations where the transformation works and a situation where the transformation is erroneous. The validation using generated models is black-box and applies to the executable artefacts of the transformation. The model generation technique is also adaptable to changes in the model transformation and can be re-applied where the input meta model changes. By a case study of the application to the Families2Persons model transformation, the advantageous properties of the technique are determined.

6.5 Case Study Two: Tree2List Transformation

In the following case study, a model generator is created to support the validation of the Tree2List model transformation, using the previously presented technique. The transformation in this case study is described as a more advanced and realistic example application of the ATLAS model transformation framework, used in tutorials of the ATLAS language. The transformation is created by the developers of the ATLAS framework, to teach advanced features of the formalism. The transformation is created as a realistic example to convert a tree data structure to a model list data structure. The primary aim of the current study is to demonstrate the adaptability of the technique to support the validation of a distinct model transformation. Further details of the transformation are published in the ATLAS tutorial [20, 57].

In the validation this exemplary model transformation of the ATLAS transformation language, the properties of the current technique are demonstrated. By case study, a demonstration is made of how the technique automatically creates a black-box model generator for the tree meta model; the created model generator is applied to the validation of the Tree2List model transformation. Furthermore, by the case study it is demonstrated that the technique is straightforwardly re-applied to the validation of a distinct model transformation, then used in the previous case study. Finally, by the models generated using the technique, a previously unknown error is detected in the Tree2List model transformation causing non-termination of



(a) MMTree meta model.

(b) MMElementList meta model.

```
module Tree2List;
create elmList : MMElementList from aTree : MMTree;

uses Lib4MMTree;

rule TreeNodeRoot2RootElement {
        from
                rt : MMTree!Node (rt.isTreeNodeRoot())
        to
                lstRt : MMElementList!RootElement (
                        name <- rt.name,
                        elements <- elmLst
                ),
                elmLst : distinct MMElementList!CommonElement
                        foreach(leaf in rt.getLeavesInOrder())(
                                name <- leaf.name
                        )
}
```

(c) Tree2List model transformation definition in ATLAS.

Figure 6.6: Tree2List model transformation and meta models.

the transformation.

The Tree2List model transformation has recently been analysed in a previous, unrelated case study [39]. In case study, the Tree2List transformation is used to illustrate a technique for the formal verification of model transformations. Formal verification of the Tree2List transformation involves the manual conversion of the transformation definition to the Coq analysis formalism. Using the Coq tool support, certain properties of the model transformation may be interactively proved. By the case study in [39], no errors are detected in the Tree2List transformation by the application of the verification technique.

The Tree2List model transformation serves to demonstrate concepts from the ATLAS transformation notation. The transformation consists of the tree meta model MMTree (figure 6.6a), the list meta model MMElementList (figure 6.6b), the definition of the transformation Tree2List (figure 6.6c) and a library of re-usable functions over trees in the Lib4MMTree. The transformation defines the conversion of a tree model to a list model. The list model created by transformation from a tree model is a depth-first traversal of the tree, to produce an ordered list from only the leaf elements of the tree model. The transformation also comes with an example tree model that is used to demonstrate the transformation. The Tree2List transformation is created to demonstrate features of the ATLAS transformation language.

By the implementation of the technique, a model generator is created automatically, for creating models of the tree meta model. The technique does not inspect the transformation definition and treats the Tree2List transformation as a black box. The model generator is created based on the trees meta model, to generate models automatically. The generator for the trees meta model produces either a specified number of models or continuously produces models conforming the tree meta model. When applied to the transformation the generated models validate and demonstrate situations where the transformation Tree2List transformation works as expected.

In terms of complexity, the created tree model generator consist of a Kodkod instance generator of approximately 250 lines of code in Java/Kodkod. The Kodkod instance generator takes approximately 1200ms to produce the first instance and uses approximately 37 megabytes of memory. The model transformation converting the Kodkod instances to ECore models consists of 6 separate rules, with an average length of 120 lines of code each in Java/SiTra. The transformation takes 62ms on average to convert an instance created by the Kodkod instance generator, taking 205 megabytes of memory [1,2].

[1]High memory utilisation is due to the I/O overhead of the Eclipse standalone environment.

[2]All experimental memory and time figures are averaged over three executions and rounded to the nearest unit. Experiments were conducted on an Intel Centrino Duo 7200, dual core 1800mhz with 1GB RAM, 32 bit Sun Java virtual machine version 1.6.0 and the 2.6.35 series of Linux kernels.

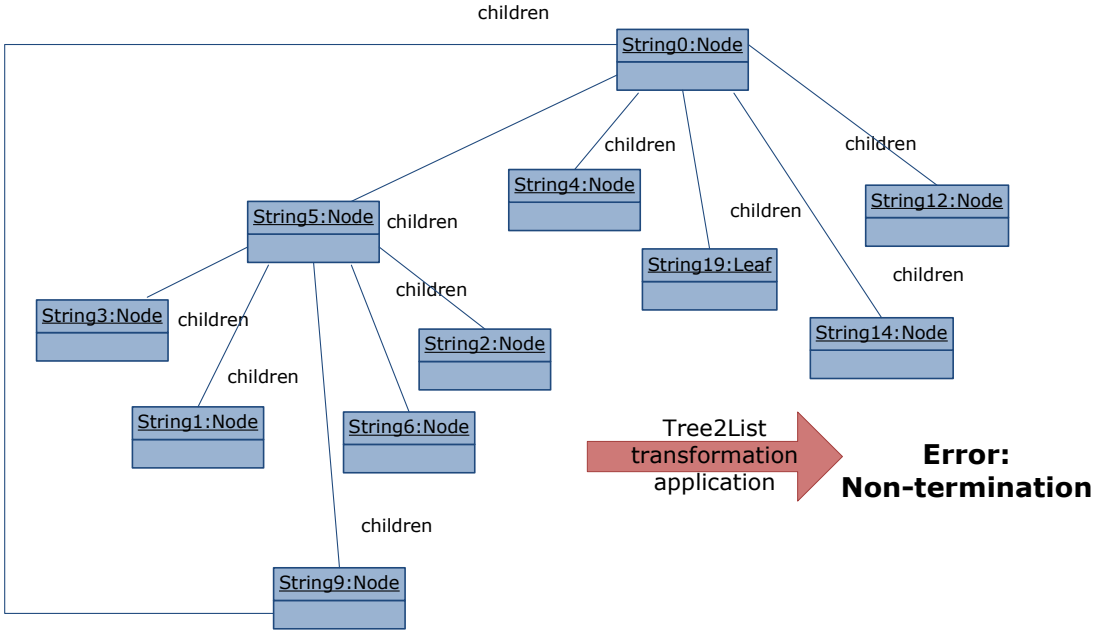


Figure 6.7: Generated model : error-causing situation.

6.5.1 Apparent Error in Tree2List Transformation

The models generated also detect an error-causing scenario where the transformation does not work as expected. Any model of the tree meta model must apply to the Tree2List model transformation. The model generator creates models, using the trees meta model as the specification of the tree modelling notation. A model is produced by the model generator that highlights an error.

The error-detecting model describes a structure with ten nodes and a single leaf element, shown in figure 6.7. Note that one of the descendent nodes of the root element, “String9” has the root as a child. So the data structured described in the model is not a tree. When applied to the Tree2List transformation, the generated model causes an infinite execution of the transformation. This infinite loop is clearly an error, as the Tree2List transformation developer assumes that only tree models will be used. However, the ECore implementation of composition allows for the creation and storage of models with cycles in the composite relationships. Indeed, when passed to the ECore validation framework the “invalid” model is validated as conforming to the tree meta model. The detected error is found without inspection of the internal working of the Tree2List transformation.

The reason such a model is generated, even though apparently incorrect, by the model generator is the lack of any constraint in the ECore meta model preventing such constructs. The ECore meta meta model does not include any restriction on a composite elements from having itself as a child. Thus, when a model is converted to Kodkod, Kodkod can generate models with composite relationships that contain cycles. ECore also allows such models to be stored, loaded and even validated against a meta model without error [1]. In order to correct this oversight, the implementation could be modified to include such constraints on the model generation, where the constraints are made available.

When performing software validation, the expected outcome must be set to determine the success or otherwise of the validation. In the presented error causing model, there are three relevant oracles to determine the outcome of the validation. The lack of any model produced as output from the transformation is used to indicate a possible error in the transformation. The (non-)termination of the model transformation within a pre-determined time is also used to determine there is a possible error in the transformation. The generic oracles used here were previously discussed in chapter 4.

As in the previous case study, this error can be corrected by the transformation developer using a range of strategies. For example, additional constraints can be added to the trees meta model to prevent graph-structures. Alternatively, the traversal algorithm can be modified to detect and reject application non-tree structures. However, in this case the error could be corrected by the ECore framework disallowing loops in composite relationships, as per the standard. Other error correction strategies are possible and must be selected by the transformation developer, taking into account the intended use of the transformation.

This section has presented a case study of the technique to create model generators, by application to the validation of the Tree2List model transformation. A model generator is created automatically, using the MMTree meta model as a specification of the tree modelling formalism. A demonstration is made on the applicability to a distinct transformation and meta model the was previously used. A sequence of valid models is generated by the genera-

[1]The process of model-to-meta-model validation in EMF is discussed in section 5.7.

tor, as valid instances of the Trees meta model and are applied as input to the Tree2List transformation. The generated models demonstrate situations where the transformation works as expected, as well as uncovering an erroneous situation, triggering the non-termination of the Tree2List transformation. The validation in the case study does not use knowledge of the internal working of the transformation, the error causing model was automatically generated from the meta model. The properties of the presented technique are demonstrated by the case study using the Tree2List transformation.

6.6 Case Study Three: EMF2Fur Model Transformation Self-Validation

A novel feature of the presented technique is the ability to create a meta model generator for the ECore meta meta model. In the following case study, a model generator is created to support the validation of the EMF2Fur model transformation used in the implementation of the current technique. The EMF2Fur transformation in this case study uses more complex meta models than in previous studies. The transformation has previously been described in detail in chapter 5 and is developed using the SiTra model transformation framework; significantly different to the ATLAS framework of the previous studies. The aim of the current study is to demonstrate the application to a larger-scale transformation and the adaptability of the technique to validate a distinct model transformation, in a distinct model transformation language. Further details of the EMF2Fur transformation can be found in chapter 5.

In the self-validation of the EMF2Fur model transformation, the properties of the current technique are demonstrated. A demonstration is made of how the technique automatically creates a black-box model generator for the ECore meta meta model; the created model generator is applied to the validation of the situations where EMF2Fur model transformation works as anticipated. Furthermore, by the case study it is demonstrated that the technique is applicable to validation of a distinct model transformation and transformation language, then used previously.

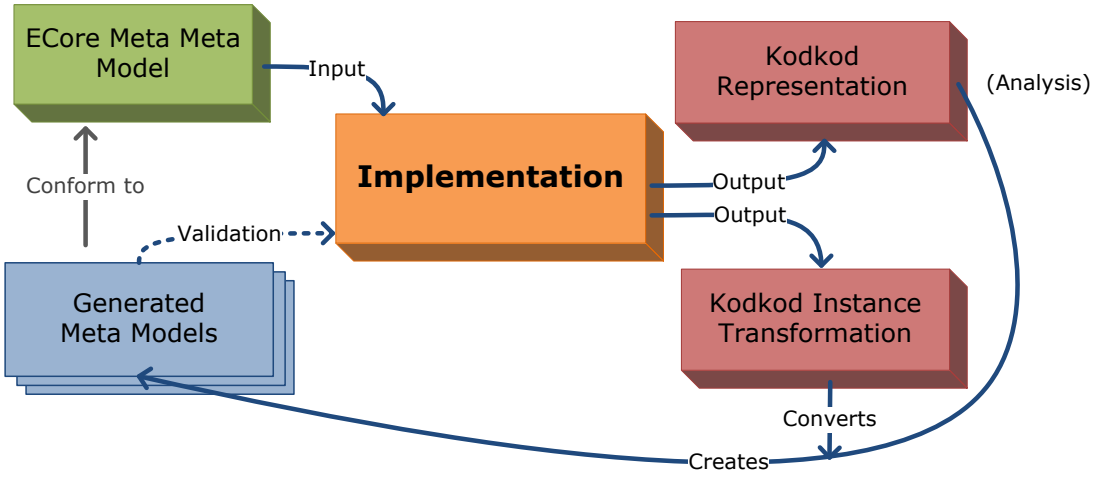


Figure 6.8: Meta model generator created for the ECore meta meta model and application to self-validation.

In the current case study, the EMF2Fur model transformation uses the ECore meta meta model as the input. The model generator produced by the technique in this case produces ECore compliant meta models. In the current application of the technique, the generated meta models may not specify any meaningful modelling formalism. However, the generated ECore meta models - as any other valid ECore meta models - must be applicable to the implementation itself. The generated meta models are applied to the validation of the implementation. In the validation, the model transformations in the implementation will use create a model generator for the generated meta models. The model generators created in this validation scenarios should create models that conform the generated meta models that where passed in to the implementation for validation. When applied to the self-validation of the implementation, as shown in figure 6.8. The generated meta models are used to demonstrate situations where the implementation works as expected.

The EMF2Fur model transformation is part of a chain of four model transformations, that together form the implementation of the technique, creating model generators from meta models. The first transformation EMF2Fur, produces a Fur model, and trace. The Fur meta model is passed to a second model to text transformation to create a Kodkod equivalent. The trace of the EMF2Fur used to create a Kodkod instance converter by a third, model-to-model transformation, followed by a fourth model-to-text transformation. This higher order

transformation is used to create a SiTra model transformation. By applying the generated ECore meta models to the validation of the EMF2Fur model transformation, the other model transformations in the chain of transformations that constitute the current implementation are triggered. The transformations that follow the EMF2Fur in the implementation are therefore also validated by the application of generated ECore meta models.

The extended ECore meta model used in this example is much larger than those used in the previous case studies. The full ECore meta meta model has fourteen meta elements and twelve relationships between meta elements, as shown in figure 6.9. The model transformation under validation implements nine rules, so is also larger than in the previous case studies. The meta models generated by the technique are valid instances of the ECore meta meta model, by a created model generator for the ECore meta meta model. The ECore meta models generated for use in validation are produced independently of the SiTra EMF2Fur model transformation definition, making validation by the generated meta models black-box. The models are also applied directly to the operational artefact of the EMF2Fur transformation, validating the execution of the transformation. The size and complexity of the ECore meta meta model and the EMF2Fur transformation demonstrate the application of the technique to a larger model transformation and meta model.

The overarching impact of the self-validation of the EMF2Fur transformation is to aid development of that transformation. In this scenario, EMF2Fur is applied to the ECore meta meta model, to create a meta model generator. The generated meta models are then self-applied to the EMF2Fur transformation for validation. In the evaluation of the EMF2Fur transformation, two oracles are used: the manual inspection of the output and the by-product inspection oracle. The main effect is the detection of elements of the ECore meta model not supported by the EMF2Fur transformation. Generated models may use features that are not supported by the EMF2Fur transformation. For example, the ECore meta model (figure 6.9) uses ERelations to denote a sub-class of a given EClass, so the meta models generated for validation can contain an inheritance hierarchy. When such generated models with inheritance are applied to EMF2Fur, by incepting the created Kodkod code, it becomes clear that the

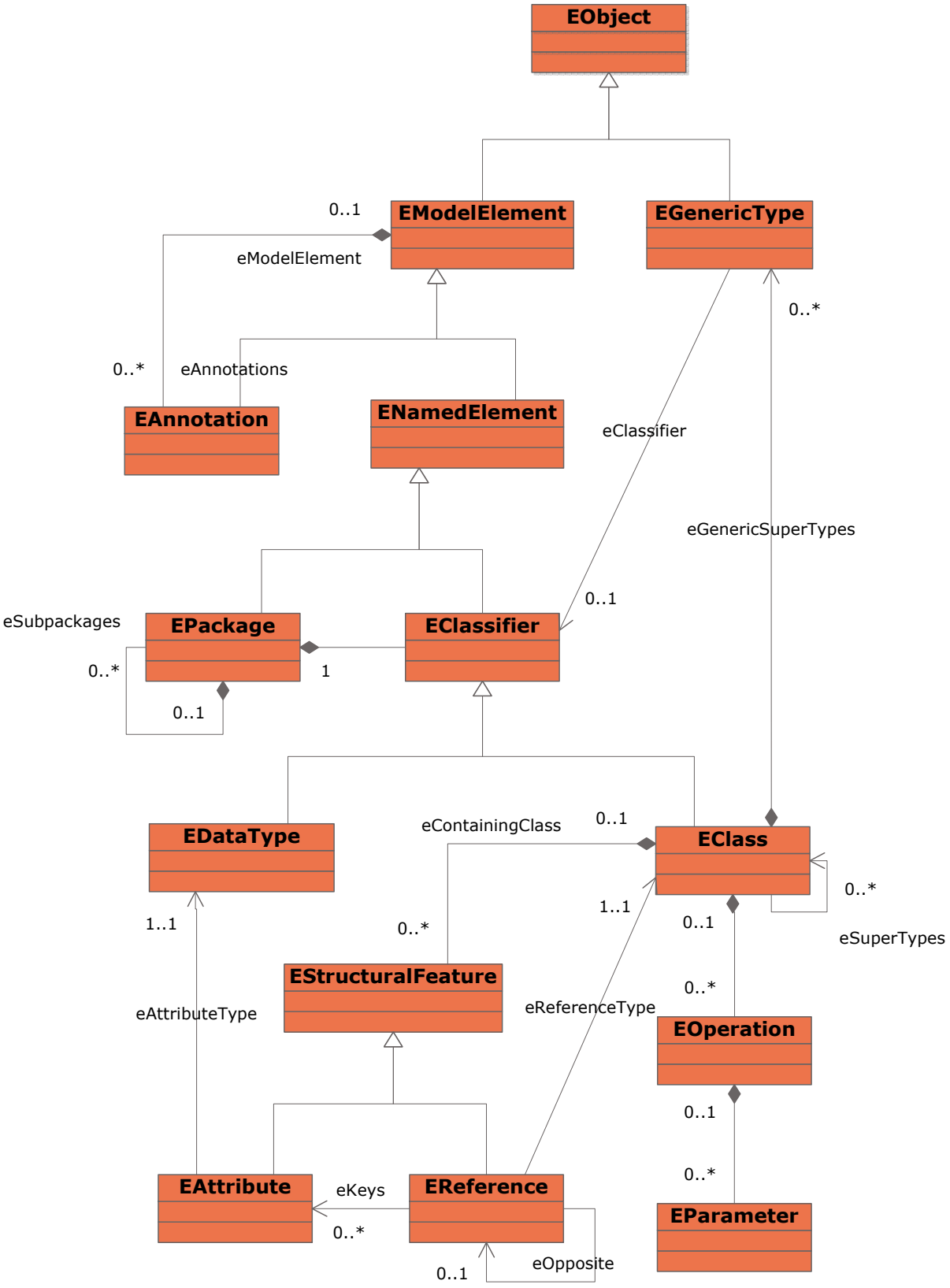


Figure 6.9: Extended ECore meta meta model.

inheritance hierarchy has not been converted and this feature of the transformation requires development. As each feature and rule of the transformation is developed, more elements of the generated meta models are converted by EMF2Fur and demonstrated as being supported. In effect, this validation is used to support the test driven development [28] of the EMF2Fur transformation, similar to [136].

In terms of complexity, the created Ecore meta model generator consist of a Kodkod instance generator of approximately 3000 lines of code in Java/Kodkod. The Kodkod instance generator takes approximately 4500ms to produce the first instance and uses approximately 55 megabytes of memory. The model transformation converting the Kodkod instances to ECore models consists of 35 separate rules, with an average length of 125 lines of code each in Java/SiTra. The transformation takes 73ms on average to convert an instance created by the Kodkod instance generator, taking 230 megabytes of memory [1,2].

In this case study, an implementation of the presented technique is validated. The validation involves the application of the implementation to the ECore meta meta model to the implementation, to create a meta model generator. The created meta model generator for ECore is applicable to the validation of the EMF2Fur model transformation, and the transformations that constitute the implementation of the technique. The model generator is created automatically and without regards to the internal workings of the EMF2Fur model transformation. The EMF2Fur model transformation uses a distinct meta model and model transformation framework than in the previous case studies, demonstrating the wide-applicability of the technique. Furthermore, the ECore meta model and EMF2Fur model transformation are larger and more complex than in the previous case studies, demonstrating the ability of the model generation technique to handle larger transformations. In the case study self-validation of the implementation is also demonstrated. The advantageous properties of the technique are presented in the EMF2Fur case study.

[1]High memory utilisation is due to the I/O overhead of the Eclipse standalone environment.

[2]All experimental memory and time figures are averaged over three executions and rounded to the nearest unit. Experiments were conducted on an Intel Centrino Duo 7200, dual core 1800mhz with 1GB RAM, 32 bit Sun Java virtual machine version 1.6.0 and the 2.6.35 series of Linux kernels.

6.7 Comparison to Existing Model Generation Technique for Transformation Validation

In this section, the technique is evaluated via a comparison to existing techniques. Three state-of-the-art techniques propose the generation of models from meta models, for validation. In Fiorentini et al. [58], a given meta model converted to Prolog to generate instances. In Ehrig et al. [56] a meta model is used to derive a generative graph grammar; an algorithm is applied to the created grammar to generate models. In Baudry [27], an ECore meta model is converted automatically converted to Alloy, to generate instances. A common theme in each is the treatment of the meta model as a specification, the generated instances as models in the modelling formalism and the application of generated models to validation. The current technique is unique in the automation of conversion and analysis, self-validation, black-box error detection and application to multiple transformations both large and small. The following study demonstrates how the current technique is unique; possessing a unique selection of properties, when compared to the state-of-the art.

In the existing techniques to generate instance models, the focus is on the automated analysis for the creation of instances. In Fiorentini et al. [58], the automated generation is done using a re-representation of a meta model using Prolog. Prolog is a declarative language similar to first-order-logic, and is applied in the technique applying the logic Constructive Type Theory "CTT" (further details are available in Fiorentini et al. [58]), to generate instances. Manual, rule-of-thumb definition of constraints must be made in Prolog to avoid state-space explosion. In the work of Ehrig et al. [56], a meta model is encoded as generative graph grammar, where instances can be automatically simulated. The graph grammar is limited in that meta models with loops, constraints or complex multiplicities can not be applied to the algorithm. In Baudry [27], the meta model is re-represented in Alloy, along with constraints and multiplicity; instance creation is done automatically via the Alloy tool. Alloy tools employ SAT solvers via Kodkod to perform analysis and instance creation. The current presented technique is similar to Baudry [27], however, using the Kodkod notation directly.

For model generation, the current technique employs automated model transformation

conversion of meta models to, and conversion of instances from the Kodkod formalism. The existing techniques also employ conversion, but use manual conversion [58], or use an algorithm for the conversion [56, 27]. These conversions and algorithms may be prone to developer error in implementation. The techniques of [56, 27, 58] do not validate the algorithms or conversions used. Model transformation is an ideal candidate for such conversions and the current proposed technique employs model transformation. A side effect is an implementation of the proposed technique can be applied to create meta model generators for use in self-validation.

The state-of-the-art techniques convert meta models to a distinct analysis formalism, but do not discuss the conversion of instances generated in the analysis formalism. Conversion of generated instances may be done manually [58] or automatically [58, 27] in existing techniques, but such conversion is specific to a particular meta model, as found in chapter 5. Furthermore, validation of the instance conversion is not considered in existing techniques. In the current technique, the conversion of instances is done automatically for each unique meta model, using a generated model transformation. The conversion of meta model and instances is also validated in the current approach, by the self-validation of the implementation.

The current technique allows for black-box validation of model transformation, by creating instance generators without regards to the transformation definition. In [27], the model transformation definition must be analysed to determine the effective meta model, making the validation white-box. Also in [27], systematic validation is possible by manually defined constraints on the model generation. Previously unknown errors have not been detected using the techniques in [56, 27]. The method of [56] is analysed by a proof that the instance generation algorithm will terminate, but only applies to simple meta models. In [27], the method is evaluated by mutation analysis, the ability to detect known errors. Previously known transformation errors are inserted manually introducing errors into a transformation. No examples of such errors or transformations are given. In Fiorentini et al. [58], an error is detected by the technique, by creating specific test data based on pre- and post- conditions.

In the current technique, errors can be detected in a purely black box way as demonstrated in the case studies.

The current technique applies to creating model generators for the validation of large, complex meta models and has been applied to the validation of three different model transformations. In [27], the model transformation must be analysed to determine the effective meta model, to reduce the state space for analysis in Alloy. Alloy is less efficient at the analysis of Alloy models, then where Kodkod is used directly for the same models [141]. In Fiorentini et al. [58], rule-of-thumb expertise must be used to avoid state-space explosion problems. In case study, the technique applied to the validation of a single model transformation [58]. In [56], the constraints or multiplicity of a meta model are not considered, neither are meta models with loops. The techniques of [56, 27] are proposed for validation but not applied in case study. In the current technique, three distinct model transformations, one of those using the full ECore meta model and a distinct model transformation framework are validated. This demonstrates the ability of the technique to apply to large transformations and distinct transformations.

By a comparative study, the relationship between the current technique and the state-of-the-art techniques from literature is found. The technique of Fiorentini et al. [58] performs automated analysis by an analysis formalism but employs manual conversion to that formalism; unlike the automated conversion in the current technique. The techniques of [56, 27, 58] all employ meta model and model conversion, but do not consider the validity of the conversions used. In the current technique, an implementation has been applied to self-validation. Techniques of [56, 27] are not applied to actual model transformation validation. Notably, the technique of Fiorentini et al. [58] is applied to detect an error in single model transformation. The presented technique is applied to the validation of three distinct transformations, in two model transformation languages, detecting errors in two model transformations. By the above comparison, the current technique relates favourably to existing state-of-the-art model transformation generation techniques.

6.8 Updated Classification of Model Transformation Dependability Evaluation Techniques

In this section, the table 3.1 is re-created in table 6.1 to include the current work and only the closely related techniques of [56, 27, 58]. The table shows a summary of the discussion in section 6.7, where the full analysis and comparison can be found. Note that the feature comparison in the table is done using results in the respective published articles; empirical evaluation is not possible due to unavailability of implementations of each technique. An explanation of the headings can be found with the original table in section 3.6.

Technique	*Analysis Automation*			*Analysis Artefact*		*Other Analysed Artefact(s)*			*Mechanism*					*Level*		*Oracle*					*Computability*			*Validity*			Errors Detected
	Automated	Semi-automated	Conceptual	Specification	Code	Meta-model	Instances	Annotations	Conversion (auto.)	Conversion (man.)	In-place Interpret.	Static	Dynamic	White-box	Black-box	Model	MM Conformance	Model Properties	Generic Criteria	Manual	Bounded	Manual guidance	Complete	Proof	Semi-formal	Example	
Fiorentini et al. [58]	●	○	○	○	●	●	○	●	○	●	○	○	●	○	●	○	●	○	●	○	●	●	○	○	○	●	●
Ehrig et al. [56]	●	○	○	○	●	●	○	○	●	○	○	○	●	○	●	○	○	○	○	○	○	○	●	●	○	○	○
Baudry [27]	○	●	○	○	●	●	○	●	●	●	○	○	●	●	○	○	○	○	○	○	●	●	○	○	○	●	○
(This work)	○	●	○	○	●	●	○	○	●	○	○	○	●	○	●	○	●	○	●	○	●	○	○	○	○	●	●

Table 6.1: Features of selected model transformation quality evaluation techniques.

●: the feature is present. ○: the feature is not present.

6.9 Summary

This chapter has presented an evaluation of the technique by three case studies and comparison to existing techniques. The case studies present how validation is enabled by the technique. It is demonstrated that model transformations can be validated by the model generators created by the technique. The studies involve the application of model generators to validation of Families2Persons, Tree2List and EMF2Fur model transformations.

The case studies demonstrate the advantageous properties of the technique. Model generators are automatically created from meta models to validate the operational, executable artefacts. Three distinct model transformations are validated: Families2Persons, Tree2List and EMF2Fur. The created model generators are purely black-box; without any need for inspection of the transformation definition. In all the case studies, the technique is shown to create model generators. The technique is adaptable to change in the meta model, where a new generator is created; or changes in the model transformation, where existing model generators still apply. As well as validation of situations where the transformation works as expected, examples are given of errors detected by application of the technique to the validation of Families2Persons and Tree2List transformations. The errors detected by the case study of the current technique are not found by previous, unrelated case studies of existing model transformation quality assurance techniques (in [98, 39]).

In the EMF2Fur transformation case study, the technique was shown to be applicable to a large and complex transformation and to model transformations in a significantly different model transformation formalism. The EMF2Fur case study shows how an implementation of the technique creates a meta model generator for the ECore meta meta model, that can be applied to the self-validation of that implementation.

A comparison has been presented between the proposed technique and existing state-of-the-art techniques to generate models for validation [56, 27, 58]. The technique has advantages over the method of [58], as the current technique uses automated conversion of meta models. The current technique considers the meta model constraints, complex multiplicities and meta models with circular definitions, unlike the technique of [56], which only

applies to simplified meta models. In comparison to the technique of [27], the current technique is advantageous by being fully black-box and using automated model transformation to convert meta models. The current technique has advantages over the existing techniques of [56, 27, 58]: by the applicability to self validation, the detection of errors in multiple distinct model transformations and the application to diverse model transformation languages.

In summary, the contributions of this chapter are as follows:

- Case study has applied model generation to assist the validation of the Families2Persons, Tree2List and EMF2Fur model transformation.
- A comparison of current techniques against existing techniques to generate models for model transformation validation, evaluating the relative merits.

In the current chapter, the technique has been evaluated by case study, discovery of properties and a comparison to the state-of-the-art. The following chapter summarises the outcomes of this work, along with direction for future research that may be used to improve the presented technique.

CHAPTER 7

CONCLUSION

Life is divided into three terms-
that which was, which is, and which will be.
Let us learn from the past to profit by the present,
and from the present to live better in the future.
— William Wordsworth

This thesis has presented a technique to generate models from meta models to support the evaluation of model transformation dependability. In model driven software development, models are the central artefact of development and the conversion of models directly informs the creation of software. Transformations are a key feature and advantage of model driven software development, allowing models to apply in more than one context. However, a model transformation with errors can transmit those errors to the software created by that transformation. A meta model is the definition of a modelling formalism, specifying the complex input of model transformations. The presented technique applies model transformation and software analysis tools to meta models, to create of model generators. An implementation of the technique is able to generate models that apply to assist the validation of model transformations.

Evaluating the dependability of developed software is a key challenge, particularly for ensuring that software tools used in the development of software. Dependability properties of software have be defined and classified as availability, reliability, safety, integrity, maintainability [22]. This work has been concerned with one aspect of dependability: reliability; specifically, dealing with pre-emptive detection of errors in model transformation.

Diverse techniques are proposed in literature to evaluate that a transformation will work as intended. A classification of model transformation dependability evaluation approaches is made in chapter 3. A key challenge is automating the processes that assist in evaluating transformation reliability. The objective of verification techniques is to analyse a model transformation for consistency with regards to some specified reliability properties. The objective of validation techniques is to demonstrate situations that work as expected, as well as possibly uncovering error causing situations. For verification, a specification of model transformation reliability properties is often-times, not available and must be manually created - an error prone process. Due to complexities of transformation, verification techniques for model transformations only apply to simplified or abstracted of model transformation. Furthermore, expertise and manual interaction in a particular software analysis formalism is often required for every transformation verified. Instead, validation is found to be a promising technique to

generate models to support automated transformation correctness evaluation.

Ensuring model transformations will work as anticipated is an important and difficult task, as found in chapter 3. A given model transformation is created once to convert any of the diverse models conforming to a modelling formalism. However, modelling formalisms and the model transformations are subject to change, for example if errors are found. To validate a transformation, models must be created and applied to the transformation; the outcome of the application determines the success of validation. Models of software conforming to a modelling formalism are created by the manual interaction of experts, a time consuming and error prone task in itself. A given model transformations can use any meta model to specify the input, the models that are converted by that transformation. Furthermore, many heterogeneous model transformation notations are available. To validate the wide range of available model transformations, it is desirable to generate models automatically to support the validation process.

The desirable properties of a model transformation validation technique are uncovered and presented in chapter 3, by an analysis of existing techniques that evaluate model transformation dependability. The automation of analyses that assist in the validation is essential due to the number, complexity and importance of model transformations in model driven software development. Analysis taking internal structure or working into account is specific to a certain model transformation formalism. Instead, the operational artefacts of a transformation must be validated (in a black-box way,) without regards to the internal structure or working of the transformation. Automation of the processes that assist validation allows the evaluation of a wide-range and numerous model transformations. Validation of transformations is also important as transformations can be used in techniques for model transformation dependability evaluation. It is of critical importance to demonstrate the transformations in those techniques are reliable, otherwise results of the evaluation may be invalid.

The proposed technique to create model generators from meta models is presented in chapter 4. By applying generated models to a transformation, situations that work as anticipated can be demonstrated, as well as possibly discovering situations that cause error in

a transformation. After models are applied to a model transformation for validation, the outcome of transformation is compared against generic expected outcomes. Development of software models is typically a manual task, requiring expertise in the modelling formalism involved. Models must be created to validate a given transformation. Ideally the process of creating models for validation is automated, however algorithmic creation of models is problematic. Meta models that define modelling formalisms are complex with arbitrary logical restrictions on the allowed models and with cycles that make algorithmic model derivation difficult. Random creation of models is also made difficult due to the logical constraints that arbitrarily restrict the validity of models in a modelling formalism. Generating models directly from a meta model necessary to assist the validation of model transformation.

Central to this work a technique to create a model generator from meta models and a multi-stage implementation in chapter 5. In practice, the technique to create model generators involves advanced model transformation concepts. Meta model transformation and model to text transformation is used to convert meta models to a software analysis formalism. The analysis formalism is typically used with tools to support in the bounded verification of structural abstractions of software, by exploiting bounded boolean satisfiability solvers. In the current technique the analysis of such techniques is used to generate instances from a representation of the meta model. The instances generated in the analysis formalism must be converted to become models that conform to the original meta model. For this task a model transformation is created by a higher order model transformation. The higher order transformation converts the trace of the meta model transformation to create an instance model transformation. The implementation is realised in practice and may be implemented in alternative ways, as found in chapter 5.

The application of the model generation technique to model transformation validation is evaluated and a comparison made to state-of-the-art model generation techniques in chapter 6. In the demonstration of the technique, model generators are automatically created by an implementation of the technique. Generated models are applied to support the validation of model transformations. The created model generators are shown to apply to the valida-

tion of a range of model transformations. Models are automatically generated and applied to validation, the outcome of validation is evaluated by six general oracles. The combination of generated models and general oracles have, in case study, discovered errors in two model transformations; that are used to demonstrate the prominent ATLAS model transformation formalism. The erroneous transformations have been previously validated [98], in the case of Families2Person, and verified [39], in the case of Tree2List, without the errors being detected. A recent technique for model transformation validation, independent from the current work, has been applied to detect errors in the Persons2Families model transformation [65]; a transformation in the opposite direction of Families2Persons that was used to evaluate the current work. In the combination of properties, the presented technique is unique compared to any of the state-of-the-art model generation techniques in literature [56, 118, 27].

Model transformations are used throughout in the presented technique to create model generators from meta models. A novelty of the technique is the ability to create a meta model generator. The generated meta models can be applied to technique, to assist in self-validation of the implementation. The ability to generate meta models and the application of generated meta models to self-validation is discussed in chapter 5 and demonstrated in chapter 6.

7.1 Summary of Contributions

This thesis has presented a method to create model generators to support the validation of model transformations. A summary of the contributions of this thesis is as follows:

- A comparative classification of model transformation dependability evaluation techniques is presented. Related techniques for verification and validation are analysed and compared. (Chapter 3.)
- By analysis of existing techniques a group of promising features is found for the basis of a novel, partly automated model transformation validation technique. The properties of the proposed validation techniques are used later in the evaluation. (Also chapter 3.)

- A technique to create model generators from model transformation is described, to support model transformation validation. Six generic oracles are also proposed to determine the success, or otherwise, of validation by automatically generated models. (Chapter 4.)

- The description of a practical implementation and possible alternate implementations is presented. The technique involves complex software transformation, an implementation is described to demonstrate the viability of the method. (Chapter 5.)

- The self-validating property of the technique to create model generators is presented. A meta model generator can be created by an implementation of the current technique and applies to validation of that implementation of the technique. (Chapter 5 and 4.)

- An evaluation is carried out to determine the utility of the presented technique when applied to model transformation validation. The evaluation is by case study and comparison to existing state-of-the-art related model generation techniques. (Chapter 6.)

7.2 Future Work

This thesis has presented a technique to create model generators from meta models and applied the generators to model transformation validation. Several further applications and extension of technique may be possible. The current model generation may be improved to automatically consider validation strategies, by the adaptation of state-of-the-art validation techniques. The current technique applies to validation but may be adapted to and applied to model transformation verification. The presented technique involves several complex transformation that may be applied to other problems, if generalised. The present technique models may be extended, adapted and generalised to synthesize several promising future applications.

The presented technique supports the evaluation of transformations in a purely black-box way, without regard to the internal structure or actions of the transformations. In general

software validation techniques, experts can use several methods and information about the system under validation to create input data, a comprehensive review is found in [148]. In the presented technique, model generation can optionally be guided by such experts to generate desirable models for model transformation validation. Furthermore, recent studies have applied "search based" techniques to optimise test data for particular validation purposes, for example to maximise error detection. A review of state-of-the-art search based techniques to optimise test case generation is found in [5]. Model based techniques are also used to inform the creation of test cases, as found in [143]. The test generation capabilities of the current technique may be improved by considering the internal working of the transformation in conjunction with search based [5], expert based [148] or model based techniques [143].

The current technique exploits automated model transformation and model-finding verification tools to generate models. The technique automatically converts the complex input specification of a model transformation, the source meta model, to create a model generator. To extend this work, entire model transformations- including both input and output meta models and model transformation definitions- may be converted to verification formalisms, similar to [10]. By doing so, verification tools can be applied to analyse the entire transformation for consistency. Several previous techniques have applied verification tools to model transformation verification with varying results. In state-of-the-art transformation verification techniques simplifications and abstraction of transformations are required [97, 10] manual conversions are required [118, 10] and expert interaction required to guide verification [118]. The automation of conversion and analysis in the current method may benefit the verification of transformation artefacts.

In the current technique, several advanced concepts of transformation are applied, including transformation tracing, transformation chains, higher order transformation and meta model transformation. The transformations work to create a representation of a meta models in a software verification formalism. Also created by transformation, is a transformation to convert instances generated by the verification formalism. The created instance transformation is on a different meta level to the original model transformation. This novel application

of model transformations in the current technique may be applicable to other problems in model driven software development. By discovery and exploration of apparently related problems, multi-level model transformations may be generalised and further applied.

Several complex model transformations are used in the implementation of the presented technique. Model to model, model to text, higher order and multi-level transformations are applied to create an implementation of the technique. The transformations involved in the implementation have been evaluated by case-study and self-application of the presented technique. If the complex transformations are to be applied to solve other problems in model driven software development, further analysis of the transformations involved may be carried out. This can be achieved by application of verification techniques to the complex transformations. However, existing verification techniques are difficult to apply to large and complex transformations requiring expert interaction [118], simplified transformations [97] and manual conversion [58]. Furthermore, the current technique creates models transformations as an output, making expert interaction for verifying each created transformation infeasible. However, the current technique has been analysed to by validation and self-application of the technique. Investigation is required on how to further analyse the transformations involved in the current technique.

LIST OF REFERENCES

[1] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14:159–192, June 1982. ISSN 0360-0300. URL http://doi.acm.org/10.1145/356876.356879.

[2] D. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier. SiTra: Simple transformations in Java. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 351–364. Springer, 2006. ISBN 3-540-45772-0. URL http://dblp.uni-trier.de/db/conf/models/models2006.html.

[3] D. H. Akehurst, O. Uzenkov, W. G. Howells, K. D. Mcdonald-maier, and B. Bordbar. An Experiment in Using Model Driven Development: Compiling UML State Diagrams into VHDL. In *in Using Model Driven Development. Forum on specification & Design Languages (FDL'07)*, 2007.

[4] M. Alanen. EMF/ECORE Integration in the Coral Modeling Framework. In *Proceedings of the NWUML*, 2006.

[5] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36:742–762, 2010. ISSN 0098-5589. URL http://doi.ieeecomputersociety.org/10.1109/TSE.2009.52.

[6] F. Alizon, M. Belaunde, G. DuPre, B. Nicolas, S. Poivre, and J. Simonin. Les modèles dans

l'action à France Télécom avec SmartQVT. In *Génie logiciel: Congrès Journées Neptune No5*, 2007. URL http://smartqvt.elibel.tm.fr.

[7] F. Allilaire, J. Bézivin, F. Jouault, and I. Kurtev. Atl-eclipse support for model transformation. In *Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France*, volume 66, 2006.

[8] K. Anastasakis. *A Model Driven Approach for the Automated Analysis of UML Class Diagrams*. PhD thesis, School of Computer Science, University of Birmingham UK, 2009.

[9] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 436–450, Nashville, USA, 2007. Springer.

[10] K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In B. Baudry, A. Faivre, S. Ghosh, and A. Pretschner, editors, *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[11] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On Challenges of Model Transformation from UML to Alloy. *Software and Systems Modeling, Special Issue on MoDELS 2007*, 2009.

[12] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for uml design models. *Software Testing Verification and Reliability*, 13(2):95–127, 2003. URL http://doi.wiley.com/10.1002/stvr.270.

[13] V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser. Using traceability to enhance mutation analysis dedicated to model transformation. In *proceedings of the Workshop MoDeVVa 2010 associated with Models2010 conference*, Oslo, Norway, Oct. 2010.

[14] ArgoUML. *ArgoUML 0.32 Manual*, 2005. URL http://argouml-downloads.tigris.org/nonav/argouml-0.32/manual-0.32.pdf.

[15] M. Asztalos, L. Lengyel, and T. Levendovszky. A formalism for describing modeling transformations for verification. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '09, pages 2:1–2:10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-876-6. URL http://doi.acm.org/10.1145/1656485.1656487.

[16] M. Asztalos, L. Lengyel, and T. Levendovszky. Toward automated verification of model transformations: A case study of analysis of refactoring business process models. *ECEASST*, 21, 2009. URL http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/287.

[17] M. Asztalos, L. Lengyel, and T. Levendovszky. Towards automated, formal verification of model transformations. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:15–24, 2010.

[18] M. Asztalos, L. Lengyel, and T. Levendovszky. A formalism for the automated verification of model transformations. In *Proceedings of the International Symposium of Hungarian Researchers on Computational Intelligence and Informatics (CINTI 2009)*, 2010.

[19] C. Atkinson and T. Kuhne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003. ISSN 0740-7459. URL http://dx.doi.org/10.1109/MS.2003.1231149.

[20] ATLAS Contributors. *ATL Transformations*, 2007. URL http://www.eclipse.org/m2m/atl/atlTransformations/. retrived on 05-01-2011.

[21] ATLAS Contributors. ATL/Tutorials - Create a simple ATL transformation, 2007. URL http://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation. retrived on 05-01-2011.

[22] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, Jan. 2004. doi: 10.1109/TDSC.2004.2. URL http://dx.doi.org/10.1109/TDSC.2004.2.

[23] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: Great. *Electronic Communications of the EASST*, 1:8, 2006.

[24] L. Baresi and P. Spoletini. On the use of alloy to analyze graph transformation systems. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 306–320. Springer Berlin / Heidelberg, 2006. URL http://dx.doi.org/10.1007/11841883_22. 10.1007/1184188322.

[25] L. Baresi, K. Ehrig, and R. Heckel. Verification of model transformations: a case study with bpel. In *Proceedings of the 2nd international conference on Trustworthy global computing*, TGC'06, pages 183–199, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-75333-8, 978-3-540-75333-9. URL http://portal.acm.org/citation.cfm?id=1776656.1776673.

[26] L. Baresi, V. Rafe, A. T. Rahmani, and P. Spoletini. An efficient solution for model checking graph transformation systems. *Electron. Notes Theor. Comput. Sci.*, 213:3–21, May 2008. ISSN 1571-0661. URL http://portal.acm.org/citation.cfm?id=1371270.1371520.

[27] B. Baudry. Testing model transformations: A case for test generation from input domain models. In *Model Driven Engineering for Distributed Real-time Embedded Systems*. Hermes, 2009. URL http://www.irisa.fr/triskell/publis/2009/Baudry09b.pdf.

[28] K. Beck. *Test Driven Development – by Example*. The Addison Wesley Signature Series. Addison Wesley, New York, 2003.

[29] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-12094-4.

[30] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: http://dx.doi.org/10.1109/FOSE.2007.25. URL http://dx.doi.org/10.1109/FOSE.2007.25.

[31] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2): 171–188, 2005. URL http://dx.doi.org/10.1007/s10270-005-0079-0.

[32] J. Bézivin, F. Jouault, and D. Touzet. An Introduction to the ATLAS Model Management Architecture. *Research Report LINA,(05-01)*, 2005.

[33] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[34] B. Boris. *Software testing techniques*. Van Nostrand Reinhold, 1990. ISBN 0442245920.

[35] J. Börstler, M. S. Hall, M. Nordström, J. H. Paterson, K. Sanders, C. Schulte, and L. Thomas. An evaluation of object oriented example programs in introductory programming textbooks. *SIGCSE Bull.*, 41:126–143, January 2010. ISSN 0097-8418. URL http://doi.acm.org/10.1145/1709424.1709458.

[36] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 85–94, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2684-5. URL http://portal.acm.org/citation.cfm?id=1190616.1191225.

[37] J. Cabot and E. Teniente. Transformation techniques for ocl constraints. *Sci. Comput. Program.*, 68(3):152–168, Oct. 2007. doi: 10.1016/j.scico.2007.05.001. URL http://dx.doi.org/10.1016/j.scico.2007.05.001.

[38] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *J. Syst. Softw.*, 83:283–302, February 2010. ISSN 0164-1212. URL http://dx.doi.org/10.1016/j.jss.2009.08.012.

[39] D. Calegari, C. Luna, N. Szasz, and A. Tasistro. Experiment with a type-theoretic approach to the verification of model transformations. In *Proceedings of Chilean Workshop on Formal Methods (ChWFM)*, November 2009.

[40] D. Calegari, C. Luna, N. Szasz, and A. Tasistro. A type-theoretic framework for certified model transformations. In *Proceedings of the 13th Brazilian Symposium on Formal Methods, SBMF 2010*, November 2010.

[41] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. Ocl contracts for the verification of model transformations. *ECEASST*, 24, 2009. URL http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/326.

[42] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[43] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag. ISBN 3-540-11212-X. URL http://portal.acm.org/citation.cfm?id=648063.747438.

[44] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. URL http://doi.acm.org/10.1145/800157.805047.

[45] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[46] J. Cuadrado and J. Molina. Modularization of model transformations through a phasing mechanism. *Software and Systems modeling*, 8(3):325–345, 2009. ISSN 1619-1366.

[47] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006. ISSN 0018-8670.

[48] A. Darabos, A. Pataricza, and D. Varró. Towards testing the implementation of graph transformations. In *Proc. of the Fifth International Workshop on Graph Transformation and Visual Modelling Techniques*, ENTCS. Elsevier, 2006.

[49] A. Darabos, A. Pataricza, and D. Varró. Towards testing the implementation of graph transformations. *Electron. Notes Theor. Comput. Sci.*, 211:75–85, April 2008. ISSN 1571-0661. URL http://portal.acm.org/citation.cfm?id=1367148.1367342.

[50] J. de Lara and G. Taentzer. Automated model transformation and its validation using atom 3 and agg. In *Diagrams*, pages 182–198, 2004. URL http://springerlink.metapress.com/openurl.asp?genre=article{&}issn=0302-9743{&}volume=2980{&}spage=182.

[51] E. W. Dijkstra. Structured programming. In J. N. Buxton and B. Randell, editors, *Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee*, 1969.

[52] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, july 2008. ISSN 0278-0070. doi: 10.1109/TCAD.2008.923410.

[53] N. Een and N. Sörensson. An extensible sat-solver. In *6th international conference on theory and applications of satisfiability testing*, 2003.

[54] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Cerioli, editor, *Fundamental Ap-*

proaches to Software Engineering, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer Berlin / Heidelberg, 2005. URL http://dx.doi.org/10.1007/978-3-540-31984-9_5.

[55] H. Ehrig, C. Ermel, F. Hermann, and U. Prange. On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 241–255, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04424-3. URL http://dx.doi.org/10.1007/978-3-642-04425-0_18.

[56] K. Ehrig, J. M. Küster, and G. Taentzer. Generating instance models from meta models. *Software and System Modeling*, 8(4):479–500, 2009. URL http://dx.doi.org/10.1007/s10270-008-0095-y.

[57] C. Faure and F. Allilaire. Atl basic examples and patterns - the tree to list example, 2007. URL http://www.eclipse.org/m2m/atl/basicExamples_Patterns/Tree2List/. retrived on 05-01-2011.

[58] C. Fiorentini, A. Momigliano, M. Ornaghi, and I. Poernomo. A constructive approach to testing model transformations. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 77–92, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13687-7, 978-3-642-13687-0. URL http://portal.acm.org/citation.cfm?id=1875847.1875855.

[59] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*, pages 29–40, 2004.

[60] F. Fleurey, Z. Drey, D. Vojtisek, and C. Faucher. Kermeta language. *Reference manual, Internet: http://www.kermeta.org/docs/kermeta-manual.pdf*, 2006.

[61] F. Fleurey, B. Baudry, P.-A. Muller, and Y. Traon. Qualifying input test data for model

transformations. *Software and Systems Modeling*, 8:185–203, 2009. ISSN 1619-1366. URL http://dx.doi.org/10.1007/s10270-007-0074-8. 10.1007/s10270-007-0074-8.

[62] P. Frankl and E. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19:202–213, 1993. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.221133.

[63] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[64] A. Gerber and K. Raymond. Mof to emf: there and back again. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, eclipse '03, pages 60–64, New York, NY, USA, 2003. ACM. URL http://doi.acm.org/10.1145/965660.965673.

[65] M. Gogolla and A. Vallecillo. *Tract*able model transformation testing. In *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*, volume 6698 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2011. ISBN 978-3-642-21469-1.

[66] J. Gradecki and J. Cole. *Mastering Apache Velocity*. Wiley, 2003.

[67] F. Hermann, M. Hülsbusch, and B. König. Specification and verification of model transformations. *ECEASST*, 30, 2010.

[68] D. Jackson. Automating first-order relational logic. *ACM SIGSOFT Software Engineering Notes*, 25(6):130–139, 2000.

[69] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002. ISSN 1049-331X.

[70] F. Jouault. Loosely Coupled Traceability for ATL. *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, 2005.

[71] F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185, 2006.

[72] F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[73] D. Karagiannis and H. Kuhn. Metamodelling platforms. *Lecture Notes in Computer Science*, pages 182–182, 2002. ISSN 0302-9743.

[74] G. Karsai and A. Narayanan. On the correctness of model transformations in the development of embedded systems. In *Proceedings of the 13th Monterey conference on Composition of embedded systems: scientific and industrial issues*, pages 1–18, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-77418-1, 978-3-540-77418-1. URL http://portal.acm.org/citation.cfm?id=1785644.1785646.

[75] G. Karsai and A. Narayanan. Model-driven development of reliable automotive services. chapter Towards Verification of Model Transformations Via Goal-Directed Certification, pages 67–83. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-70929-9. URL http://dx.doi.org/10.1007/978-3-540-70930-5_5.

[76] S. Kelly, K. Lyytinen, and M. Rossi. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *Advanced Information Systems Engineering*, pages 1–21. Springer, 1996.

[77] H. Kern. The Interchange of (Meta) Models between MetaEdit+ and Eclipse EMF Using M3-Level-Based Bridges. In *8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA*, 2008.

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

[79] D. Kolovos, R. Paige, L. Rose, and F. Polack. Unit Testing Model Management Operations. In *Proc. 5th Workshop on Model Driven Engineering Verification and Validation (MoDeVVa), IEEE ICST, Lillehammer, Norway*, 2008.

[80] J. Kozikowski. *A Bird's Eye view of AndroMDA*, 2005. URL http://www.andromda.org/docs/contrib/birds-eye-view.html.

[81] I. Kurtev. State of the art of qvt: A model transformation language standard. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 377–393. Springer Berlin / Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-89020-1_26. 10.1007/978-3-540-89020-1_26.

[82] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: An initial appraisal. In *International Symposium on Distributed Objects and Applications, DOA 2002*, 2002. URL http://doc.utwente.nl/55814/.

[83] I. Kurtev, K. Van Den Berg, and F. Jouault. Evaluation of rule-based modularization in model transformation languages illustrated with ATL. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1202–1209. ACM, 2006. ISBN 1595931082.

[84] J. Küster. Systematic validation of model transformations. In *Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004)*, Portugal, 2004.

[85] J. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006. ISSN 1619-1366.

[86] J. Küster, R. Heckel, and G. Engels. Defining and validating transformations of uml models. In *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 145–152, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7803-8225-0. URL http://portal.acm.org/citation.cfm?id=1153917.1153982.

[87] J. M. Küster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *MoDELS Workshops*, pages 193–204, 2006. URL http://dx.doi.org/10.1007/978-3-540-69489-2_24.

[88] J. M. Küster, T. Gschwind, and O. Zimmermann. Incremental development of model transformation chains using automated testing. In *MoDELS*, pages 733–747, 2009. URL http://dx.doi.org/10.1007/978-3-642-04425-0_60.

[89] M. Lamari. Towards an automated test generation for the verification of model transformations. In *SAC*, pages 998–1005, 2007. URL http://doi.acm.org/10.1145/1244002.1244220.

[90] K. Lano. Using B to verify UML Transformations. October 2006.

[91] K. Lano. *Model Transformation Specification and Verification*, pages 349–395. John Wiley & Sons, Inc., 2009. ISBN 9780470522622. URL http://dx.doi.org/10.1002/9780470522622.ch14.

[92] K. Lano and D. Clark. Model transformation specification and verification. In *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, pages 45–54, 2008.

[93] K. Lano and S. Kolahdouz-Rahimi. Specification and verification of model transformations using uml-rsds. In *Proceedings of the 8th international conference on Integrated formal methods*, IFM'10, pages 199–214, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16264-9, 978-3-642-16264-0. URL http://portal.acm.org/citation.cfm?id=1929463.1929478.

[94] M. Lawley and J. Steel. Practical Declarative Model Transformation With Tefkat. *MoDELS Satellite Events*, pages 139–150, 2005.

[95] H. Ledang and H. Dubois. Proving model transformations. In *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on*, pages 35–44, 2010.

[96] Y. Lin, J. Zhang, and J. Gray. *A Testing Framework for Model Transformations*, pages 219–236. Springer, 2005. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.9495.

[97] L. Lúcio, B. Barroca, and V. Amaral. A technique for automatic validation of model transformations. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 136–150, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16144-8, 978-3-642-16144-5. URL http://portal.acm.org/citation.cfm?id=1926458.1926472.

[98] J. A. Mc Quillan and J. F. Power. White-Box Coverage Criteria for Model Transformations. 2009.

[99] MediniQVT. *MediniQVT Website*. URL http://projects.ikv.de/qvt.

[100] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. ISSN 1571-0661. doi: DOI:10.1016/j.entcs.2005.10.021. URL http://www.sciencedirect.com/science/article/pii/S1571066106001435. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[101] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, OMG, 2003.

[102] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2. URL http://doi.acm.org/10.1145/378239.379017.

[103] J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 376–390. Springer Berlin / Heidelberg, 2006. URL http://dx.doi.org/10.1007/11787044_28.

[104] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing*. Wiley, second edition, June 2004. ISBN 0471469122.

[105] A. Narayanan and G. Karsai. Specifying the correctness properties of model transformations. In *Proceedings of the third international workshop on Graph and model transformations*, GRaMoT '08, pages 45–52, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-033-3. URL http://doi.acm.org/10.1145/1402947.1402957.

[106] A. Narayanan and G. Karsai. Towards verifying model transformations. *Electron. Notes Theor. Comput. Sci.*, 211:191–200, April 2008. ISSN 1571-0661. URL http://portal.acm.org/citation.cfm?id=1367148.1367351.

[107] A. Narayanan and G. Karsai. Verifying model transformations by structural correspondence. *ECEASST*, 10, 2008. URL http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/157.

[108] A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai. Automatic domain model migration to manage metamodel evolution. In A. Schürr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 706–711. Springer Berlin / Heidelberg, 2009. URL http://dx.doi.org/10.1007/978-3-642-04425-0_57.

[109] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Verlag, 2002.

[110] OMG. *UML 2.0 superstructure final adopted specification*, 2003.

[111] OMG. *OMG XML Metadata Interchange (XMI) Specification Version 2.0*. Object Management Group, Framingham, Massachusetts, May 2003.

[112] OMG. *OMG: MOF Model to Text Transformation Language*. OMG, 2004. URL http://www.omg.org/cgi-bin/apps/doc?ad/04-04-07.pdf.

[113] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*. OMG, 2004. URL www.omg.org.

[114] OMG. *OCL Version 2.0*, 2006. Document id: formal/06-05-01.

[115] OMG. *MOF QVT Final Adopted Specification*. Object Modeling Group, 2007.

[116] J. Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. Constructing and visualizing transformation chains. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 17–32, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-69095-5. URL http://dx.doi.org/10.1007/978-3-540-69100-6_2.

[117] I. Poernomo. Proofs-as-model-transformations. In *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, ICMT '08, pages 214–228, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-69926-2. URL http://dx.doi.org/10.1007/978-3-540-69927-9_15.

[118] I. Poernomo and J. Terrell. Correct-by-construction model transformations from partially ordered specifications in coq. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, pages 56–73, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16900-7, 978-3-642-16900-7.

[119] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag. ISBN 3-540-11494-7. URL http://portal.acm.org/citation.cfm?id=647325.721668.

[120] V. Rafe and A. T. Rahmani. On the analysis and verification of graph transformation systems. In *proc. of 3th International Conference on Information, Knowledge and Technology (IKT07)*, 2007.

[121] M. Richters and M. Gogolla. On formalizing the uml object constraint language ocl. In *ER*, pages 449–464, 1998.

[122] M. Richters and M. Gogolla. A metamodel for ocl. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 156–171, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66712-1. URL http://portal.acm.org/citation.cfm?id=1767297.1767315.

[123] L. Rose, D. Kolovos, R. Paige, and F. Polack. Model migration with epsilon flock. In L. Tratt and M. Gogolla, editors, *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin / Heidelberg, 2010. URL http://dx.doi.org/10.1007/978-3-642-13688-7_13.

[124] A. A. Sani, F. Polack, and R. Paige. Generating formal model transformation using a template-based approach. In A. Miyazawa, editor, *Proceedings of the Third York Doctoral Syposium 2010*, 2010.

[125] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, University of Oxford, 1998.

[126] D. Schmidt. Model-driven engineering. *IEEE computer*, 39(2):25–31, 2006.

[127] E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, sept.-oct. 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1231147.

[128] S. Sen, B. Baudry, and J.-M. Mottu. On combining multi-formalism knowledge to select models for model transformation testing. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:328–337, 2008.

[129] S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 148–164, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02407-8. URL http://dx.doi.org/10.1007/978-3-642-02408-5_11.

[130] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[131] S. M. A. Shah, K. Anastasakis, and B. Bordbar. Using traceability for reverse instance transformations with SiTra. In *Design and Architectures for Signal and Image Processing (DASIP 2008). Special Session on Formal Models, Transformations and Architectures for Reliable Embedded System Design.*, Bruxelles, Belgium, 2008.

[132] S. M. A. Shah, K. Anastasakis, and B. Bordbar. From uml to alloy and back again. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '09, pages 4:1–4:10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-876-6. URL http://doi.acm.org/10.1145/1656485.1656489.

[133] S. M. A. Shah, K. Anastasakis, and B. Bordbar. From uml to alloy and back again. In S. Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 158–171. Springer Berlin / Heidelberg, 2010. URL http://dx.doi.org/10.1007/978-3-642-12261-3_16. 10.1007/978-3-642-12261-3_16.

[134] J. Siegel. *Developing in OMG's Model Driven Architecture.* Object Management Group, 2002.

[135] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[136] J. Steel and M. Lawley. Model-based test driven development of the tefkat model-transformation engine. In *ISSRE*, pages 151–160, 2004. URL http://doi.ieeecomputersociety.org/10.1109/ISSRE.2004.23.

[137] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition).* Addison-Wesley Professional, 2 edition, Dec. 2008. ISBN 0321331885.

[138] G. Taentzer, K. Ehrig, E. Guerra, J. D. Lara, T. Levendovszky, U. Prange, and D. Varro. Model transformations by graph transformations: A comparative study. In *Model Transformations in Practice Workshop at MoDELS 2005, Montego*, page 05, 2005.

[139] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, Apr. 2004. URL http://coq.inria.fr.

[140] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 18–33, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02673-7. URL http://dx.doi.org/10.1007/978-3-642-02674-4_3.

[141] E. Torlak and D. Jackson. Kodkod: A relational model finder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

[142] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[143] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

[144] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and Systems Modeling*, 3:85–113, 2004. ISSN 1619-1366. URL http://dx.doi.org/10.1007/s10270-003-0050-x. 10.1007/s10270-003-0050-x.

[145] D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007. ISSN 0167-6423.

[146] D. Varró and A. Pataricza. Automated formal verification of model transformations. In J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, editors, *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, pages 63–78. Technische Universität München, September 2003.

[147] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, editors, *UML 2004 - The Unified Modelling Language*, volume 3273 of *Lecture Notes in Computer Science*, pages 290–304. Springer Berlin / Heidelberg, 2004. URL http://dx.doi.org/10.1007/978-3-540-30187-5_21.

[148] S. Vegas, N. Juristo, and V. Basili. *Identifying relevant information for testing technique selection: An instantiated characterization schema.* Springer Netherlands, 2003.

[149] E. Vépa, J. Bézivin, H. Brunelière, and F. Jouault. Measuring model repositories. In *Proceedings of the 1st Workshop on Model Size Metrics (MSM'06) co-located with MoDELS'2006*, 2006.

[150] J. Wang, S.-K. Kim, and D. Carrington. Verifying metamodel coverage of model transformations. In *Proceedings of the Australian Software Engineering Conference*, pages 270–282, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2551-2. URL http://portal.acm.org/citation.cfm?id=1129030.1129458.

[151] J. Wang, S.-K. Kim, and D. Carrington. Automatic generation of test models for model transformations. In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 432–440, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3100-7. URL http://portal.acm.org/citation.cfm?id=1395083.1395694.