A MODEL DRIVEN APPROACH FOR THE AUTOMATED ANALYSIS OF UML CLASS DIAGRAMS

by

KYRIAKOS ANASTASAKIS

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

School of Computer Science
The University of Birmingham
May 2009
Abstract

The Unified Modeling Language (UML) is widely considered as the *de facto* standard for the design of Object Oriented systems. UML class diagrams are used to depict the static structure of a system with its entities and the relationships between them. The Object Constraint Language (OCL) is a textual language based on first-order logic and can be used to define constraints on the elements of class diagrams. The lack of strong formal semantics for the UML makes it difficult to analyse UML models. This work utilises Alloy to analyse UML models. More specifically, this work employs the Model Driven Architecture (MDA) technology to achieve an automated transformation of UML class diagrams enriched with OCL constraints to Alloy. This is accomplished by defining a number of transformation rules from UML and OCL concepts to Alloy concepts. However, due to the different philosophies of the UML and Alloy, the languages have a number of fundamental differences. These differences and their effect on the definition of the transformation rules is discussed. To bridge the differences and to achieve fully automated analysis of UML class diagrams though Alloy, a UML profile for Alloy is developed. Details of our implementation of the model transformation in the SiTra transformation engine and a number of case studies are also presented.
Acknowledgments

First, I would like to thank my supervisor, Behzad Bordbar, for being a constant source of inspiration and motivation throughout the course of this project. I would also like to thank Geri Georg and Indrakshi Ray for the fruitful collaboration. Many thanks also go to Mark Ryan and Hayo Thielecke for providing criticisms on the progress of the work during the thesis group meetings.

This work would not have been possible without the financial support of the Greek State Scholarships Foundation (I.K.Y.). I would also like to thank the School of Computer Science for providing financial support for the final six months of this project.

It is needless to say how grateful I am to my family for their constant support. Thanks go to the people at the School of Computer Science, who made every day in Birmingham pleasant and worthwhile: Philipp (who also helped me with the thesis submission), Arjun, Thanos, Elias, Nikos, Thodoris, Pietro, Andrea, Seyyed, Ariff, Mohamed, Chris, Andy, Thomas, Marek, Ben, Damien, Peter, Neale and many others. I also need to thank the people behind the Alloy mailing list and especially Felix Chang for providing prompt and valuable support on the Alloy language.
Preface

Here we present some conventions we follow in this thesis. In this thesis we follow the convention of the UML specification [111, page 92, constraints 4,5]. More specifically if an association end multiplicity has not been defined, it denotes a 1..1 multiplicity.

Unless otherwise indicated in the text, the version of Alloy Analyzer that was used to analyse the Alloy models presented in this thesis, is Alloy Analyzer 4.0 RC11. The SAT solver used for the analyses was MiniSat with Unsat Core SAT. The Alloy Analyzer was running on the server of the School of Computer Science, called Gromit. At the time of the experiments Gromit has a dual core AMD Opteron 175 CPU running at 2.2GHz. The memory of the system is 4GB. The experiments presented here were executed multiple times and any performance information provided is the mean time of the runs.

Alloy is a relatively new language and during the course of our work its notation and tool support were evolving. This work is based on the Alloy language version 3 and was adapted to support some of the features of Alloy version 4. Finally, for reasons of brevity the *Alloy Analyzer* is sometimes referred to as *the Analyzer*. 
# CONTENTS

1 Introduction .......................... 1  
1.1 Overview of our Approach .................. 3  
1.2 Contribution .......................... 4  
1.3 Publications .......................... 7  
1.4 Thesis Outline .......................... 8  

2 Background and Related Work ....... 10  
2.1 Preliminaries ......................... 10  
2.1.1 The Unified Modeling Language (UML) ........ 11  
2.1.2 Alloy .......................... 12  
2.1.3 Model Driven Architecture (MDA) ........ 16  
2.2 Related Work .......................... 20  

3 UML and Alloy metamodels ......... 26  
3.1 A UML Profile for Alloy ............... 27  
3.1.1 Constraints of the Profile .......... 29  
3.1.2 Stereotypes ........................ 32  
3.2 A MOF Compliant Alloy Metamodel ....... 35  
3.2.1 Existing Approaches ............... 36  
3.2.2 Proposed Approach ............... 37  
3.2.3 Remarks .......................... 42  

4 UML2Alloy: Transforming Static Systems ......... 44  
4.1 Transformation Rules for Class Diagrams ........ 45  
4.1.1 Package2Modules .................. 45  
4.1.2 Classes2Signatures ............... 48  
4.1.3 Properties2Fields ............... 53  
4.1.4 Associations .................. 62  
4.1.5 Types .......................... 71  
4.1.6 Integers .......................... 72  
4.1.7 Enumerations .................. 74
LIST OF FIGURES

1.1 Overview of our Approach ............................................. 4
2.1 An Example Alloy Model ............................................ 16
2.2 Results Produced by the Alloy Analyzer .......................... 17
2.3 Model Transformation in the MDA ................................. 18
3.1 Overview of our Profiling Approach .............................. 28
3.2 The NamedElement Metaclass is Required to have a Name in our Profile ........................................ 28
3.3 The Stereotypes Defined by our Profile ......................... 32
3.4 The EBNF Representation of a Simple Grammar ................ 35
3.5 Overview of a Bidirectional Mapping Between the EBNF and MOF Notation of Alloy ................................. 37
3.6 EBNF Representation and MOF Metamodel of the sigDecl Part of the Grammar ..................................... 40
4.1 Simplified UML Metamodel ........................................ 46
4.2 Simplified Metamodel for Alloy Signatures .................... 47
4.3 Example Generalization Class Diagram and Instance .......... 50
4.4 Multiple Inheritance .................................................. 53
4.5 Example Subsetting Property ...................................... 59
4.6 Example Derived Union Property ................................. 61
4.7 Example Association End Representation in Alloy ............ 64
4.8 Transforming N-Ary Associations to Binary .................... 70
4.9 Representing UML Datatypes in Alloy ........................... 72
4.10 A Simple Alloy Model with Integer Values .................... 74
4.11 Representing UML Enumerations in Alloy ....................... 75
4.12 Class Promotion of the Marriage Association Class .......... 76
4.13 An Equivalent Model of the Qualified Association Between Bank and Person of Figure 5.1 ................................. 77
4.14 Using the Extends Keyword to Declare Disjoint Subsignatures .................................................. 79
4.15 Using the In Keyword to Declare Overlapping Subsets ....... 80
4.16 Overview of the Transformation Rules .......................... 83
4.17 A Simple Example to Illustrate our Transformation Rules ... 85
LIST OF TABLES

4.1 Property Collection Types (from [111, Sec. 7.3.44]) . . . . . . . . . . . . . . 60
4.2 Instance of an N-ary Association . . . . . . . . . . . . . . . . . . . . . . . 70
5.1 Truth Table for the OclIsTypeOf() Operation . . . . . . . . . . . . . . . . . . 98
5.2 OclIsTypeOf() Using Set Inclusion . . . . . . . . . . . . . . . . . . . . . . . 98
CHAPTER 1

INTRODUCTION

The Unified Modeling Language (UML)\(^1\) [49, 116, 124] is a popular family of modelling languages used to specify various static and dynamic aspects of systems. The UML is often referred to as the ‘de facto’ [116, 91] language, used in the industry mainly for modelling Object Oriented systems. The UML offers a rich number of diagrammatic notations ideal to support the modelling of different views of a system. For example, use case diagrams [49, Chapter 9] can be used to model the interactions between the users of a system and the system, class diagrams [49, Chapter 3] depict the static structure of a system with its entities and their relationships, while sequence diagrams [49, Chapter 4] draw a scenario of the sequence of interactions between the instances of the entities of the system. To complement the UML diagrammatic notation, the Object Constraint Language (OCL) [146, 145] can be used to express constraints (i.e. logical statements that must be satisfied by all valid instances of the system) and specify the effect of operations in a declarative way.

Exploiting the possibility to analyse UML models and reason about the properties of a system early in the development process has received considerable attention [15, 46, 85, 96]. Proposed methods suggest to express UML concepts with well-established formalisms

\(^1\)This work is based on the UML specification version 2.0 [109, 111].
such as Z [149], Object-Z [127] and Higher Order Logic (HOL) [32]. However, often tool support for such formalisms relies on theorem provers, which usually require user guidance and expertise to use.

The lack of tools with automated analysis capabilities was one of the incentives for the development of the Alloy language, according to Jackson [77]. Alloy [77] is an increasingly popular declarative textual modelling language based on first-order relational logic. The language provides support for notions of object orientation (such as parent-child classification hierarchies). The Alloy Analyzer is a tool which provides support for fully automated analysis of Alloy models with the help of SAT solvers. The tool provides the capability to produce a random instance of the model that conforms to the constraints of the system (simulation), to check if a requirement holds according to the specification (assertion) and to debug overconstrained models [126]. In this work we propose to automate the transformation of UML class diagrams enriched with OCL constraints into the Alloy language for the purpose of analysis.

The proposed approach can be used in the early phases of the software development lifecycle to analyse the design of a system. Often UML class diagrams are used to develop the design of a system. OCL expressions are also used to express system constrains. Such UML class diagrams and OCL constraints can be automatically transformed to Alloy using the approach and the tool support presented in this thesis.

In particular, using the simulation capabilities of the approach (inherited by the simulation capabilities of the Alloy Analyzer) we can generate a random instance of the model. Using this feature it is possible to generate a number of instances conforming to the model and to manually inspect that no unexpected instances are produced, thus increasing the confidence of the modeller in the design.

It is also possible to check if the model corresponding to the design of system is overconstrained. In such a case the presented approach can be used to identify which elements
of the UML class diagram and/or OCL constraints are responsible for the conflict, using the capabilities of the Alloy Analyzer to debug overconstrained models. Finally, it is possible to check whether the design of the system conforms to user requirements. For example, assume the case of a client and server communicating over a secure connection utilising cryptography techniques. In such a scenario a user requirement is that the design of the system does not allow someone intercepting the communication to acquire sensitive information exchanged between the client and the server. As demonstrated in Section 7.3.2 it is possible to check such a user requirement with the presented approach, using assertions. An overview of the proposed approach is presented in the next section.

1.1 Overview of our Approach

Figure 1.1 depicts an outline of our approach to transform UML class diagrams and OCL constraints into Alloy for the purpose of analysis. In particular, the UML class diagrams metamodel (Class Diagrams MM) and the OCL metamodel (OCL MM) were retrieved from the UML [111] and OCL [108] specifications respectively. Based on the class diagrams metamodel, a UML profile for Alloy was developed. Moreover a subset of OCL expressible in Alloy was identified. A number of transformation rules were defined from the UML profile for Alloy and the OCL subset metamodel elements to the Alloy metamodel elements (Alloy MM). The Alloy metamodel was developed using the EBNF representation of the Alloy grammar [77]. Using this approach every UML class diagram that conforms to the UML profile for Alloy and every OCL statement that is expressible in Alloy is automatically transformed to an Alloy model, which is an instance of the Alloy metamodel.

A detailed account of the contribution of this thesis is presented in the following section.
1.2 Contribution

The contribution of this thesis is the following:

- A UML profile for Alloy was developed. In particular:
  
  1. A subset of the UML metamodel, which is expressive enough to model basic class diagram concepts (such as Classes, Attributes, Associations, Generalization relations), was identified. (Section 3.1 and Appendix E).
  
  2. The UML metamodel subset was then extended with additional constraints in order to exclude UML concepts, which cannot be expressed in Alloy. This led to the development of a UML profile for Alloy. (Section 3.1.1 and Appendix E).
  
  3. A number of UML Stereotypes were also defined as part of the profile, to provide the ability to express Alloy concepts in UML class diagrams. This is required in order to achieve automated analysis of UML class diagrams, through Alloy. (Section 3.1.2).

- A Meta Object Facility (MOF) representation of the Alloy metamodel was developed, which was necessary to apply the Model Driven Architecture (MDA) technology to
automate the transformation from UML to Alloy. The metamodel was developed manually, but systematically from the Alloy grammar, using the algorithm presented in Section 3.2.

- The transformation rules from the UML metamodel elements into the Alloy metamodel elements are defined. More specifically:

  1. A description of the UML and the corresponding Alloy metamodel elements is presented (Chapter 4).

  2. The mapping rules from UML to Alloy are defined with the help of the Queries/Views and Transformations (QVT) specification. Natural language is also used to describe each QVT rule (Chapter 4).

  3. A number of UML concepts are not supported by our existing transformation rules. We present existing methods to express such concepts, using simpler UML constructs, which are supported by our transformation. (Sections 4.1.8 - 4.1.10).

  4. An example is used to demonstrate how the transformation rules can be used to transform a UML model to an equivalent Alloy model. (Section 4.2).

- We define the mappings from OCL elements into Alloy. Moreover, we present how OCL concepts, which do not have a direct equivalent Alloy representation (such as the notion of undefinedness) are treated by our transformation. (Chapter 5).

- The transformation rules defined in Chapters 4 and 5 are applicable on class diagrams, without any dynamics. These transformation rules are extended to allow the representation of dynamic concepts of UML class diagrams. In particular:

  1. We discuss the difference between static and dynamic models in the context of Alloy. (Section 6.1).
2. Existing literature is presented on the idioms of representing dynamic systems in Alloy. (Section 6.2)

3. We discuss how dynamic elements (such as OCL pre and post-conditions) can be mapped into Alloy. An algorithm is developed to automate the mapping of OCL pre and post conditions to Alloy. Finally, an example is provided to demonstrate how UML and OCL dynamics can be transformed to Alloy (Section 6.3).

- The transformation described in this thesis is implemented in a tool called UML2Alloy, to facilitate fully automated analysis of UML class diagrams. UML2Alloy inherits the analysis capabilities of the Alloy Analyzer (i.e. it provides support for simulation, counterexample generation and the ability to debug overconstrained UML class diagrams with OCL constraints). The details of the implementation are described in Section 7.1. In order to implement the rules we have extended the SiTra model transformation engine with traceability capabilities, as explained in Section 7.2. It is noteworthy that Babkin [26] and Pons and Garcia [118] report they have successfully used an early prototype version of UML2Alloy to automatically transform UML class diagrams with OCL constraints to Alloy.

- We provide a number of studies to demonstrate the feasibility of the approach presented in this thesis. In particular:

  1. A precise UML model of a Sudoku puzzle was developed and automatically transformed to Alloy using our UML2Alloy implementation. The transformation resulted in the Alloy Analyzer providing a solution for the Sudoku puzzle. This study was mainly used as a feasibility study of our approach. (Section 7.3.1).

  2. UML2Alloy has also been utilised for the analysis of model transformations. In order to analyse a model transformation using Alloy, the metamodel(s) of the
language(s) participating in the transformation have to be expressed in Alloy. UML2Alloy was used to automatically transform the MOF representation of the source and target language metamodels into Alloy. The analysis uncovered potential problems in the definition of the transformation rules. (Section 7.3.2)

3. UML2Alloy has also been used in the analysis of secure systems. In particular, the model of the login sequence of a client and server architecture, with the presence of a man-in-the-middle attack was developed. The attack was modelled using UML class diagrams and OCL. The system was transformed into Alloy, using UML2Alloy. The analysis demonstrated that if a security mechanism, such as the TLS protocol is implemented in the login sequence, certain security properties may hold. (Section 7.3.3).

Finally, this thesis contributes in a wider scope in the context of Model Driven Engineering. In particular this work uses a number of OMG standards (QVT, MOF2Text, MOF, UML, OCL) to transform UML class diagrams to Alloy, two languages with many similarities, but also many fundamental differences. With Domain Specific Model Languages (DSML) becoming increasingly popular, it is expected that complex model transformations between substantially different languages will be required, to allow interoperability and exchange of information between systems developed using different DSMLs. This thesis reports our experiences on such a complex model transformation, using OMG standards. (Chapter 8).

1.3 Publications

The following publications are a result of this this work:

1.4 Thesis Outline

Chapter 2 introduces the reader to preliminary concepts, such as the UML, MOF and Alloy, necessary to follow the material in this thesis. Moreover we discuss related work on formalising various diagram the UML. Chapter 3 presents a UML profile for Alloy and an algorithm
we used to develop an Alloy metamodel from the EBNF grammar of the language. The transformation rules from the UML class diagrams metamodel elements to the Alloy metamodel elements are presented in Chapter 4. We also discuss minor but important differences between the UML and the corresponding Alloy metamodel elements. In Chapter 5 we present the transformation rules from OCL into Alloy, while Chapter 6 presents an algorithm on how to transform the dynamics of UML into Alloy. In Chapter 7 we describe the implementation of the transformation rules and present a number of case studies, while Chapter 8 concludes the thesis with a discussion of our experiences and proposals for future directions.
CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter presents a brief introduction to preliminary concepts, such as the Unified Modeling Language (UML), the Model Driven Architecture (MDA) and the Alloy language that will be used later on in this document. Moreover, a review of existing literature on the formalisation of UML models for analysis and verification is discussed. Since 1997, when the Precise UML (pUML) [9] group was established to promote formal semantics on UML, a large amount of work has appeared with proposals to formalise various aspects of the UML. Therefore the related work discussed here does not cover the whole breadth of proposals to formalise the UML; however it highlights the most influential work.

2.1 Preliminaries

This section presents a gentle introduction to concepts, that will be used throughout this thesis. First an introduction to the concepts of the Unified Modeling Language (UML) and the Object Constraint Language (OCL) that will be used later on are presented. Alloy is then introduced and finally the concept of Model Driven Architecture (MDA) is presented.
2.1.1 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) [116, 49] is a family of graphical languages used to depict both static and dynamic aspects of a system at different levels of abstractions. The UML is often referred to as the *de facto* notation for the design of systems. It has replaced other previously popular object oriented modelling notations, such as the Object Modeling Technique (OMT) [125] and the Booch method notation [28].

The current version of the UML\(^1\) defines 13 different kinds of diagrams. In the next paragraphs the UML class diagrams, which are used in this thesis are briefly described. For more details on the rest of the UML diagrams, please refer to [116, 49, 111].

The Object Management Group (OMG) [5], which is a computer industry consortium, is responsible for developing and maintaining the UML specification. The OMG promotes more widespread use of the language in the industry. The UML specification consists of two documents, the infrastructure [109] and the superstructure [111] specifications. The infrastructure specification provides the fundamental notions of the language. The superstructure specification reuses the infrastructure libraries and provides the constructs and the notation for the different UML diagrams. This is essentially the notation, which can be used by a UML modeller to model a system.

The UML language is defined using the Meta Object Facility (MOF) [105]. The MOF is an OMG specification, which defines a notation that can be used to define the metamodels of languages. A metamodel of a language can be represented as a UML class diagram, which describes the concepts of a language (i.e. the abstract syntax of the language). In particular, the UML is specified using a four-layer hierarchy [109, Sec. 7.10]. More precisely, the topmost level of the hierarchy (often referred to as *M3*) is called the meta-metamodel. This layer specifies the notation that is used to define the abstract syntax of the UML. The MOF

\(^1\)As explained in the preface, this work is based on the UML specification version 2.0.
language is at this level. The next level is the metamodel level (also referred to as $M2$), which specifies the notions and the syntax of the UML. The UML abstract syntax definition is at this level. The next level is the model level (also referred to as $M1$), which consists of any models that can be defined using the UML language. Finally, the last level is the run-time instances (also referred to as $M0$). At this level there are instances of the user model.

This work focuses on UML class diagrams. A UML class diagram is a model that depicts the entities of a system, the attributes and operations of each entity, the relationships between the entities as well as the classification hierarchy of the entities. The abstract syntax of class diagrams (i.e. the class diagrams metamodel elements) is described in detail in Chapter 3.

To express constraints on the elements of a class diagram, the Object Constraint Language (OCL) was developed [146, 145, 108]. The OCL is a textual, declarative language based on first-order logic and set theory. In addition to expressing constraints on class diagrams, OCL can also be used to specify the effect of the execution of an operation, using pre and post conditions. A pre condition is an OCL statement that has to evaluate to true before the execution of an operation, while a post condition is a statement that has to evaluate to true when the operation terminates.

### 2.1.2 Alloy

Alloy [77] is a declarative textual modelling language based on first-order relational logic. The Alloy language has been inspired to a high degree by the popular Z notation [149]. An Alloy model consists of a number of *signature* declarations, *fields*, *facts* and *predicates*. Each signature denotes a set of *atoms*\(^2\), which are the basic entities in Alloy. Atoms are *indivisible* (they cannot be divided into smaller parts), *immutable* (their properties remain the same over time) and *uninterpreted* (they do not have any inherent properties) [77]. Each field has

\(^2\)In our work when we refer to ‘instances of signatures’, we mean the set of atoms a signature denotes.
to be declared under a signature and represents a relation between two or more signatures. Such relations are interpreted as sets of tuples of atoms. Alloy introduces facts which are statements that define constraints on the elements of the model, using first-order logic and relational expressions. Parameterised constraints, which are referred to as predicates, can be referred from other predicates or facts. Alloy is supported by a fully automated constraint solver, called Alloy Analyzer [2], which allows analysis of system properties by producing random instances of the model (simulation functionality). It is also possible to check that certain properties of the system (assertions) are satisfied. Assertions are properties of the system that need to be satisfied and should follow from the specification. The Alloy Analyzer works by automatically translating an Alloy model into a boolean expression with the help of the KodKod [138] model finder. The boolean expression is then automatically analysed by third party SAT solvers, such as SAT4J [1], MiniSAT [43] and Berkmin [64] and the result of the analysis is displayed to the user. To tackle the state explosion problem [140], the user specifies a scope on the model elements to bound the domain. A scope is a positive integer number, which limits the number of instances of each model element in an instance of the system that is being analysed by the solver. If an instance that violates the assertion (a counterexample) is found within the scope, the assertion is not valid. However, if no instance is found, the assertion might be invalid in a larger scope. For more details on the notion of scope, please refer to [77, Sect. 5]. Another valuable feature of the Alloy Analyzer is its support to debug overconstrained models (this is also known as an UnSAT Core functionality) [126]. In particular, if a model is overconstrained and the analyser cannot find an instance conforming to the model, the tool highlights the conflicting statements that make the model overconstrained. Finally, depending on the SAT solver used for the analysis, the Alloy Analyzer can provide instance enumeration (i.e. each time the analyser produces a different instance that conforms to the model).
One important characteristic of the Alloy language is that it treats scalars and sets as relations. For example, a relation between two atoms \( A1 \) and \( A2 \) is represented by the pair: \{\((A1,A2)\)\}. A set like: \{\(A1,A2\)\} is represented by a set of unary relations: \{(\(A1\)), \(A2\)\}. Finally a scalar, is represented as a singleton unary relation. For example, the scalar \( A1 \), will be represented in Alloy as: \{\(A1\)\}. Treating both scalars and sets as relations, is an interesting property of Alloy, which makes it distinguishable from other popular modelling notations and particularly UML.

Presenting all the details of the Alloy language is out of the scope of this work. The language is thoroughly explained by Jackson in [77]. However, in the following we will describe the small Alloy model of Figure 2.1 to demonstrate the most commonly used features of the language. Line 2 of the model declares an abstract signature called ‘\( Status \)’. An abstract signature is a signature without any direct instances. Line 3 declares a signature that will be represented by a singleton set (notice the literal ‘\( one \)’ before the signature declaration). This signature is called ‘\( Rich \)’ and is a subsignature (i.e. a subset) of the ‘\( Status \)’ signature, because of the ‘\( extends Status \)’ part of the declaration. Similarly line 4 declares a signature called ‘\( Poor \)’. Line 5 declares another signature called ‘\( Person \)’. This signature has two fields, a field called ‘\( stat \)’ (line 6) and another field called ‘\( spouse \)’ (line 7). The ‘\( stat \)’ field declares ordered pairs, whose first coordinate consists of instances of the signature under which the field is defined (i.e. \( Person \)) and second coordinate consists of instances of the \( Status \) signature. The keyword ‘\( one \)’ in the declaration suggests that there is exactly one instance of a \( Status \) related to each \( Person \) (i.e. ‘\( stat \)’ is a function). Moreover line 7 declares that ‘\( spouse \)’ is an ordered pair whose both the first and second coordinates consist of instances of the \( Person \) signature. The ‘\( lone \)’ keyword in the declaration, specifies that there can be a \( Person \) related to no \( Person \), through the \( spouse \) relation (i.e. \( spouse \) can be considered as a partial function).
Line 8 declares a fact, which is a constraint of the system. Lines 9-11 specify the body of the constraint, which is a simple first-order formula. More precisely, the body of the fact specifies that, for each Person $p$ in the system, if $p$ is rich, her spouse has to be rich and similarly if the spouse is rich the Person $p$ has to be rich. Lines 12-15 specify a second fact in the model. The second fact specifies that there are at least two different Persons in the model ($p$ and $p'$), who are both poor and the PoorMarriage predicate holds for those two Persons. The PoorMarriage predicate (line 16) specifies that the two Persons passed as parameters in the predicate, are spouses. Line 17 defines an empty simulation command, which will produce a random instance of the model that conforms to the facts. The ‘for 4’ part of the command specifies the scope for the elements of the system. In particular, the Alloy Analyzer will attempt to produce an instance of the model, using up to 4 atoms for each of the signatures declared in the model. In this particular example the analyser will use a scope of four only for the Person signature, since the Status signature has been specified as abstract (i.e. without any direct instances) and the Rich and Poor signatures are specified as singleton. Finally, line 18 is an assertion, which asks the Alloy Analyzer to confirm that each person cannot be a spouse of herself, for a scope of 4.

Executing the simulation command the Alloy Analyzer produces the random instance of Figure 2.2a. The instance is depicted in the diagrammatic form produced by the Alloy Analyzer. A textual representation of the diagram is the following:

| Person   | { (Person0), (Person1) } |
| Poor     | { (Poor0) }              |
| Rich     | { (Rich0) }              |
| stat     | { (Person0, Poor0), (Person1, Poor1) } |
| spouse   | { (Person0, Person1), (Person1, Person0) } |

Both from the textual and the diagrammatic representations of the instance, we can notice that there are two Persons, Person0 and Person1. Person1 is the spouse of Person0 and both
module example

abstract sig Status{}

one sig Rich extends Status{}
one sig Poor extends Status{}

sig Person{
  stat: one Status,
  spouse: lone Person }

fact who_is_rich{
  all p:Person | (p.stat = Rich => p.spouse.stat = Rich) or
  (p.spouse.stat = Rich => p.stat = Rich) }

fact poor_marriage{
  some p,p':Person | (p!= p' and
  p.stat = Poor and p'.stat = Poor and
  PoorMarriage[p,p'])}

pred PoorMarriage(p,p':Person){ p.spouse = p' }

run {} for 4

check { all p:Person | p.spouse != p } for 4

Figure 2.1: An Example Alloy Model

Persons are Poor. Executing the assertion (line 18 in Figure 2.1) the Alloy Analyzer produces the counterexample of Figure 2.2b. The counterexample depicts a Person, Person0 who is poor and her spouse is Person1. However, the spouse of Person1 is Person1 again. Therefore, the assertion that a Person cannot be a spouse of herself is not valid and additional constraints are required in the model to prevent this scenario.

This simple example demonstrates a subset of the capabilities of the Alloy language and its related tool, the Alloy Analyzer. For more information on the language and the tool please refer to [77, 2].

2.1.3 Model Driven Architecture (MDA)

The Model Driven Architecture (MDA) [51, 87] is a framework proposed by the Object Management Group (OMG) [5], which promotes models as first-class citizens in the software de-
velopment process. Central to the MDA is the idea of a model transformation, which maps models in a source language into models expressed in a destination (or target) language\(^3\). Models in the MDA are instances of MOF metamodels. As depicted by Figure 2.3, an MDA transformation is defined from the source metamodel to the destination metamodel\(^4\). Using this approach every model, which is an instance of the corresponding metamodel, can be automatically transformed into an instance of the destination metamodel. For example, to transform a UML class diagram to Alloy, an MDA transformation that maps the metamodel elements of class diagrams to the metamodel elements of the Alloy language is required.

The OMG specification called ‘Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification’ (or QVT for short) [107] addresses the issue of how to specify transformations on metamodels compliant with the MOF. In particular, the QVT standard proposes two declarative languages (the Relations and Core languages) and one imperative language (the Operational Mappings language). The Relations language is a high level declarative language.

\(^3\)The source and destination languages may be the same. For example, the MDA approach may be used in model refactoring [50].

\(^4\)Transformations mapping elements of the source metamodel into elements of the target metamodel are called unidirectional. If each transformation rule maps not only elements of the source metamodel to the target metamodel, but also the reverse, the mapping is called bidirectional [37]. This work presents a unidirectional transformation from UML to Alloy and as a result we focus on discussing unidirectional transformations in this thesis.
The mappings can be defined using a number of mapping rules, based on pattern matching that define when the transformation rule is executed. The Relations language, specifies both a textual and a graphical notation for defining the mappings. The Core language is a low level language for the definition of a model transformation and is equally expressive to the Relations language [107]. Mappings defined in the Relations language can be automatically transformed into mappings defined in the Core language, using a set of transformation rules from the Relations to the Core language. These transformation rules are part of the current QVT standard [107]. The Operational Mappings language extends OCL with imperative features (such as the ability to define loops). Finally, the QVT standard allows the invocation of Black Box transformations, which can be transformations defined in a language other than the QVT. Black Box transformations define an Application Programming Interface (API) that the QVT transformation may utilise. A QVT tool is a tool that given the source, the target metamodels, the transformation rules and a model conforming to the source metamodel, it can execute the model transformation and automatically produce a target model.

A number of methods for the specification of model transformations have appeared in the literature, mostly while the QVT standard was in the process of being developed [56, 37, 38]. Those methods can be organised in two categories based on their notation; a number of methods use textual notations to define the transformation rules [82, 115, 88], while oth-
ers [142, 12, 40] use graph transformations [27] to define the transformation rules. Please refer to Czarnecki and Helsen [38] for a detailed presentation and rigorous feature-based categorisation of proposed languages and methods to define model transformations. In this work we will use QVT Operational Mappings language for the specification of the transformation rules. Therefore, in the following we will present only the QVT Operational Mappings language in more detail.

The QVT Operational Mappings language uses MOF as a repository for metamodels. It also makes use and extends a subset of OCL (called Essential OCL) [108, Chapter 13] with imperative features (such as ‘while’ loop expressions). The general syntax of a mapping rule defined in the Operational Mappings language is [107, p. 47]:

```plaintext
mapping sourceElement::mappingName(params):returnElement
when {...}
where {...}
{
  init{...}
  population{...}
  end{...}
}
```

The `sourceElement` refers to the element of the source metamodel that the mapping rule will transform. The `mappingName` is the name of the mapping rule, `params` are optional comma separated parameters to the rule and `returnElement` refers to the element of the target metamodel. The `when` clause of the rule is a boolean expression that has to evaluate to true for the rule to execute. The `where` part of the mapping rule defines the conditions that have to be satisfied by the model elements involved in the mapping (i.e. it is a post-condition). The `init` part contains statements that will be executed before the main body of the mapping rule is carried out. The `population` is the main section of the rule that defines how the result of the mapping is populated. Finally the `end` part is executed before exiting the operation.
Apart from the declaration part of a mapping rule, all other parts are optional. It is possible to define the body of a rule, without explicitly defining the population section, in which case it is inferred\(^5\). Finally, it is possible for a rule to inherit or merge one or more mapping rules [107, Sec. 8.1.12].

### 2.2 Related Work

The need for the ability to analyse UML models has been identified from the early days of the UML. This led to the formulation of the precise UML group (pUML) [9]. The purpose of the group was to provide formal semantics for the UML specification. Evans [45] has documented some preliminary results of that effort. Since then, the UML has received wide acceptance in the industry and a number of proposals to formalise various aspects of the UML have emerged. Most approaches propose the manual or automated translation of various UML models (e.g. class diagrams, activity diagrams) to a well-established formalism, such as Z [149], B [10], CSP [70] and Petri-Nets [119].

More specifically, Evans et al. [46] propose the use of Z [149] as the underlying semantics for UML class diagrams. Kim [85] suggests to use Object-Z [127], which is an extension of Z with support for concepts used in the Object Oriented paradigm. In particular, Kim [85] proposes to use UML class diagrams to model the static aspects of a system and UML statemachines to model the dynamics. The author defines mappings from the UML class diagrams and statemachines metamodels to the Object-Z metamodel to achieve the transformation, but to the best of our knowledge no implementation of the transformation has been developed. Roe et al. [121] also suggest to use Object-Z for the formalisation of UML and they specify how to transform OCL into Object-Z. However, existing tools for Z and Object-Z, provide support for theorem proving capabilities, which require special expertise and knowledge to

---

\(^5\)This is the approach we have followed in the definition of the transformation rules in Chapter 4
Marcano and Levy [96] transform UML class diagrams and OCL constraints to the B [10] formal language. The Atelier-B [3] prover is then employed to check the conformance of the operations pre and post conditions to model the invariants of the model. If the prover discovers an error, the user needs to be able to understand the B specification in order to debug the model. Transforming UML models into B is also proposed by Snook and Butler [129]. In particular, the authors propose a UML profile for B (called UML-B) and they have automated the transformation of UML models conforming to the profile, into B in a tool called U2B [128]. Instead of OCL they use μB, a language similar to B, to define constraints and actions. They provide support for UML statemachines and refinement to ensure the generated system conforms to the specification. However, the analysis process is not automated and as the authors state: “The overall lesson is that proof is not easy (even with semiautomatic provers) and if it is to be achieved, consideration must be given to provability in generating the models (whether writing B by hand or translating from UML-B)”. [129]

Richters [120] has provided formal semantics for the UML class diagrams and OCL, based on set theory and first-order logic. His work was incorporated in the UML 2.0 standard [108, Appendix A] and has led to the development of the UML Specification Environment (USE) tool [120], which is an instance evaluator with the ability of simulation. More specifically it is possible to utilise the USE tool to generate snapshots that conform to the model. It is also possible to check if a specific instance of the model conforms to the invariants. This method requires that the instances to be checked are generated manually. To overcome this issue Gogolla et al. [62] suggest a scripting language that automates the process of generating instances. This method can be potentially used to automatically check a large number of instances against the model. In contrast, our approach uses the Alloy Analyzer, which automatically searches the state space exhaustively (up to the user specified
scope), without the requirement to learn and use a scripting language to generate the instances. However, unlike our approach USE provides support for UML concepts, which are not expressible by Alloy and therefore are not supported by our method. For example, USE has inherent support for OCL’s three-valued logic. A comparative study on the capabilities of the Alloy Analyzer and the USE tool has been conducted by Aydal et al. [24].

The KeY tool [15] formalises UML class diagrams and OCL using a variant of dynamic logic [65] and provides an interactive theorem prover environment for the analysis of UML models and their implementation. The tool provides a number of predefined patterns of expressions that can be used to assist the modeller in the specification of the system. KeY also comes with an embedded theorem prover, which provides support for semi-automated execution of proof obligations (e.g. that the conjunction of the invariants of a subclass implies the invariants of the superclass). HOL-OCL [32] is another tool that transforms OCL to Higher Order Logic (HOL) formulas that can be analysed by the Isabelle [102] theorem prover. All these methods require guidance and special expertise to operate the theorem prover environments. Most application developers lack such expertise. Our method, which relies on the analysis capabilities of the Alloy Analyzer and is fully automated, can be used as a first line of defence against flaws in the design of a system. If no counterexample is produced by our method, other techniques based on theorem provers like KeY and HOL-OCL can be used to ensure a property is not violated. Such techniques are more time consuming and require human intervention and expertise. This will save time and resources by using our method to rapidly discover a number of flaws that would otherwise require more time and resources to uncover.

Exploiting Alloy to analyse UML class diagrams and OCL has also been proposed before by Massoni et al. [97]. However, they have specified the mapping rules in natural language, do not provide an implementation of their method and propose a mapping for a small subset
of UML and OCL. Some existing work manually translates existing UML and OCL specifications of systems into Alloy [41, 59]. Dennis et al. [41] use Alloy to expose hidden flaws in the UML design of a radiation therapy machine. Georg et al. [59] have used Alloy to analyse the runtime configuration of a distributed system. Unlike our work, those approaches conduct the translation from UML to Alloy manually, a procedure which is tedious and error prone.

Another line of work compares UML languages and Alloy [76, 66]. In this thesis, to specify the mapping rules from UML to Alloy and justify why a specific UML metaelement maps to corresponding Alloy metaelement, we provide a description of the UML and the corresponding Alloy concepts. In Chapters 4 and 5 we present a comparison between each UML metaelement and the corresponding Alloy metaelement by describing often subtle but important differences. This is similar to the studies on the comparison of languages of UML and Alloy by Jackson [76] and He [66].

Alloy has also been used for the analysis of UML concepts. In particular, Vaziri and Jackson [143] model part of the UML metamodel (with the corresponding OCL well-formedness statements) in Alloy and analyse the constructed Alloy specification. The authors argue that Alloy is simpler to use and understand compared to OCL and present a number of inconsistencies in the UML specification. Zito and Dingel [150] model the notion of UML package merge in Alloy. Their analysis showed that the notion of package merge is not commutative (i.e. merging packages $A$ and $B$ is not the same as merging $B$ and $A$). The subset of UML and OCL we support in our work is extensively used in the UML standard to provide the well-formedness rules for the UML class diagrams. It is therefore possible to use the method presented in this work to automatically transform into Alloy a subset of the UML metamodel, including the OCL well-formedness rules, and reason about the consistency of certain parts of the UML standard. In particular it is possible to reason whether there are any conflicting well-formedness constraints, or whether the well-formedness constraints allow the contraction of
syntactically ill-formed UML models.

Finally, there are a number of UML tools which are oriented towards checking the run-time conformance of an implementation. For example, the Dresden OCL toolkit [72] can generate Java code from UML class diagrams enriched with OCL constraints. The generated code checks at runtime whether the implementation violates any of the constraints. For an extended study of this category of tools the reader is referred to [34]. In contrast to such approaches, our method deals with the analysis of the system at an abstract level before the implementation. As a result, our method can expose bugs and security issues of a system, early in the development process.

Varró [141] developed an approach to formalise UML and other visual languages in general. His approach is based on Graph Transformations [123]. Visual languages can be interpreted as graphs and Graph Transformation approaches in the context of MDA [147, 11] present formal semantics of the languages as algebraic terms and use graph transformations to manipulate those terms [36]. Varró [141] requires to define the formal operational semantics of the visual language to be formalised in terms of Graph Transformation rules. The author represents a model using transition systems and then applies model checking techniques on them.

A number of approaches [101, 44, 67] are concerned with formalising UML state diagrams in the Communicating Sequential Processes (CSP) formalism [70]. CSP provides the mathematical foundations for concurrency modelling and is supported by a number of tools. CSP can used to determine whether properties of systems such as safety and liveness are satisfied. It seems that CSP is best suited for reasoning about UML artifacts, such as sequence and statemachine diagrams.

Petri nets [119] is a very popular graphical formalism for modelling and checking the properties of concurrent, distributed and parallel systems. A Petri net is made of places, to-
tokens, transitions and directed arcs. Places can be interpreted as the states of a system, which hold a number of tokens. Transitions are the transitions that cause the various states of the system to change, by consuming the tokens and the arcs show the direction of the tokens from the states to the transitions and back to the states. One of the proposed extensions of Petri nets, are coloured Petri nets [79]. In coloured Petri nets tokens have colours, allowing them to hold values and making mapping from UML to coloured Petri nets possible. Coloured Petri nets have again been extended to object Petri nets [90], which support object oriented concepts like inheritance and polymorphism. Pettit and Gomaa [117] propose the use of coloured Petri nets to model and validate the dynamic behaviour of UML models and especially architectural models depicted by collaboration diagrams. They propose the translation of each object and message communication in the collaboration diagram to an equivalent coloured Petri net template.
As discussed in Section 1.1, this work utilises an MDA compliant approach, to automate the transformation between UML, OCL and Alloy. The MDA requires a MOF compliant metamodel for the source and target languages (UML, OCL and Alloy). The MOF metamodels of UML and OCL have already been defined by the OMG in the respective specification documents [111, 108]. In this work we use the metamodels of UML class diagrams and OCL. We extend the metamodels, using the profiling extension mechanism provided by UML, in order to forbid the definition of UML models that cannot be represented in Alloy. This is necessary since the UML specification allows concepts such as anonymous classes, which do not have an equivalent notion in Alloy. This has led to the development of a UML profile for Alloy, which is presented in Section 3.1.

The Alloy language is specified in [77, Appendix B]. The syntax of the language is specified in EBNF [14] form. Natural language is also used to define additional syntactic restrictions (e.g. that a signature and its fields cannot share the same declaration identifier). In order to utilise the MDA technology, this EBNF representation of Alloy needs to be converted to a MOF compliant metamodel. The MOF compliant Alloy metamodel is presented in Section 3.2.
3.1 A UML Profile for Alloy

The UML and Alloy are languages for the modelling of systems, but with different philosophies. Such differences between the UML languages and Alloy have been well documented [76, 21, 66]. More specifically, Alloy is a language designed for abstract modelling of systems and supports fully automated analysis. To achieve this, the language is fully declarative, has simple set theoretic semantics and is based on first-order relational logic [75]. On the other hand the UML, as the name reveals (i.e. Unified) is a family of languages developed for the modelling of a diverse range of systems. For example, it can be used to capture user requirements, model business processes, or represent the architecture of Object Oriented systems [116, 49]. The flexibility of the UML can be attributed to its extension mechanisms that allow to easily adapt the language to specific domains, such as modelling Enterprise Java Beans (EJB) [13], embedded systems or business processes. In order to adapt the UML to different domains, the UML standard defines two extensibility mechanisms [109, Chapter 7.1]. The first mechanism, leads to the generation of new languages, by reusing the UML Infrastructure specification [109]. The shortcoming of this method is that the new language that will be produced from this process, will not be supported by existing UML CASE tools. The second extensibility mechanism of UML is the definition of UML profiles [109, Chapter 13]. A UML profile is a special kind of a UML package that can be used to define a number of stereotypes, which are used to provide additional semantics to existing elements of the UML metamodel. Since most existing UML CASE tools provide support for profiles, we chose to extend UML using this approach.

In this work we are extending the UML by developing a UML profile for Alloy, in order to bridge the differences between the two languages. Figure 3.1 shows an outline of our approach. The UML profile we are defining is making use of the UML and OCL metamodels.
In particular the UML profile for Alloy extends the UML superstructure metamodel [111, Chap. 7] for class diagrams, by defining additional constraints on the UML metaelements. The additional constraints are used to rule out UML models that cannot be represented in Alloy. For example, Alloy requires that all model elements have a unique name, while the UML standard allows the definition of unnamed model elements [111, Sec. 7.3.3]. Consequently, our profile requires all UML model elements to have a name. This is depicted in Figure 3.2, where the metaclass *NamedElement* is augmented with the OCL constraint, which states that it is required by all *NamedElements* in a UML model to have a name.

The UML is a general purpose language, which specifies a number of diagrams that allow a modeller to depict different views of a system (e.g. interactions between the different components, the different states various components of a system can be in, or the sequence
of actions of different components). Since Alloy is ideal for analysing structural properties of systems [74], our work focuses on the analysis of static UML models of a system. Therefore we need to select a part of the UML metamodel, which is expressive enough to model basic class diagram concepts (such as Classes, Attributes, Associations, Generalization relations, etc.). This subset is provided by Part I of the UML superstructure specification [111]. More specifically, Section 7.3 of the UML superstructure defines the abstract syntax for class diagrams. The UML metamodel for class diagrams is specified by the UML metamodel, in Figures 7.3-7.18 of the superstructure specification [111] (Root, Namespaces, Multiplicities, Expressions, Constraints, Classifiers, Features, Operations, Classes, Datatypes and Packages diagrams). The next section presents the most important restrictions of our profile on UML elements.

3.1.1 Constraints of the Profile

This section presents a gentle introduction on the additional constraints on the UML metamodel elements imposed by our profile, in order to rule out UML models which cannot be represented in Alloy. For a more detailed description of the UML metaelements and the constraints imposed on the UML metamodel by our profile, please refer to Appendix E. A number of UML class diagram constructs cannot be directly represented in Alloy. Therefore our profile restricts the UML metamodel with additional OCL constraints to disallow the definition of such constructs. In particular, the UML notion of attribute and operation redefinition [111, Sec. 7.3.3] is not allowed, since it cannot be directly represented in Alloy. Moreover the UML profile for Alloy does not allow the definition of the notions of package merge and package import [111, Sec. 7.3.40].

Additionally the UML standard allows for a class to have a number of nestedClassifiers [111, Sec. 7.3.7]. This allows the definition of inner classes. In Alloy inner classes
cannot be directly represented. Consequently, the UML profile for Alloy does not allow the
definition of inner classes.

Another notion, which is not directly supported in Alloy is that of multiple inheritance. The UML profile for Alloy forbids multiple inheritance. For more details on this issue, please refer to Section 4.1.2.

Names

As explained earlier, in UML a NamedElement may not have a name. In Alloy on the other
hand all elements (i.e. signatures, fields) need to have a name. In order to overcome this
difference, our profile requires that all UML model elements have a name. Moreover Alloy,
as a textual language, uses a number of reserved keywords (such as module, sig), which
no user defined model element can use (for a complete list of the Alloy keywords please consult [77, Appendix B.1]). Our profile forbids the definition of UML NamedElements, which clash with the Alloy keywords.

Namespace

All UML model elements are defined in a namespace [108, p. 72]. For example, classes in
a class diagram are defined in the namespace of the package where they belong and class
attributes are defined in the namespace of the class. Model elements of an Alloy model also
belong to a namespace [77, p. 254]. However, the notion of a namespace in Alloy and UML
are slightly different. For example, the UML specification defines that: ‘The set of attribute
names and class names need not be disjoint’ [108, p. 178]. In Alloy on the other hand
signature names, have to be distinct from their field names\(^1\).

In order to avoid issues with the differences of UML and Alloy with respect to the names-

\(^1\)As explained in Sections 4.1.2 and 4.1.3, Alloy signatures correspond to the notion of UML classes and
Alloy fields correspond to the notion of UML class attributes.
tion is attribute and association end names, where different classes that do not belong to the same class hierarchy may share attributes and association ends with the same name.

**Types**

The UML specification defines a number of primitive types (e.g., String, Real, etc.), which can be used when developing UML models. On the other hand, Alloy has a simple type system and the only predefined type it supports is Integers. Therefore, while in UML primitive types and their operations are part of the metamodel, in Alloy they need to be defined explicitly on the model level (i.e., a String has to be declared as an Alloy signature).

In order to bridge this difference, the UML profile for Alloy does not allow the definition of primitive types, other than integers. If a type other than integer is required, it has to be explicitly modelled on the UML class diagram as a `dataType`. For more information and an example of how this can be achieved, please refer to Section 4.2.

Moreover the UML standard allows a `TypedElement` (i.e., a class attribute) to have no type [111, Sec. 7.3.52]. This probably infers an incomplete model, which cannot be automatically analysed. Consequently, to facilitate automated analysis, our profile requires that all UML `TypedElements` have a type.

**Immutability**

An important distinction between UML and Alloy is that UML has inherent notion of states. As a result it is possible for a class attribute to have different values over time. Alloy models on the other hand, are static and do not have a built in notion of statemachines [77, Appendix B.5.1]. An important effect of this difference, is that while in UML class attributes values may change over time, signature fields (which correspond to the UML notion of class attributes) values cannot. In order to reflect this difference, our profile constraints that UML attributes and association ends are `readonly`. This issue and how to model dynamic systems in Alloy is
3.1.2 Stereotypes

In order to carry out fully automated analysis of models, the Alloy language has notions such as the scope, simulation and assertion commands. On the other hand the UML does not have such concepts. Since our work aims to make UML class diagrams fully analysable, using Alloy, we need to extend the UML to introduce concepts such as the scope, simulation and assertion commands. This is achieved with the help of the stereotypes presented in this section.

Figure 3.3 depicts the stereotypes defined in our UML profile for Alloy. The analysis stereotype is used on Packages, scopedElement, singleton and enforce are used on Classes and the assertion and simulation stereotypes are used on Constraints. Finally the dynamic stereotype can be used on Properties.

**analysis**

The analysis stereotype is used on UML packages that are going to be analysed using our method. A UML class diagram is required to have exactly one package stereotyped as anal-
ysis. The analysis stereotype defines three attributes (also called tagged values), the defaultScope, intScope and time. These tagged values are used during the transformation to set the default Alloy scope\(^2\), the scope for integer numbers and the scope for the time\(^3\) respectively. The defaultScope, intScope and time are positive integer numbers. Following Alloy’s approach, if no defaultScope tagged value has been defined in the model, the default value is 3. Similarly if no default tagged value has been defined in the model for integer values, the scope for integer numbers is set to 4 and if no scope for the time has been defined, the scope for time is set to 3.

**scopedElement**

Each class in a class diagram can be stereotyped as a scopedElement. The scopedElement stereotype defines a tagged value (scope), which is used to limit the number of instances of an element when a system instance is being checked by the SAT solver. This is used to override the defaultScope attribute of the analysis stereotype in order to define a different scope for the particular class on which the stereotype is applied.

**singleton**

A singleton stereotype can be applied to a class and is used to define the well known singleton design pattern [55]. Classes annotated with this stereotype, are restricted to have exactly one instance in the model.

**enforce**

In general an instance of a UML class diagram may be partial (i.e. some classes may not have any instances). This stereotype is used on classes that are required to have at least one

\(^2\)The term scope here is used in the context of Alloy [77, Sec. 5.1.2] (i.e. a scope is a number that denotes the maximum number of model elements the Alloy Analyzer will use for the analysis).

\(^3\)The scope for time is used only when modelling dynamic systems. Please refer to Chapter 6 for more details.
instance during the analysis.

**simulation**

In Alloy, a first-order logic statement can be used to simulate a model. This statement corresponds to the Alloy `run` command [77, Section 4.6]. Similarly in a UML class diagram an OCL constraint can be used to simulate the model. We use the `simulation` stereotype for this purpose. More specifically, an OCL statement stereotyped as simulation, will be automatically transformed to an Alloy simulation (`run`) command. An Alloy run command can be used with the Alloy Analyzer to create a random instance of the model that conforms to the statement and the constraints of the model.

**assertion**

Similarly to simulation commands, `assertion` commands can be used to check if a statement that depicts a property of the system, is satisfied by the model. In a class diagram, an OCL statement can be used to depict an assertion. The `assertion` stereotype can be used on OCL statements, which will be transformed to Alloy assertions. It is important to note that OCL constraints annotated with the `simulation` or `assertion` stereotypes cannot have any pre and post conditions.

**dynamic**

As we discussed earlier in Section 3.1.1, the UML profile for Alloy enforces that by default `Properties` (i.e. class attributes and association ends) are readonly. However, often values of properties change over time. Such properties need to be stereotyped as `dynamic`.

The additional constraints applied on the UML metamodel elements and the stereotypes presented in this section, complete the source language metamodel. The target language (i.e. Alloy) metamodel is presented in the next section.
3.2 A MOF Compliant Alloy Metamodel

Alloy is a textual language and its syntax is defined in EBNF [14] form. EBNF stands for Extended Backus-Naur Form and is an extension of the Backus-Naur Form (BNF) notation commonly used to describe the grammar of textual languages [14]. The BNF notation of a language consists of a set of production rules and symbols. Each production rule defines how a symbol on the left hand side of the rule can be replaced by the symbol on the right hand side of the rule.

For example, let us assume the grammar fragment depicted in Figure 3.4. This grammar can be used to represent a binary number. The first production rule (line 1) shows how the number can be produced (i.e. by recursively replacing \( N \) by \( NN \)), while the second rule (line 2) defines the characters (i.e. the numbers 0 and 1) of the grammar. In this grammar 0 and 1 are terminal symbols, since they are part of the string that will represent a statement in the grammar. Likewise the symbol \( N \) is a nonterminal symbol, since it is not part of the string that will represent a statement of the grammar.

\[
\begin{align*}
[1] \quad N &::= NN \\
[2] \quad N &::= 0 \mid 1
\end{align*}
\]

Figure 3.4: The EBNF Representation of a Simple Grammar

Contrary to the domain of compilers, where EBNF is used to depict the syntax of a language, in the MDA the Meta Object Facility (MOF) notation is used to define the syntax of a language. Usually MOF is used to represent the abstract syntax of a language. Since our work uses the Alloy MOF representation as an intermediate step in the transformation,

---

4 In this thesis we use the term 'Metamodel' to denote a MOF representation of the abstract syntax of the language, while the term 'grammar' is used to denote the EBNF representation of the concrete syntax of a language.

5 The EBNF extends BNF with the addition of regular expressions.

6 The MOF representation of a language consists of a UML class diagram that depicts the elements of the language and OCL constraints to represent syntactic well-formedness rules.

---

35
we are only interested in a MOF representation of the abstract syntax of the Alloy language. In the next section we briefly present a survey of existing approaches to represent the EBNF notation of a language using MOF.

### 3.2.1 Existing Approaches

The problem of representing the EBNF notation of a grammar using MOF has already received considerable attention [148, 19, 99]. More specifically Wimmer and Kramler [148] present a semi-automated method to create a MOF metamodel of a textual language from its EBNF specification. Their method creates a metamodel, which has a number of annotations to allow for the automated bi-directional transformation between the MOF metamodel and the EBNF notation. The solution proposed in this work is inspired by this approach. However the annotations used by this approach are part of the concrete syntax of the language and are not required in our method.

Alanen and Porres [19] define two separate mappings, one that transforms the EBNF to a MOF compliant representation and one that transforms the MOF metamodel to the EBNF representation. They employ techniques and conventions on the generated MOF representation, that are not necessary for our work. For example, they follow the convention that *the order between the properties [in the MOF representation] is kept by naming the properties in alphabetically rising order* [19].

Muller et al. [99] investigate the automated transformation between the EBNF grammar of a language and its MOF compliant representation. They define a number of bi-directional transformation rules to automate the transformation between the model and text based representation of the language. However, they do not describe how to acquire the MOF compliant representation from the EBNF notation of a language, which is what we need in our work.

Consequently most existing work is on the (semi)automated transformation between the
EBNF and a MOF representation of a language. Additionally these approaches use annotations and naming conventions for the bidirectional transformation between the EBNF and MOF representations of languages. However, we believe that if OMG’s MOF2Text QVT specification [106] is used, such annotations and naming conventions are not required for the transformation of the EBNF representation (concrete syntax) of a language to a MOF representation (abstract syntax).

### 3.2.2 Proposed Approach

The approach presented in this section facilitates the bidirectional mapping between the EBNF representation of the concrete syntax and a MOF representation of the abstract syntax of the Alloy language. Figure 3.5 depicts an overview of how this is accomplished. In particular, to achieve the EBNF to MOF (EBNF2MOF) transformation we have developed an algorithm that we use to manually develop the MOF compliant representation. To achieve the reverse transformation we use the MOF2Text QVT specification [106]. This approach, unlike existing approaches presented in the previous section, allows us to keep the MOF representation free from elements of the concrete syntax.
EBNF2MOF

In order to generate the MOF compliant Alloy metamodel from the EBNF we have developed an algorithm, which was used to manually construct the Alloy metamodel. Our algorithm comprises of the following steps. How these steps can be used to generate the MOF metamodel, is exemplified later on in this section.

1. The non-terminal symbols of a production rule are represented as MOF classes.

2. If a terminal symbol is an identifier, it will be represented as a MOF class and an id attribute will be added to the class.

3. The optional terminal symbols of a rule are transformed to boolean attributes of the MOF class to which the non-terminal related to the terminal was transformed.

4. The left hand side of a production rule with alternatives is transformed to an abstract MOF class. A number of concrete classes that extend the abstract class are then introduced, to represent each of the alternative choices.

5. Each left hand side of a rule is represented by class according to our algorithm in Step 1. If the right hand side does not have any alternative rules or only terminals, an association is generated between the class representing the left hand side of the rule and the class(es) representing the elements of the right hand side.

The association ends of the classes representing the left hand side of the rule, are non-navigable, while the association ends of the classes on the right hand side are navigable. Multiplicities of the association ends depend on the multiplicity of the symbols in the grammar. For example, if a symbol on the right hand side of a rule is a non-terminal and optional the multiplicity of the association end is 0..1.
If the multiplicity of an association end is zero to many or one to many, it is always represented by an *ordered* association end. The incentive for this is that in some cases we need an ordering over the elements of the grammar. For example, an operation may have more than one parameters, in which case the association end of the parameters is ordered. This allows us to specify, when calling that operation, the order of variables that will be passed as parameters.

6. If all the right hand side parts of a rule are alternatives and each alternative only has terminals as its elements, the class on the left hand side of the rule is depicted as an enumeration class and each alternative terminal is represented as an enumeration literal.

7. Finally elements of the concrete syntax (e.g. braces, parentheses) are not transformed to any element of the MOF abstract syntax.

Applying this algorithm we have been able to manually transform the EBNF of the Alloy grammar to a MOF compliant form. To achieve the reverse (i.e. transform the MOF metamodel back to EBNF) we can use the MOF2Text language, which is an OMG specification [106].

**MOF2Text**

The *MOF Models to Text Transformation Language* [106] is a template based language for the translation of MOF compliant models to a textual notation. A MOF2Text transformation consists of a number of rules, which specify how each model element is mapped to text. The standard [106] addresses issues such as traceability (i.e. which model element generated which part of text) and directing the output of the transformation to text files. We utilise this method in order to define the translation of the MOF compliant Alloy model, back to its textual notation.
In the following we show an example of how the approach developed in this section, can be applied on a subset of the Alloy grammar.

**Example**

Figure 3.6a depicts an excerpt of the Alloy grammar for the declaration of Signatures and Figure 3.6b shows the MOF metamodel generated by applying our EBNF2MOF algorithm. In Figure 3.6a the symbols in bold represent terminals of the grammar, symbols within brackets represent optionals, while the + character represents one or more repetitions of a symbol.

(a) EBNF of a Subset of the Alloy Grammar

(b) Generated MOF Abstract Syntax

Figure 3.6: EBNF Representation and MOF Metamodel of the *sigDecl* Part of the Grammar

Applying rule 1 of the algorithm presented earlier, on the grammar of Figure 3.6a we represent every nonterminal symbol of the grammar (i.e. *sigDecl*, *mult*, *sigRef* and *sigBody*) as a MOF class in Figure 3.6b. Applying rule 2 the terminal *sigId* is represented as a class with an *id* attribute. Applying rule 3 the optional *abstract* terminal symbol is transformed to the *isAbstract* boolean attribute of the *SigDecl*. Applying rule 5 we get the associations between the classes and the multiplicities of the association ends. Finally, applying rule 6 we represent the *mult* symbol as an enumeration.

The metamodel fragment of Figure 3.6b does not forbid the creation of incorrect Alloy models. For example, it allows the definition of two separate signatures with the same id. In
order to avoid such problems, we define a number of well-formedness rules on the generated Alloy metamodel. The well-formedness rules are a number of Object Constraint Language (OCL) constraints on the metamodel. For example, to express that an Alloy model cannot have two signatures with the same id, we define the following well-formedness constraint:

\[
\text{inv: } \text{SigDecl.allInstances() -> forAll}(s1, s2: \text{SigDecl} | \\
\quad s1 <> s2 \implies s1.\text{sigid}.id <> s2.\text{sigid}.id)
\]

Once the MOF metamodel of the Alloy syntax is constructed using the EBNF2MOF algorithm, the inverse transformation (i.e. generation of the Alloy textual notation) can be carried out using the MOF2Text OMG standard [106]. The following depicts the MOF2Text rule that transforms the signature declaration $\text{SigDecl}$ metamodel element of Figure 3.6b, back to the Alloy textual notation.

```plaintext
@code-explicit('')

template public sigDecl2Text(s:SigDecl)
if (s.isAbstract)
  'abstract '
/if

if (s.mult -> notEmpty())
  if (s.mult=mult::ONE)
    'one '
  elseif (s.mult=mult::LONE)
    'lone '
  elseif (s.mult=mult::SOME)
    'some '
  /if
/if

'sig '

for (sid:SigId | s.sigid) separator(',')
  sid.id
```
The first line declares that the rule should output as text (i.e. not attempt to execute) everything that is enclosed in single quotes. The second line declares the name of our template, which will transform a `SigDecl` to text. Lines 3-5 create the `abstract` literal, if the Signature is abstract. Lines 7-15 add the multiplicity markings (i.e. `one`, `lone` or `some`) if any. Line 17 adds the `sig` string and lines 19-21 add the ids the signature declares (`sigId`), one next to the other separated by the comma character. Lines 23-25 create the `extends` part of the grammar, if the signature extends another signature. Finally the `sigBody2Text` template rules is called to transform the signature body (`sigBody`) of the signature to text.

### 3.2.3 Remarks

In this section we have presented a method for the transformation of the Alloy grammar concrete syntax, to a MOF compliant abstract syntax representation and the reverse. The proposed approach was carried out manually, but systematically using the EBNF2MOF algorithm presented above. It has to be noted that our approach has only been applied to the Alloy grammar. Exploring the applicability of this method to other context free grammars is out of the scope of this work.

The well-formedness rules on the generated Alloy metamodel were inferred from the Alloy language reference manual [77]. These rules ensure that the generated Alloy model will be syntactically correct. As explained in Appendix D, the well-formedness rules are
necessary if we want to apply model transformation testing techniques [48, 89] on our model transformation implementation in the future.

An interesting remark is that some of those well-formedness rules cannot be violated when an Alloy model is generated by our UML to Alloy transformation. This is because for most UML metamodel elements, the transformation to the Alloy metamodel elements is homomorphic (i.e. the structure of the model is preserved). For example, a UML class maps to an Alloy signature and a UML subclass maps to an Alloy subsignature, as explained in Section 4.1.2. A well-formedness rule on the generated Alloy metamodel is that an Alloy signature cannot extend itself. Similarly in the UML metamodel a well-formedness rule is that a UML class cannot extend itself. Considering that our UML class to Alloy signatures transformation is horizontal, if a well-formed UML model is provided as input in our UML to Alloy transformation, an Alloy model where a signature extends itself can never be generated.
This chapter presents the transformation rules from the UML class diagrams metamodel elements to the Alloy metamodel elements. For each rule an informal reasoning on why a particular UML concept maps to a particular Alloy concept is provided. The rules are specified using OMG’s QVT standard [107]. For a number of transformation rules some remarks are provided, which highlight often subtle, but crucial differences between the UML and the corresponding Alloy metamodel elements. Finally, in Section 4.2, an example UML class diagram is presented and it is transformed to Alloy using the QVT rules discussed in this chapter.

In this chapter we use the operational mappings part of the QVT specification to define the transformation rules. Operational mappings do not implicitly record trace information of the model transformation (i.e. during the execution of the model transformation the rules do not implicitly record which source elements are mapped to which target model elements). In the definition of the QVT transformation rules presented in this chapter, we do not specify how trace information is recorded; instead Section 7.2 describes how our implementation handles
trace information.

4.1 Transformation Rules for Class Diagrams

The transformation rules presented in this section can be applied to any UML class diagram that adheres to the UML profile for Alloy presented in Section 3.1 and they can be applied on the static elements of a UML class diagram. Our definition of static includes all class diagram elements supported by the UML profile for Alloy, excluding those stereotyped as ‘dynamic’ (i.e. class attributes and association ends, which are not read only) and class Operations. How we deal with those elements is presented in Chapter 6.

Moreover in this section we present the UML and Alloy metamodel elements that take part in the transformation. The metamodels are presented in a more concise form (for example we do not show the complete elements hierarchy, but depict only concrete classes of the metamodel and show all the inherited attributes). Figure 4.1 shows such a simplified metamodel for expressing UML class diagrams. It has been developed taking into account the additional constraints imposed by the UML profile for Alloy discussed in Section 3.1. For example, at most one superclass of a class is allowed to be defined using the profile. This simplified metamodel has been developed only for illustrative purposes to assist the description of the transformation rules presented in this chapter. Similarly Figure 4.2 presents a simplified metamodel to express Alloy signatures. Elaborate versions of the UML and Alloy metamodels are presented in Appendix E and Appendix D respectively.

4.1.1 Package2Modules

A UML package represents a collection of classes, associations, datatypes and other UML elements [111, Sec. 7.3.37]. An Alloy module represents an Alloy model, which similarly to the UML definition of a package, is a collection of Alloy elements (Signatures and Rela-
Therefore a UML package maps conceptually to an Alloy module. For this reason, we transform a UML package to an Alloy module declaration.

Figure 4.1 shows that a UML package is a metaelement, which has a name and contains a number of PackageableElements (e.g. classes). Figure 4.2 shows the part of the Alloy metamodel for the header of the the module declaration. In Alloy a Header is part of a module declaration, which declares a ModuleId.

The following QVT transformation rule depicts the specification of our transformation rule from UML Packages to Alloy modules:

**QVT Specification of Rule:**

---

1We do not take into account concepts such as PackageMerge, since they are not allowed in our profile
This Package to Module Header QVT specification consists of two mapping rules. The `package2header` maps a UML `package` to an Alloy `header` metaelement and the `package2moduleId` maps the `name` of the package to the `id` (the name) of the generated Alloy module. Our cur-
rent transformation supports the translation of one UML Package to an Alloy module at a time. The `package2header` rule is executed if the Package is stereotyped as ‘analysis’.

It is noteworthy that in this case a UML metaelement (i.e. Package) is mapped to two Alloy metaelements, a `Header` and a `ModuleId`. We could have defined a single mapping that accepts a UML Package and generates a `Header` and `ModuleId`. However, we have consistently defined mappings from UML to Alloy using binary mappings (i.e. each rule has a single UML element as input and a single Alloy element as output). This makes the implementation of the transformation rules more convenient as discussed in Chapter 7.

### 4.1.2 Classes2Signatures

In this section we present the transformation rules, which map UML classes to Alloy signatures. This is achieved with two independent transformation rules. The first rule is applied to top level classes (i.e. classes, which are not subclasses of any other class), while the second rule is applied to subclasses.

**Top Level Classes**

**Reasoning for the Rule:**

The UML specification indicates that ‘a class provides a common description for a set of objects sharing the same properties’ [108, p. 178]. Similarly an Alloy signature is a representation of an entity of the system being modelled, it denotes a set of atoms and defines the properties (in terms of relations) of the entity [77, Sec. 4.2]. If we consider that the notion of a UML object maps to the notion of an Alloy atom and that class attributes can be mapped to Alloy fields, we can conceptually map a UML class to an Alloy signature.

The UML standard also allows the definition of abstract classes, which are classes that cannot be instantiated [111, Sec. 7.3.8]. Similarly in Alloy, a signature defined with the abstract keyword, denotes a set that does not have any direct elements [77, Sec. 4.2].
**Rule Description:**

Our rule specifies that every UML class will be transformed to a signature in Alloy (*ExtendsSigDecl* metaelement) with the same name. In the Alloy metamodel the name of the signature is represented by the *SigId* metaclass. The value of the isAbstract attribute of the generated signature, will be the same as the value of the isAbstract attribute of the class. The QVT specification of this rule is provided in the following:

**QVT Specification of Rule:**

```qvt
// This rule is invoked when the Class
// is top level
mapping Class::Class2Sig():ExtendSigDecl
when {self.superClass ->isEmpty()}
{ isAbstract := self.isAbstract;
  sigId := self -> map mapClassName2SigId -> asOrderedSet();
}

mapping Class::mapClassName2SigId():SigId{
  id := self.name;
}
```

**Remark : Object Id vs Atom**

Since a UML class is mapped to an Alloy signature, the instances of a class will map to instances of a signature. In UML a class denotes a set of object identifiers (*object ids*) [108, Ap. A.1.2.1]. In Alloy a signature denotes a set of atoms. Atoms are *indivisible* (they cannot be divided into smaller parts), *immutable* (their properties remain the same over time) and *uninterpreted* (they do not have any inherent properties) [77]. An *object id* in UML is used to uniquely identify an instance of a class, in the same way that an *atom* in Alloy, identifies an instance of a signature. Thus the notion of object id maps to the notion of an atom when dealing with static systems. However, considering the notion of object ids the same as the
notion of atoms, has certain implications when modelling dynamic systems, as described in Chapter 6.

**SubClasses**

This rule is used to transform a subclass (i.e. a class with one parent class) to a subsignature.

**Reasoning for the Rule:**

In UML a generalization hierarchy induces a subset relation on the object ids of the classes participating in the hierarchy [108, Section A.1.2.2] [120, Section 3.4.1]. Figure 4.3a shows an example UML class diagram of a generalization, between a Person, a Student and a Lecturer. Figure 4.3b depicts the interpretation of the instances of this diagram. In this figure the Person class is represented by a number of object ids ($I(Person)$). The object id of the Student class ($I(Student)$) and the Lecturer class ($I(Lecturer)$) are disjoint subsets of the object ids declared by the Person.

![Figure 4.3: Example Generalization Class Diagram and Instance](image)

In Alloy the *extends* keyword can be used to declare a signature that extends another signature (called a *subsignature*) [77, Section B.7.2]. The atoms of each subsignature are a subset of the atoms of the supersignature, which is extended. Subsignatures define disjoint subsets, like the Student and Lecturer subsets in Figure 4.3b. Assuming that for static systems, the notion of an object id maps conceptually to the notion of an atom, we can transform
UML subclasses to Alloy subsignatures.

**Rule Description:**

The transformation is achieved with the QVT rule presented below. The `Subclass2Subsig` mapping, populates the `isAbstract` attribute and `sigId` object of the subsignature (lines 3-4 of the rule). We also populate the `extendsSigRef` association end with a `sigIdRef` object, that references the `SigId` to which the superclass was transformed to (lines 5-6 of the rule).

The operation `getTranslatedSigId` (lines 8-10 of the rule) queries the traces model to find to which `SigId` the superclass of this class was transformed to. We need to use the `first()` operation, because as depicted in Figure 4.3b, the `sigId` associationEnd denotes an ordered collection.

The `getTranslatedSigId` uses the `late resolve` [107, Sec. 8.1.10] clause. This ensures that the query is executed at the end of the transformation, when all classes have been transformed to signatures. This is done to ensure that when the `Subclass2Subsig()` rule is executed on a specific class, the superclass has already been transformed into Alloy. Another method would have been to specify the transformation in two phases. The first pass would transform all classes to signatures and the second pass would update the references of the subsignatures to the supersignatures.

**QVT Specification of Rule:**

```plaintext
mapping Class:Subclass2Subsig():ExtendsSigDecl

when {not self.superClass -> isEmpty()}
{ isAbstract := self.isAbstract;
  sigId := self -> map mapClassname2SigId -> asOrderedSet();
  extendsSigRef := object SigIdRef{
    sigId := self -> getTranslatedSigId();
    
  };
}
```
Remark: Multiple Inheritance

A UML class can extend more than one superclass. For example, in Figure 4.4a an instance of a CameraPhone is an instance of both a Phone and a Camera at the same time [120, Sec. 3.4.1]. More formally, as defined by Richters [120] and depicted in Figure 4.4b:

\[ I(\text{CameraPhone}) = I(\text{Phone}) \cap I(\text{Camera}) \]

where \( I(\text{CameraPhone}), I(\text{Phone}) \) and \( I(\text{Camera}) \) denote the set of object ids of the instances of the CameraPhone, Phone and Camera classes respectively.

Using our transformation rules, the diagram of Figure 4.4a will be transformed to an Alloy model with three signatures, a Phone, a Camera and a CameraPhone signature. By definition top level Alloy signatures (like Camera and Phone) define disjoint sets. More formally:

\[ I(\text{Phone}) \cap I(\text{Camera}) = \emptyset. \]

It is therefore evident that in this case: \( I(\text{CameraPhone}) = \emptyset \). Therefore multiple inheritance cannot be directly represented in Alloy, in a general way. In some specific cases, however, multiple inheritance can be represented in Alloy, as demonstrated by Jackson [77, p. 94]. More specifically, if the classes participating in the multiple inheritance have a common superclass (as in the so-called ‘diamond’ multiple inheritance), they can be expressed in Alloy. Nevertheless, this case of multiple inheritance is not generic enough to be incorporated in our method. Since no generic way exists to support multiple inheritance in Alloy, our current profile forbids the existence of UML models with multiple inheritance (see Section 3.1.1).

Remark: Abstract Classes vs. Abstract Signatures

Even though in general the notion of abstract classes maps to the notion of abstract signatures, there is a minor difference. In particular in Alloy if an abstract signature is not extended by
another signature, it is not considered to be abstract and can have direct instances. According to Jackson [77, page 268], the reason for this design is that an abstract signature without any subsignatures is probably an indication of an error or an incomplete model. Therefore the Alloy Analyzer does not treat abstract signatures without any subsignatures as abstract. On the other hand in UML an abstract class not extended by other classes is still considered abstract. Consequently, to deal with this difference our profile does not allow the definition of UML abstract classes which are not extended by any subclass.

4.1.3 Properties2Fields

In this section we specify how a UML property metaelement is mapped to an Alloy field. The UML metaelement property is used to represent a class attribute or an association end. In the following we provide an informal reasoning on why attributes and association ends can be mapped to an Alloy field.

**Reasoning for the Rule:**

**Attributes:** A UML attribute is a function that maps the type of the class to which the attribute belongs, with the type of the attribute [120, Sec. 3.3.3]. Assume the Phone class of Figure 4.4a. According to Richters [120], the pin attribute is defined as:

\[ \text{pin} : \text{Phone} \rightarrow \text{Integer} \]
An Alloy field denotes a relation between the set of atoms of the signature where the field is declared, to the set of atoms of the type of the field. Since a function can be expressed as a relation, it is possible to define the notion of a UML attribute in Alloy. More specifically the *pin* attribute of Figure 4.4a can be represented by the following Alloy field declaration:\(^2\):

\[
\text{pin} : \text{Phone} \rightarrow \text{one Int}
\]

This declaration specifies a relation called *pin*. The *one* keyword in the declaration constrains the relation between the Phone and the Integers to be a function. Such an Alloy declaration is interpreted on the instance level by a set of ordered pairs, where in the first coordinate are instances (i.e. atoms) of Phone and in the second coordinate are instances of Int. More specifically, some example instances of pin could be:

\[
\text{pin} = \{(\text{Phone0}, 13), (\text{Phone1}, 21), (\text{Phone2}, 17)\}
\]

**Association Ends:** An association end can be represented as an attribute of the class, which owns the association end. The reason for this is that according to the UML specification, a navigable association end owned by a class is also an attribute of that class [111, p. 41]. Since an association end can be regarded as an attribute, using a discussion similar to the above, we can assume that an association end is a relation between the class that owns the association end and the class where the association end is connected.

In the following we provide a description and the QVT rule of the transformation of UML properties to Alloy fields.

**Rule Description:**

Lines 2-3 of the QVT rule presented later on ensure the rule is executed when the property being mapped is not an aggregation or composition or stereotyped as ‘*dynamic*’. This

\(^2\)Please note that in Alloy the arrow is used to denote a relation rather than a function.
is because our work does not support aggregation and composition, as discussed later on. Moreover the ‘dynamic’ stereotype is used for mutable properties and is treated differently, as discussed in Chapter 6. Line 6 of the rule uses the PropName2VarId mapping to map the name of the property to the varId of the DeclSetExp that is generated. The varId is in essence the name of the field being created. Line 7 of the rule maps the property multiplicity to the field multiplicity as we will see in the following paragraph. Finally lines 9-17 map the type of the property to the Alloy expression (expr), that defines the ‘type’of the generated field. If the type of the property is a user defined type (i.e. a Class), then we use the getMappedSigId() mapping (lines 21-22) to acquire a reference to the sigId the user defined class was mapped to.

**QVT Specification of Rule:**

```plaintext
mapping Property::Property2Field():DeclSetExpr
{
  when self.aggregation = AggregationKind::none and
  not isStereotypedBy(self, "dynamic");
}

varId := self -> map PropName2VarId() -> asOrderedSet();
multi := self -> map UmlMulti2AlloyMulti();
expr := object SigIdRef {
  if (self.type -> oclIsTypeOf(Class)) then
    sigId := self.type -> getMappedSigId();
  else if self.type.name = "Integer" then
    sigId := object SigId {
      id := "Int";
    };
  end if
}

mapping Peroperty::PropName2VarId():VarId{
```

55
20  id := self.name; }
21  query Class::getMappedSigId(): SigId{
22    return (self -> late resolveOne(ExtendsSigDecl)).sigId -> first()}

The property UML metamodel element, has a number of metaattributes that define special characteristics of property instances. For example, a property has a multiplicity and it can be an aggregation or a composition. The following describes how these property metaattributes are treated by our transformation.

Multiplicity

The UML superstructure specifies that a Property is a StructuralFeature, which in turn is a subclass of a MultiplicityElement [111, Sec. 7.3.44]. Therefore an attribute or an association end in a UML class diagram has an implicit notion of a multiplicity.

Reasoning for the Rule:

The UML multiplicity is comprised of two integer numbers, (lower and upper) that define the lower and upper bound of the cardinality of the collection that will represent the instances of the class. In UML, typically four types of multiplicities are commonly used, at most one (0..1), exactly one (1..1), at least one (1..*) and any (0..*).

Alloy provides a number of keywords to define the multiplicities of relations. The keywords, one, lone, some and set can be used, when defining signatures or fields [77, p. 95]. The keyword one restricts the set to have a cardinality of exactly one. The keyword lone restricts the cardinality to at most one. The keyword some restricts the cardinality to be at least one, while set defines a cardinality of zero or more. It is therefore possible to use these four keywords in Alloy, to define the common four types of UML multiplicities. The following QVT rule depicts how the four most common cases of multiplicities described above, are handled.
QVT Specification of Rule:

```qvt
mapping Property::UmlMulti2AlloyMulti::Multiplicity{
    if (self.lower=0 and self.upper=1) then
        return Multiplicity::LONE;
    else if (self.lower=1 and self.upper=1) then
        return Multiplicity::ONE;
    else if (self.lower=1 and self.upper=-1) then
        return Multiplicity::SOME;
    else
        return Multiplicity::SET;
    endif
    endif
    endif
}
```

Aggregation and Composition

In UML special kinds of binary association exist to denote a Whole-Part Relationship (WPR) [23] between classes. More specifically the UML provides the notions of aggregation (shared aggregation) to denote weak ownership and composition (composite aggregation) to denote strong ownership. The exact meaning of aggregation and its difference from composition has received considerable attention. Fowler refers to aggregation as ‘one of the most frequent sources of confusion’ [49, p. 67] and Rumbaugh et al. suggest to ‘think of it as a modeling placebo’ [124, p. 148], while Henderson-Sellers and Barbier ask: ‘what is this thing called aggregation?’ [68]. Moreover the UML standard specifies that: ‘Precise semantics of shared aggregation varies by application area and modeler’ [111, Sec. 7.3.2 ].

It is therefore evident that there is not a clear view on the exact semantics of aggregation and composition amongst the UML community. However, despite the different interpretations, there are some primary properties of aggregation and composition that most researchers agree on [61, 124]. More specifically the weak ownership semantics of aggregation do not
allow self references. The strong ownership semantics of composition, in addition to not allowing self references, impose that in a WPR, the part can exist only if an instance of the whole exists and two wholes cannot share the same part(s).

These rules have been formalised by Gogolla and Richters in [61]. The authors present a methodical way of refactoring aggregations or compositions as standard binary associations enriched with OCL constraints to depict the additional semantics of aggregation and composition. This methodology is well suited for our transformation to Alloy, since we have already defined the transformation rules for binary associations and a subset of OCL. As a result we have not explicitly defined transformation rules for aggregation and composition. We require that they are expressed as standard binary association and OCL is used to capture the additional semantics. In order to forbid self references, an OCL transitive closure operator is needed as described in page 114.

Subsets

A UML Property that belongs to a Subclass, may subset one or more properties of one or more of its superclasses. This corresponds to set-theoretic concept of subsetting [111, Sec. 7.3.3].

In Alloy it is possible to represent the subsets relations, using additional constraints. For example assume the UML model of Figure 4.5a. This is transformed to the Alloy model of Figure 4.5b. Notice that the constraint in line 7 of the Alloy model requires that all elements of the firstyearattend relation belong to the set of the attend relation.

Redefinition

A property of a subclass can redefine one or more of the properties of one or more of its superclasses [111, Sec. 7.3.44]. The redefinition is used ‘...to augment, constraint or override

\footnote{In a Whole-Part Relationship, the whole cannot be part of itself.}
the specification as it applies specifically to instances of the specializing classifier’ [111, p. 126]. A redefining property has usually the same name as the redefined property. The notion of redefinition corresponds to the well known concept of operation and attribute overriding in Object Oriented programming.

Alloy does not directly support the notion of redefinition. More specifically signatures that belong to the same hierarchy may not define fields with the same name. This is also the stance the UML formal semantics [108, p. 182] takes on this issue. Our approach follows the UML formal semantics view on the issue and as discussed in Section 3.1.1, we do not allow UML class diagrams with redefinition in our approach.

It can be argued that this constraint of the Alloy language may limit considerably the domain of Object Oriented models, which can be analysed by our approach. Indeed attribute overriding is an important aspect of Object Oriented modelling/programming. However, our experience is that when modelling abstract specifications for the purpose of analysis, property overriding is often not required.

isOrdered and isUnique

The UML metaelement Property is a MultiplicityElement [111]. As such, an attribute or an association end can be ordered and/or unique. By default properties of classes are unordered and unique. This means that the order of the instances of a property is not predetermined.

![Diagram](image)

(a) Subsetting

![Alloy Representation](image)

(b) Alloy Representation

Figure 4.5: Example Subsetting Property
and an instance of a property cannot be related to the class more than once. Depending on the combination of the values of the isOrdered and isUnique metaattributes, a property may denote different types of collection, as depicted in Table 4.1.

Properties with $\text{isOrdered}=\text{false}$ and $\text{isUnique}=\text{true}$ correspond to a Set. Since an Alloy Field of a specific signature instance denotes a set, it is possible to directly represent an unordered and unique property in Alloy. Our profile only allows the definition of attributes and association ends, which are unordered and unique. The rest of the combinations depicted in Table 4.1, can be represented in Alloy, but this brings additional challenges to the transformation as discussed in Section 5.7.

isDerived

A UML Property may be derived, i.e. its value depends on the value of other elements in the class diagram. Usually in such cases OCL is used to specify how the value of a derived property can be computed [111, p. 14]. Using our OCL to Alloy transformation rules, it is possible to express derived attributes in Alloy.

isDerivedUnion

The value of a property may be restricted to be the union of the values of the properties that are annotated as subsets. Assume the UML diagram of Figure 4.6a, depicting a simplified model of a two year study college. The modules a student can attend are the derived union of the module (s)he can attend in the first year and the second year. Figure 4.6b depicts
an Alloy model representing the class diagram. Lines 10-13 depict the ‘subsets’ semantics of the firstyear and secondyear properties of the Student. Line 13 depicts the fact, that the attend Property of the Student is a derived union of the firstyearattend and secondyearattend subsets.

```
[1] sig Student{
[2]  attend: set Module}
[3] sig Modules()
[4] some sig FirstYearStudent extends Student{
[5]  firstyearattend: set FirstYearModule}
[6] sig FirstYearModule extends Modules()
[7] sig SecondYearModule extends Modules()
[8] sig SecondYearStudent extends Student{
[9]  secondyearattend: set SecondYearModule}
[10] fact {firstyearattend in attend}
[11] fact {secondyearattend in attend}
[12] fact {
[13]  attend = firstyearattend + secondyearattend}
```

(a) Subsetting  
(b) Alloy Representation

**Figure 4.6: Example Derived Union Property**

**isReadOnly**

If a property is marked as readonly, once a value is set for that property, its value cannot change. As we have discussed earlier on (see Section 3.1.1), our profile imposes that by default a property is readonly. This reflects the fact that Alloy fields are immutable (i.e. their value cannot change over time). As a result if a property is readonly it will be treated in the way it is presented in this section. If the property is not readonly, it will be handled in the way presented in Chapter 6.

**Default**

A UML property may have a default value. The default value can be a ValueSpecification (for our profile this is either an integer number if the type of the attribute is integer, or an OCL statement that describes how the default value is calculated). Default values are treated by our transformation so that the OCL statement that expresses how the default value is calculated,
is transformed to an Alloy fact. The OCL statement is transformed using the transformation rules presented in Chapter 5.

**Static**

A property may be defined as *static*. If a property is static, it is part of the specification of the classifier (i.e. class) to which the property belongs, rather than part of the instances of the class [111, Sec. 7.3.19]. This is well suited for models that represent source code, where static attributes and methods are often used.

Our transformation deals with static properties by transforming them to Alloy fields like any other property, since the notion of a static field does not exist in Alloy. However, an additional constraint is inserted in the produced Alloy model, to ensure that all instances of the signature to which the field belongs have the same value of the field. This interpretation is not exactly equivalent with the UML point of view that the field belongs to the class rather than the instances of the class, but is still consistent with the fact that all possible instances of the class have the same value of the field (even though an instance of a class is not necessary in order to access the value of the field). Consequently the results of the analysis in Alloy will not be affected by this transformation decision.

**isLeaf**

A property can be defined as a *leaf*. If this is the case, the property cannot be specialised further through redefinition. Since our transformation does not support redefinition, the value of this metaattribute is ignored.

**4.1.4 Associations**

An *Association* in a UML class diagram is used to depict a relation between classes. An association between two classes, is called *binary*. If more than two classes participate in the association, the association is called *n-ary* (e.g. if three classes participate, the association
is called ternary). As discussed in Section 3.1.1, our UML profile for Alloy only allows the definition of binary associations. In the following paragraphs we discuss how binary associations are transformed to Alloy and provide guidelines, based on existing research, on how n-ary Associations may be refactored to binary and hence transformed using the presented transformation.

**Reasoning for the Rule:**

From a simple set-theoretic point of view an association denotes a set of tuples relating the classes that the association connects [111, p. 37]. As we have already discussed, in Alloy a field denotes a set of tuples of the signatures that participate in the relation [77, Sec. 3.2.2]. Considering that classes map to signatures (see Section 4.1.2), it is obvious that the notion of an association maps conceptually to the notion of an Alloy field. However, while the UML has two notions to depict relations between classes (i.e. association end and association), Alloy has one (i.e. field). Consequently we are transforming only association ends to Alloy fields. Associations are mapped to Alloy facts, that express the semantics of bidirectionality and association end multiplicities. In the following paragraph we show how this is achieved.

**Example:** Assume the example UML class diagram of Figure 4.7a, where one instance of class A is related to zero or more instances of class B. Using the Classes2Signatures and Properties2Fields transformation rules presented in sections 4.1.2 and 4.1.3 respectively, lines 1-4 of the Alloy model of Figure 4.7b are generated.

However, the association between A and B in Figure 4.7a is bidirectional (i.e. if an instance of A is related to an instance of B, the instance of B has to be related to the same instance of A). Moreover the association end multiplicities restrict that one A can be related to a set of Bs and each set of Bs has to be related with exactly one A.

The bidirectionality fact is expressed in line 5 of the generated Alloy model of Figure 4.7b, which restricts that the a relation is the transpose of the b relation. The multiplicity constraints
imposed by the association ends, are shown in lines 6 and 7. In the following we provide an informal description of the QVT rule that is used to generate the bidirectionality fact.

(a) UML Association

```
[1] sig A{
[2] b: set B}
[3] sig B{
[4] a: one A }

// Relation b is the transpose of a
[5] fact{a = ~b}

// Multiplicity fact for b
[6] fact{b in A one -> set B}

// Multiplicity fact for a
[7] fact{a in B set -> one A}
```

(b) Alloy Representation

Figure 4.7: Example Association End Representation in Alloy

**Association Bidirectionality Rule**

**Informal Rule Description for Bidirectionality:**

Figure D.2 depicts the Alloy metamodel part for expressions. If an Association has two association ends (Properties), which are both navigable, then a symmetry fact is generated in Alloy. This is expressed in lines 2-3 of the QVT rule depicted in the following. To determine if an association end is navigable, the UML specification provides a metaoperation called `isNavigable()`, which returns true if an instance of the metamodel element `property` is navigable. The specification of the `isNavigable()` metaoperation, taken from [111, p. 121] is the following:

Property::isNavigable() : Boolean

`isNavigable = not class->isEmpty() or self.owningAssociation.navigableOwnedEnd->includes(self)`
Lines 6 - 17 of the rule are used to generate the symmetry fact (for the example UML model of Figure 4.7a, the QVT rule would generate the fact of line 5 of the Alloy model of Figure 4.7b). More specifically line 6 of the QVT rule shown below instantiates the `ConstraintSeq` element of the generated Fact (i.e. the Fact body). Line 7 of the rule creates the `CompareOpConstraint` object. The operation of the `CompareOpConstraint` is an `EqualsCompareOperation` (i.e. the constraint is used to express that two expressions are equal). Lines 9-11 of the rule generate the left hand side of the constraint, which is a `VarId`, whose `id` is the name of one end of the bidirectional association. Lines 12-16 generate the right hand side of the `CompareOpConstraint`. The right hand side consists of a `Transpose` operation on the `VarId`, whose `id` is the name of the second end of the bidirectional association.

**QVT Specification of Rule for Bidirectionality:**

```plaintext
mapping Association::Association2SymmetryFact():FactDecl
{}.

{ when self.memberEnd -> size() = 2 and
  self.memberEnd -> forAll(p:Property | p.isNavigable());
}

{ constraintSeq := object ConstraintSeq{
  constraint := object CompareOpConstraint{
    compareOp := object CompareOperation.EqualsCompareOperation{};
    left := object VarId {
      id := (self.memberEnd->first()).name
    };
    right := object UnaryExpression {
      unaryOperation := object UnaryOperation.Transpose{};
      expr := object VarId {
        id := (self.memberEnd -> at(2)).name
      };
    };
  };
};
```
Association Multiplicities Rule  In the following we present the rule for defining multiplicity facts (such as those of lines 6 and 7 in Figure 4.7b).

Informal Rule Description for Multiplicities:

Our transformation rules generate a Multiplicity fact for each navigable association end that participates in the binary association. Lines 4-7 of the following QVT rule use the Property2MultiFact() mapping to create the multiplicity facts for the navigable association ends of a binary association. Lines 9-33 of the rule generate for each association end the actual constraint.

QVT Specification of Rule for Multiplicities:

```qvtRule
1  mapping Association::Association2MultiFact():set(FactDecl)
2    { when self.memberEnd -> size() = 2;
3    }
4    { return self.memberEnd ->
5      xcollect(me | if (me.isNavigable)
6        -> map Property2MultiFact());
7    }
8
9  mapping Property::Property2MultiFact():FactDecl
10    { when self.isNavigable();
11     }
12    { constraintSeq := object ConstraintSeq {
13       constraint := object DeclConstraint {
14         exp := object VarId{
15           id := self.name
16         };
17         declExpr := object DeclRelExpr{
18           mult1 := getOppositeProperty(self)
19           -> map UmlMulti2AlloyMulti();
20           mult2 := self
21             -> map UmlMulti2AlloyMulti();
22         }
23       }
24    }
25```
Remark: N-Ary Associations

The interpretation of n-ary associations in UML is particularly problematic. More specifically, Génova et al. [57] expose the issue of interpreting the minimum multiplicity of n-ary associations, while Richters [120] discusses the lack of expressiveness in OCL to take full advantage of the meaning of n-ary associations. Here we discuss those issues in more detail and present how n-ary associations can be refactored to binary, as suggested by Gogolla and Richters [60].

Figure 4.8a presents an example ternary association taken from Génova et al. [57]. The association specifies that an Employee can participate in a Project, using at most one Skill. The 0..1 multiplicity of the association end on the Skill has three possible interpretations.
according to Génova et al. [57].

The ‘actual tuples’ interpretation states that the only possible tuples that can participate in the association have at least one skill. Assume the example tuples in Table 4.2. Since all instances of the ternary association are triples, the only allowed instances are the ones shown in Table 4.2 (i.e. if John participates in a project Databases, he has to use a skill).

According to the ‘potential tuples’ interpretation, the possible instances of the association are the cartesian product of the instances of the pair Employee-Project. Some of those instances are related to a skill (i.e. the John-Java pair is related to the Programming skill, while the John-Database pair is related to no skill). However using this interpretation, if the minimum multiplicity on the Skill was 1, all possible pairs of Employee-Project should be related to a skill. In this case, Table 4.2 should have 8 rows (i.e. the cartesian product of the two Employees and the four Projects).

The third interpretation is the ‘limping links’, where a pair Employee-Project may exist without a Skill (i.e. the instances are no longer triples, but can be ordered pairs if there is no Skill related to the Employee-Project pair). This implies the existence of an implicit binary association between Employee and Project, as explained by Génova et al. [57].

In addition to the ambiguity in the meaning of the minimum multiplicity of n-ary associations, OCL’s treatment of n-ary associations is also problematic. In particular as Richters presents, the way OCL navigation expressions treat n-ary associations is equivalent to binary associations [120, Sec. 4.9.2]. Assuming the following OCL expression applied on the person John of Table 4.2:

```
context Employee
self.project
```

The statement will return the set: \{Java,Cobol,UML\}. It is not possible to view with an
OCL navigation expression which skills were used for which project, unless structures such as tuples and sequences are used by the modeller.

On the other hand, Alloy’s relational logic, provides the means for easy manipulation of ternary relations. In particular in Alloy the association of Figure 4.8a between the Employee, Project and Skill, could be expressed as:

```alloy
sig Employee{
    assoc: Project set -> set Skill
}
```

The relation `assoc` is a triple of the type: `{(Employee,Project,Skill)}`. The Alloy expression: `Employee.assoc`, will return ordered pairs of type: `{(Project,Skill)}` that associate which Skills were used for which Project. It is therefore evident that unlike OCL, Alloy does not treat n-ary associations as binary.

In order to avoid such issues, we require that all ternary relations are refactored to binary ones, using the method suggested by Gogolla and Richters [60]. More specifically, the ternary association of Figure 4.8a can be refactored to the equivalent model of Figure 4.8b. The ternary association is replaced by the class `Association`, which in turn is related to the other classes with binary associations. In this diagram the following OCL constraint is required to forbid an association to link the same Employee, Project and Skill more than once:

```ocl
Association.allInstances() ->
    forAll(a,a’:Association |
        (a.employee=a’.employee and
        a.project=a’.project and
        a.skill=a’.skill) implies a=a’)
```

69
Using this approach we also avoid the minimum multiplicity issue described in this section. The modeller can use multiplicity annotations (and OCL statements if required) on the familiar binary associations, to express the desired multiplicity constraints.

However, it has to be noted that even though the diagrams of Figures 4.8a and 4.8b are semantically equivalent, there is loss of conceptual information about the domain being modelled during the transformation. More specifically, Figure 4.8a clearly shows that there are three classes participating in the association, while in Figure 4.8b there are four classes, with the Association class being an ‘artificial’ class, used to depict the semantics of the original ternary association. This method is well-known in the domain of relational databases, where a link table is used to connect more than two related entities [39].

![Diagram](image)

(a) An N-Ary Association          (b) Equivalent Binary Association

**Figure 4.8: Transforming N-Ary Associations to Binary**

**Remark: Association Redefinition and Specialization**

In the UML metamodel an Association is a Classifier. As such, it can be redefined and specialized. As we have explained earlier, Alloy does not directly support the concept redefinition. Thus, our profile does not allow the concept of Association redefinition and specialization.
4.1.5 Types

Both the UML and OCL are typed languages. Every UML model element and OCL expression denotes a type [120]. On the other hand Alloy has a very simple type system, which is based on the set theoretic semantics of the language [135]. While the UML provides a number of predefined (primitive) types (such as String, Integer, Real), Alloy only provides inherent support for Integer types. In this section we describe how DataTypes and Primitive types are transformed to Alloy.

DataTypes

UML Datatypes are like UML classes, with the exception that they are identified by their value, rather than than the object id [111, Sec. 7.3.11]. So it is not possible to have two separate DataTypes with the same value. Therefore a UML Datatype can be treated like a UML class during the transformation to Alloy, with the additional constraint that no two Datatype instances can have the same value.

More specifically, in our transformation, all model elements with the ‘DataType’ stereotype are transformed to Alloy signatures (like classes). Additionally the attributes of the DataTypes (if any) are transformed to Alloy fields (just like attributes of classes). However on the Alloy model we add a constraint that two DataTypes instances cannot have the same attributes values.

For example, assume the DataType depicted in Figure 4.9a. This DataType represents a Point in a two dimensional space. The Point has two coordinates, $x$ and $y$. From the definition of the DataType [111, Chapter 7.3.11], if two points have the same attribute values, they are indeed the same point. As a result the UML model of Figure 4.9a is transformed to the Alloy model depicted in Figure 4.9b. The fact of the model expresses that if two distinct points have attributes with the same values, they are the same point.
**Primitive Types**

In general Alloy does not provide inherent support for UML primitive types. Even though primitive types are not supported, it may be possible to simulate them, on the model level. For example, a *String* datatype may be defined on a UML model and used when an attribute of type String exists in the model. However, since no generic method exists for representing UML primitive types in Alloy, the UML profile for Alloy does not allow any primitive types, apart from Integers.

If primitive types are required in a model, they have to be explicitly modelled on the model level, as a DataType. For more details on how Integer types are handled by our transformation, please consult Section 4.1.6

**Remark**

A method of representing String types in Alloy, is by expressing them as a sequence of characters. Appendix A provides a small example and a discussion of how String types and operations may be simulated with Alloy. Incorporating this preliminary work into our model transformation remains for further research.

### 4.1.6 Integers

The mapping of UML Integer types to Alloy is carried out in the following manner. If a UML property is of type *Integer*, an Alloy field of type ‘*Int*’ is generated. The way Alloy handles

---

![Figure 4.9: Representing UML Datatypes in Alloy](image)

---

72
Integers is explained by Jackson [77, Sec. 4.8]. In particular, Alloy has two kinds of Integer representations. An Integer atom, which can be used like any other signature (i.e. it can participate in relations) and an Integer value, which is the actual value of the Integer atom. This approach is similar to the correspondence between an Integer object and the integer primitive type in Java.

While the keyword ‘Int’ is used to refer to Integer atoms, the keyword ‘int’ (notice the first letter is lowercase), is used to refer to integer values. It is possible to ‘cast’ between an Integer atom and its value, using the int and Int keywords respectively. The modeller does not need to be concerned with this issue, since the latest version of the Alloy Analyzer automatically casts between Integer atoms and values depending on the context [2].

**Remark: Scope and Integers**

As we have discussed in Section 2.1.2, for the analysis of a model, a scope needs to be specified, which is the number of elements the Alloy Analyzer will use to carry out the analysis. A scope can also be specified for integer numbers in the model. The scope for integer numbers, is a number that denotes the bitwidth that will be used by the Alloy Analyzer to represent the number. For example, a scope of 4 for integer numbers, can be used to represent $2^4 = 16$ numbers. Since integer numbers in Alloy are signed, a scope of 4 is used to represent numbers in the range from -8 to +7.

Therefore it is important for the user to specify the scope for integer numbers with great care. A guideline would be to specify the scope for integer numbers so that the scope covers the minimum and maximum expected integer numbers in the model. However, it is extremely important to note that the Alloy Analyzer performs silent overflowing on integer number expressions. For instance if the scope for integer numbers is set to 3, the following OCL expression: $2 + 2 < 3$ will be true! This is because $2 + 2 = 4$, but since the maximum number that can be represented is 3, the expression $2 + 2$, will silently overflow and will return -4.
sig Person{ age: one Int, income: one Int }

all p:Person | p.age > 0 && p.age <= 120

fact{ all p:Person | (p.age > 40) => p.income > 16000 }

run { some Person} for 6 but 15 int

Figure 4.10: A Simple Alloy Model with Integer Values

Obviously \(-4 < 3\), so the expression: \(2 + 2 < 3\), will return true.

Moreover Alloy’s performance greatly depends on the size of the Integer values we have in the model. For example assume the simple Alloy model depicted in Figure 4.10. The model defines a signature called Person, with two integer fields, age and income. There is a constraint that the age of all Persons in the model is greater than 0 and less than 120 years old. Additionally the income of Persons whose age is greater than 40 years old is more than 16000. We asked the analyzer to find a random instance of this model for 6 Persons and a scope for integers of 15 (i.e. the range of integer numbers is: \(-16384\) to \(+16383\)). The Alloy Analyzer could not find an instance satisfying this model after 14 hours and we aborted the execution.

This example shows that using a large scope for integer numbers may require enormous computational power, even for simple models. In general as Jackson [77, Sec. 4.8] explains, it is recommended to avoid the use of integer numbers in an Alloy model where possible. This is also discussed in the example of Section 7.3.1.

4.1.7 Enumerations

An Enumeration is a Datatype, which consists of a number of user defined Enumeration Literals. An instance of an Enumeration can be any of the Enumeration Literals [111, Sec. 7.3.16]. A UML Enumeration is transformed to an Alloy abstract signature and the Enumeration Literals are defined as singleton sets that extend the abstract signature. For example, the simple
UML Enumeration of Figure 4.11a is represented by the Alloy model of Figure 4.11b.

![UML Enumeration](image1)

![Alloy Representation of Enumeration](image2)

(a) A UML Enumeration  
(b) Alloy Representation of Enumeration

Figure 4.11: Representing UML Enumerations in Alloy

### 4.1.8 Association Classes

A UML *AssociationClass* represents a model element, which is both an association and a class [111, 7.3.4]. For example, let us assume the class diagram of Figure 5.1. A Person may get married at a specific place on a specific date. The place and date are attributes of the association, which relates two people that have been married.

The UML standard does not provide formal semantics of association classes. The only constraint the standard imposes, is that an instance of an association class can connect exactly one instance of the associated classifiers [111, Sec. 7.3.4].

Association classes (like n-ary associations) can be refactored to binary associations, using a method called *association class promotion*, described by Fowler [49]. Since our transformation already deals with binary associations, we require that association classes are refactored to binary associations, using the association class promotion method. More specifically, an association class can be replaced by a class with two associations connecting the classes that participated in the association class. An additional constraint is inserted, to constraint that no two classes can relate the same elements.

For example, assume the *Marriage* association class of Figure 5.1. Figure 4.12, shows how this association class can be refactored to a binary association. In particular, the association class *Marriage* is replaced by the class *Marriage* and two associations connecting the newly generated class with the *Person* class that participated in the original association class.
The multiplicities of the association ends of the newly generated association are swapped (as shown by the dashed arrows). Finally an OCL constraint restricts that a Marriage instance cannot relate two Persons more than once.

Figure 4.12: Class Promotion of the Marriage Association Class

4.1.9 Qualified Associations

The UML standard specifies the notion of qualified associations [111, Sec. 7.3.44] as follows:

‘A qualifier declares a partition of the set of associated instances with respect to an instance at the qualified end (the qualified instance is at the end to which the qualifier is attached). A qualifier instance comprises one value for each qualifier attribute. Given a qualified object and a qualifier instance, the number of objects at the other end of the association is constrained by the declared multiplicity.’

More precisely, assume the class diagram of Figure 5.1. The relation between a Bank and a Person is qualified by the accountNumber. The qualified association indicates that a Bank has zero or more customers per accountNumber.

Akehurst et al. [18] propose that such a qualified association denotes that each Bank is related to a set of tuples such as: Tuple(accountNumber: Integer, customers: Set(Person)),

76
with the constraint that there is a unique accountNumber for each set of Persons. Informally this means that accountNumber is the unique key that may relate a Bank account with the customers that are the holders of the account.

Our approach requires that qualified associations are represented as standard binary associations, with OCL to capture the additional semantics. Akehurst et al. [18] present such a method of representing qualified associations as binary associations. In particular they replace the qualified association, with a class that explicitly represents the tuple of the qualified association. Using their method the Bank-Person qualified association of Figure 5.1 is replaced by the class diagram of Figure 4.13. The Tuple class has been introduced to show the relationship between the Person and the Bank classes. Finally the OCL constraint on the Bank states that for each tuple that relates the accountNumber with the Person, the value of the accountNumber attribute is unique.

Figure 4.13: An Equivalent Model of the Qualified Association Between Bank and Person of Figure 5.1

### 4.1.10 GeneralizationSet2Signatures

In UML 2.0 a Generalization relation can belong to a GeneralizationSet, which is used to show how the specific class partitions the set of object ids of the superclass. The UML standard specifies the following four combinations of how the object ids of the superclass can be partitioned by a GeneralizationSet [111, Section 7.3.20]:

- {complete, disjoint}
• \{incomplete, disjoint\} (This is the default)

• \{complete, overlapping\}

• \{incomplete, overlapping\}

*Disjoint* subclasses are classes that do not have any elements in common. For example in Figure 4.14a a Person can either be a Student or a Lecturer but not both. The sets of the object ids of the Student and the Lecturer are disjoint. *Overlapping* are subclasses that are not disjoint (i.e. the sets of object ids of the subclasses may overlap).

If a generalization hierarchy is annotated as *complete*, the subclasses completely partition the objects ids of the parent class. For example, in Figure 4.14c *Man* and *Woman* completely partition the object ids of the class *Person*, since a *Person* has to either be a *Man* or a *Woman*, but nothing else. *Incomplete* are subclasses that are not *complete* (i.e. the set of object ids of the subclasses do not completely partition the object ids of the superclass).

In Alloy there are two ways to express signature hierarchy, one is to use the *extends* keyword and the other to use the *in* keyword. The extends keyword defines subsignatures, which are *disjoint* and *incomplete*. On the other hand the *in* keyword in Alloy can be used to define *subsets* that can have elements in common (i.e. overlapping subsets).

**Representing GeneralizationSet in Alloy**

The mapping of the four combinations of the UML GeneralizationSet in Alloy is performed in the following way. Assume the UML class diagram of Figure 4.14a, which defines an \{incomplete, disjoint\} generalization. This simple model transformed to the Alloy model of Figure 4.14b. The *extends* keyword creates two *incomplete* and *disjoint* subsets of a *Person*, a *Student* and a *Lecturer*.

A UML class diagram that defines *complete* and *disjoint* subsets needs to be amended with an additional constraint. For example, assume the UML class diagram of Figure 4.14c.
This is transformed to the Alloy model depicted in Figure 4.14d, where a constraint is added (lines 4,5) to denote that a Person can either be a Man or a Woman, but nothing else.

\[
\begin{array}{l}
\text{(a) UML Diagram \{incomplete,disjoint\}}\\
\text{(b) Alloy \{incomplete,disjoint\}}\\
\text{[1] sig Person{} }\\
\text{[2] sig Student extends Person{}}\\
\text{[3] sig Lecturer extends Person{}}\\
\text{[4] fact{ all p:Person | } }\\
\text{[5] p in Man || p in Woman }\\
\end{array}
\]

Figure 4.14: Using the Extends Keyword to Declare Disjoint Subsignatures

In order to transform UML overlapping generalization relations into Alloy, we use the \textit{in} keyword. Using the \textit{in} keyword in Alloy we can define \textit{subsets} that can have elements in common (overlapping according to the UML terminology). For example, Figure 4.15a depicts that a \textit{Person} can be \textit{Tall} and \textit{Slim} at the same time. This is transformed to the Alloy model of Figure 4.15b.

Similarly Figure 4.15c depicts an \textit{Employee} that can be \textit{Employed} or \textit{SelfEmployed}. An Employed Employee can also be SelfEmployed, but nothing else. This simple model is transformed to the Alloy model of Figure 4.15d, with the constraint of lines 4 and 5 that every Employee is either Employed or SelfEmployed.

Our transformation deals with all 4 cases of GeneralizationSet presented in this section and transforms them to the corresponding Alloy models.
4.1.11 Profile Elements

In Section 3.1, we presented our UML profile for Alloy. The profile defines a number of stereotypes (see Figure 3.3), which can be used to facilitate fully automated analysis of UML models through Alloy, by expressing Alloy notions such as the scope and assertion commands, in UML. In this section we explain how these stereotypes are transformed to Alloy.

Simulation

The simulation stereotype can be applied to a constraint. If a constraint is stereotyped as simulation, it will be mapped to an Alloy run command. ‘An [Alloy] run command causes the analyzer to search for an example that witnesses the consistency of a function or predicate.’ [77, Appendix B.7.5]. Consequently the transformation of a UML constraint stereotyped as simulation into Alloy is used to generate a random instance that conforms to the invariants of the model and the constraint.
Assertion

Similarly to the simulation stereotype, the assertion stereotype is transformed to a check command in Alloy. ‘A check command causes it [the Alloy Analyzer] to search for a counterexample showing that an assertion does not hold.’ [77, Appendix B.7.5]. Therefore the assertion stereotype, is used when we want to find an instance of the model that conforms to the invariants of the model, but violates the constraint stereotyped as assertion.

Enforce

The enforce stereotype may be applied on a class. If a class is stereotyped as enforce, we require the Alloy Analyzer to enforce that at least one instance of the specified class exists, when carrying out the analysis. To explain the usage of this stereotype further, we need to consider how the Alloy Analyzer works. The analyser tries to find a minimal instance that satisfies the constraints of the model. As a result a partial instance of the model (i.e. where one or more classes of the model have no instances at all) is acceptable. The enforce stereotype is used to avoid this situation. Our transformation uses the some Alloy keyword to enforce the existence of at least one instance of a specific signature.

Singleton

The singleton stereotype is used to express the well known singleton pattern [55]. The singleton pattern is used to restrict that exactly one instance of the specified class(es) can exist. Our transformation to Alloy, uses the one Alloy keyword to restrict a signature to be a singleton set.

Analysis

The analysis stereotype is required to be applied to exactly one package in the model. During the transformation only the package stereotyped as analysis and its contents, will be trans-
formed to Alloy. The analysis stereotype defines two tagged values, the `defaultScope` and `intScope`, which are used by the modeller to specify the default scope for all classes and the scope of the bitwidth for integer numbers in the model.

**ScopedElement**

A class may be stereotyped as `scopedElement`. The `scopedElement` class defines a `scope` tagged value and is used to override the default scope for that specific class. This stereotype is used when transforming simulation or assertion commands.

**Dynamic**

How the ‘Dynamic’ stereotype is handled, is explained in detail in Chapter 6.

**Overview**

Figure 4.16 depicts an overview of the UML class diagrams to Alloy transformation rules presented in this section. The shaded UML concepts are not directly supported by our transformation. Instead we require that these concepts are expressed using simpler UML concepts, such as binary associations and OCL. The corresponding sections of this thesis present methods from the literature on how to achieve it.

The following section shows an example UML class diagram and how it is transformed to Alloy, using the transformation rules presented in this section.

**4.2 Example**

This section presents an example UML class diagram, which is inspired from the UML 2.0 metamodel [111]. The class diagram of Figure 4.17 defines a package called `Example`, which is an application of the UML2Alloy profile. The package specifies a `defaultScope` of 6 and the bitwidth for integer numbers (i.e. the `intScope`) of 5 (i.e. the range for integer numbers
Figure 4.16: Overview of the Transformation Rules
is -16 to +15). The example model defines an abstract class \textit{Element}. An Element can be either a \textit{NamedElement} or a \textit{RelationShip}. A \textit{NamedElement} has a \textit{name}, which is of type \textit{String} and a \textit{visibility}, which is of type \textit{VisibilityKind}. Since the UML profile for Alloy presented in Chapter 3 does not have any PrimitiveTypes apart from integers, the String primitive type needs to be defined on the model level as a \textit{dataType}. The \textit{RelationShip} class has two stereotypes, the \textit{enforce} and the \textit{scopedElement} and specifies a \textit{scope} tagged value. The scope tagged value defines that \textit{at most one} instance of a scopedElement should exist in the model (i.e. it overrides the defaultScope of 6 for the RelationShip class). Finally the model defines a \textit{DirectedRelationShip} class, which is a subclass of a \textit{RelationShip}, with one or more source and one or more target elements.

Applying the transformation rules presented earlier in this chapter, we can automatically transform the example class diagram of Figure 4.17 to an instance of the Alloy metamodel. Figure 4.18 is a UML object diagram that shows part of the generated instance of the Alloy metamodel. Using the rule presented in Section 4.1.1 we transform the UML package called ‘Example’ into an Alloy module called ‘Example’. This is represented by the \textit{ModuleId} object in Figure 4.18. Using the rules presented in Section 4.1.2 the UML classes are mapped to Alloy signatures. This is shown by the two \textit{ExtendsSigDecl} objects of Figure 4.18, which declare the \textit{Element} and \textit{NamedElement} \textit{SigIds}. Using the transformation rules of Section 4.1.3 the class attributes are transformed to Alloy fields of the respective signatures. This is shown by the ‘\textit{nDecl}’ and ‘\textit{vis}’ objects of type ‘\textit{Decl}’ (i.e. declaration) in the object diagram of Figure 4.18. Those two objects correspond to the UML class attributes ‘\textit{name}’ and ‘\textit{visibility}’ of the ‘\textit{NamedElement}’ in the class diagram of Figure 4.17.

Applying a MOF2Text transformation, we can transform the generated instance of the Alloy metamodel to a textual Alloy model, which can be provided as input to the Alloy Analyzer. Appendix F shows the complete Alloy code, which was automatically generated
from the UML class diagram of Figure 4.17.

Figure 4.17: A Simple Example to Illustrate our Transformation Rules
Figure 4.18: A Partial Alloy Metamodel Object Diagram
CHAPTER 5

TRANSFORMATION RULES FOR OCL

This chapter presents the transformation rules from OCL into Alloy. In contrast to the transformation rules for class diagrams presented in the previous chapter, the OCL transformation rules introduced in this chapter are not presented using the OCL metamodel. This is because the OCL is a textual language and as such, it has a rather complicated metamodel. For better illustration/presentation here we present the OCL to Alloy transformation rules using a template based transformation. More specifically, for each OCL element we provide the concrete syntax according to the OCL standard [108] and present the concrete syntax of the equivalent element in Alloy. A ‘TRANSFORM()’ method is used to indicate that the expression inside the parentheses is transformed to the equivalent Alloy statement, using the corresponding transformation rule presented in this work. For example, if the boolean-expression of a forAll OCL statement is another forAll statement, the transformation rule for the forAll operation will be executed recursively. The UML class diagram of Figure 5.1 will be used as a running example throughout this chapter.

---

1It has to be noted that the template based approach is used for better illustration/presentation of the transformation rules. The actual implementation of the OCL to Alloy transformation, the OCL and Alloy metamodels.
5.1 Navigation

In OCL the dot (.) operation is applied on objects and collections to navigate to different objects and collections in a class diagram [108, Sec. 7.5.3]. The name of the opposite association end is used to perform the navigation operation. Assume the following expression in the context of a Company in the class diagram of Figure 5.1: ‘self.manager’

This statement will return the manager of the Company on which the statement is evaluated (for more information on the ‘self’ keyword please refer to Section 5.2). The manager returned by this expression will be an object of type Person. If the upper multiplicity of the opposite association end is greater than one, the result will be a collection, instead of one object.

In Alloy the navigation dot (.) is treated as the relational join [77, Sec. 3.4.3.2]. For example, assume an instance of Figure 5.1, with the following instances:

Company = {(Company0),(Company1)}
Person = {(Person1), (Person2), (Person3)}
manager = {(Company0,Person1),(Company1,Person2)}

The expression: ‘this.manager’, evaluated in the context of Company0 will return ‘Person1’ using the following reasoning:

\{(Company0)\} . \{(Company0,Person1),(Company1,Person2)\} = \{(Person1)\}

There is a clear similarity between the navigation dot in OCL and the relational join in Alloy. Considering that the syntax of dot navigation expressions in OCL is: ‘object.associationEnd’, the transformation into Alloy is given by the following rule:

TRANSFORM(object).TRANSFORM(associationEnd)

**Remark**

Navigating over multiple associations in OCL returns a Bag structure rather than a Set [108, Sec. 7.5.3]. Since Bags are not supported by our transformation, all Bags are flattened to Sets in Alloy. This may cause potential problems, but it is necessary in order to take advantage of the simplicity of the Alloy semantics. As a workaround the modeller can explicitly specify the notion of a ‘Bag’ on the model level when it is necessary. For example, a UML class called ‘Bag’ can be introduced on the UML class diagram.

### 5.2 Context

An OCL statement is specified in a context, which is the context in which the expression is evaluated. The modeller can use the OCL keyword [108, Sec. 7.3.1] ‘self’ to refer to the instance of the object, on which the expression is evaluated.

\(^2\)The keyword ‘this’ in Alloy is equivalent to the keyword ‘self’ in OCL.
As described in Section 3.1, our UML profile for Alloy allows OCL statements to be specified in the context of a UML class (i.e. class invariants) or an operation (i.e. operation pre and post conditions). Here we present how class invariants are transformed to Alloy. For details on how operation pre and post conditions are treated, please refer to Section 6.3.

Alloy has the notion of signature facts [77, Sec. 4.5.1], which is a concept similar to that of UML class invariants. Signature facts are invariants specified in the context of a signature. The instance of the signature on which the fact is evaluated can be accessed using the special Alloy keyword ‘this’ (the Alloy keyword ‘this’ corresponds to the OCL keyword ‘self’).

Assume the UML class diagram of Figure 5.1 and the OCL invariant of Figure 5.2a. The invariant expresses if a manager manages a company, (s)he cannot manage another company. Using the OCL transformation rules presented later on in this chapter, the OCL invariant of Figure 5.2a, would map to the following Alloy statement:

```alloy
all c:Company | c != this =>
c.manager != manager
```

However, there is a potential problem if we use signature facts, as explained in [77, p. 119]. More specifically, by default all references to the fields of a signatures (i.e. in this example manager) are expanded with the this keyword. Therefore the simple expression: c.manager would expand to c.this.manager, which is not syntactically correct. To avoid the expansion we should have used the @ symbol after the navigation dot, during the transformation:

```alloy
all c:Company | c != this =>
c.@manager != manager
```

However, this is an unnecessary complication in the transformation, because for each OCL navigation expression to be transformed, it is necessary to check whether the @ symbol
is required. As a result, we are transforming class invariants to ordinary Alloy facts (i.e. facts without an implicit context), introducing the context of the transformation ourselves, using the following algorithm:

1. For every OCL class invariant, we generate an Alloy predicate (For more details on the notion of the Alloy predicate please refer to Section 2.1.2). The name of the predicate is name of the OCL invariant ³.

2. If a reference to the keyword ‘self’ exists in the invariant, create a parameter in the predicate. The parameter shall be called ‘self’ and the type of the parameter will be the type of the signature to which the UML class of the invariant was transformed.

3. The OCL specification defines that an invariant has to be true for all instances of a class and can be represented using the forall OCL operation [108, Sec. A.3.1.5]. Therefore in Alloy, a fact is generated that references the predicate and requires that all instances of the signature respect the predicate.

4. Finally, the body of the OCL statement is transformed using the transformation rules presented in the rest of this chapter.

Using this algorithm the OCL invariant of Figure 5.2a is transformed to the equivalent Alloy statements of Figure 5.2b. Following step 1 of the algorithm, the Alloy predicate called ‘invariant’ is generated (line 2). According to step 2 of the algorithm, the parameter ‘self’ is added to the predicate (line 2). Using step 3 of the algorithm, a fact is generated that requires the invariant to hold for all Companies (line 1). Finally, using step 4 of the algorithm the body of the predicate is generated (lines 3-4) using the transformation rules for OCL presented later on in this chapter.

³Please recall that our profile requires that all OCL invariants are given a unique name.
5.3 Boolean Operations

OCL allows the use of the standard boolean operations for conjunction (\textit{and}), disjunction (\textit{or}), negation (\textit{not}) and implication (\textit{implies}). Here we show how such expressions are transformed to Alloy. If we do not take into account the notion of undefinedness in OCL each OCL expression presented in this section is semantically equivalent to the corresponding Alloy expression (i.e. they produce the same truth tables). How undefined OCL expressions are treated by our transformation is explained in Section 5.14. In particular, since OCL is based on three-valued logic while Alloy is based on two-valued logic the modeller needs to specify explicitly how OCL undefined expressions are treated by the transformation, using the \texttt{oclIsUndefined()} operation.

Conjunction

The OCL syntax for conjunction is:

\begin{center}
\texttt{expr1 and expr2}
\end{center}

In Alloy the syntax for conjunction is\textsuperscript{4}:

\begin{center}
\texttt{&&}
\end{center}

\textsuperscript{4}The conjunction operator in Alloy is either a double ampersand (\texttt{&&}) or the keyword \texttt{and}.  

\begin{itemize}
\item \texttt{fact\{all c:Company \mid invariant(c)\}}
\item \texttt{pred invariant(self:Company) { all c:Company \mid c \neq self } => c.manager \neq self.manager} 
\end{itemize}

(a) Simple OCL Invariant on Figure 5.1

(b) Equivalent Alloy Statement

Figure 5.2: OCL Class Invariant and Alloy Representation
expr1 && expr2

The transformation rule that transforms a conjunction operation from OCL to Alloy is the following one:

TRANSFORM(expr1) && TRANSFORM(expr2)

Disjunction

The OCL syntax for disjunction is:

expr1 or expr2

In Alloy the disjunction operator is\(^5\): ||

The OCL disjunction operation is transformed to Alloy using the following rule:

TRANSFORM(expr1) || TRANSFORM(expr2)

Negation

The OCL syntax for negation is:

not expr

In Alloy the negation operation is the exclamation mark symbol\(^6\). Therefore the transformation rule for negation is the following one:

! TRANSFORM(expr)

\(^5\)The disjunction operator in Alloy is either a double vertical bar symbol (||) or the keyword ‘or’.

\(^6\)The negation operator in Alloy is either an exclamation mark symbol or the keyword ‘not’.
**Implication**

The syntax of the *implies* OCL operation is:

\[
\text{condition implies expression}
\]

If the *condition* is true, then the *expression* needs to hold.

Alloy supports the implication operator and the syntax is:

\[
\text{condition} \Rightarrow \text{expression}
\]

Consequently the transformation of *implies* OCL statements are carried out using the following rule:

\[
\text{TRANSFORM(condition)} \Rightarrow \text{TRANSFORM(expression)}
\]

### 5.4 If-then-else Expressions

The OCL syntax for if-then-else expressions is the following [108, Sec. 10.3.3]:

\[
\text{if condition then}
\quad \text{expr1}
\quad \text{else}
\quad \text{expr2}
\quad \text{end if}
\]

Alloy provides support for the if-then-else construct, with the help of the implies operator. More specifically the Alloy syntax for if-then-else expressions is:

\[
\text{condition} \Rightarrow \{\text{expr1}\}
\quad \text{else}\ {\text{expr2}}
\]
Therefore the OCL if-then-else construct is transformed, using the following simple rule:

\[
\{\text{TRANSFORM(condition)}\} \Rightarrow \\
\{\text{TRANSFORM(expr1)}\}
\]

\text{else}

\[
\{\text{TRANSFORM(expr2)}\}
\]

### 5.5 Operations on all Expressions

A number of operations are defined on all types expressions (i.e. not only boolean expressions). Equality and inequality are two such operations.

#### Equality

Both OCL and Alloy use the standard equality symbol to state or query if two expressions are equal. Consequently our transformation rule is the following one:

\[
\text{TRANSFORM(expr1)} = \text{TRANSFORM(expr2)}
\]

#### Inequality

The OCL syntax for inequality is the following one:

\[
\text{expr1} \neq \text{expr2}
\]

In Alloy negation and equality are used to express inequality. As a result the inequality OCL operation is transformed using the following method:

\[
\text{TRANSFORM(expr1)} \neq \text{TRANSFORM(expr2)}
\]

### 5.6 Predefined Operations on Objects

The OCL standard specifies a number of predefined operation on objects [108, Sec. 7.5.9]. In this section we present those operations and discuss how they can be mapped to Alloy.
5.6.1 oclIsTypeOf

The syntax of oclIsTypeOf is the following [108, Sec. 7.5.9]:

\[
\text{object.oclIsTypeOf(t)}
\]

This statement returns true if object is of type t. It is not possible to directly express this OCL statement in Alloy. This is because in reality Alloy is an untyped language [77, Sec. 107]. Its type system is very simple and is based on set containment.

For example, assume the class diagram of Figure 5.3 and the following two OCL expressions:

-- First OCL expression
context A
self.oclIsTypeOf(A)

-- Second OCL expression
context B
self.oclIsTypeOf(A)

In OCL the first constraint will return true, while the second one will return false. We could map those two OCL statements to Alloy using set containment statements. The equivalent Alloy statements would be:

--Translation of first expression
pred firstOCL(a:A){
a in A }

96
Unlike OCL, both Alloy statements will evaluate to true, because \( B \) is a subset of \( A \) and every element of \( B \) is also an element of \( A \). Consequently the \textit{oclIsTypeOf} OCL operation cannot be directly expressed in Alloy.

However it is possible to express it in a more verbose manner. In particular, assume the simple class diagram of Figure 5.3, where \( a \) and \( b \) are objects of the \( A \) and \( B \) classes respectively. Table 5.1 provides the truth table of the four combinations of the \textit{oclIsTypeOf()} operation on the \( A \) and \( B \) classes and their objects. The statements in the first and fourth rows return \textit{true}, while the rest of the statements return \textit{false}. These four combinations of the \textit{oclIsTypeOf()} operation, can be expressed using set inclusion, as shown in Table 5.2.

It is possible to query if the object on which the \textit{oclIsTypeOf()} operation is being applied, belongs to the set of all the instances of the class, but does not belong to the instances of the subclass(es) of that class. In particular, as shown in the first row of Table 5.2 the expression \( a.oclIsTypeOf(A) \) returns the same result as the expression that checks if \( a \) belongs to the set of all instances of \( A \), but does not belong to the set of instances of its subclass(es). Using this algorithm the rest of the rows in Table 5.2 give the same result as the statements in Table 5.1. The \textit{includes()} and \textit{excludes()} operations can be expressed in Alloy using the transformation rules presented in Sections 5.7.1 and 5.7.1. Consequently the \textit{oclIsTypeOf()} statement:

\[
\text{object.oclIsTypeOf(Class)}
\]

can be expressed in a form that can be directly transformed to Alloy, using the following algorithm:

\[
\text{Class.allInstances()} \rightarrow \text{includes(object)}
\]
for each s: subclass(Class)
    s.allInstances() \rightarrow \text{excludes(object)}

next s

The algorithm states that set inclusion can be used to check if an object is of certain type and for each subclass of the Class, the set exclusion operation needs to be used.

**Remark**

An important remark is that this method will work if the oclIsTypeOf() operation is used against user defined class types. It will not work if it is applied on predefined types, such as

<table>
<thead>
<tr>
<th>OclIsTypeOf Expression</th>
<th>Set Inclusion Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.oclIsTypeOf(A)</td>
<td>A.allInstances() \rightarrow \text{includes(a)} and B.allInstances() \rightarrow \text{excludes(a)}</td>
</tr>
<tr>
<td>b.oclIsTypeOf(A)</td>
<td>A.allInstances() \rightarrow \text{includes(b)} and B.allInstances() \rightarrow \text{excludes(b)}</td>
</tr>
<tr>
<td>a.oclIsTypeOf(B)</td>
<td>B.allInstances() \rightarrow \text{includes(a)}</td>
</tr>
<tr>
<td>b.oclIsTypeOf(B)</td>
<td>B.allInstances() \rightarrow \text{includes(b)}</td>
</tr>
</tbody>
</table>
5.6.2 oclIsKindOf

The syntax of oclIsKindOf is the following [108, Sec. 7.5.9]:

\[
\text{object.oclIsKindOf}(t)
\]

This statement returns true if \textit{object} conforms to the type \textit{t}. The difference from \textit{oclIsTypeOf}, is that \textit{oclIsKindOf} returns true, if \textit{t} is either the direct type of the \textit{object} or a supertype of the \textit{object}.

In Alloy \textit{oclIsKindOf} can be represented with set containment, using the following statement:

\[
\text{object} \in \text{t}
\]

This statement returns true if \textit{object} belongs to the set \textit{t}. The statement will also return true if \textit{t} is a supertype (superset) of \textit{object}. Therefore the result of this Alloy statement is equivalent to the result \textit{oclIsKindOf} in OCL. As a result the transformation rule is the following one:

\[
\text{TRANSFORM(object) in TRANSFORM(t)}
\]

5.6.3 Unsupported Operations

A number of OCL operations on objects, cannot be expressed in Alloy. These are presented in this section.

\textit{oclIsInState}

This operation is not supported by our transformation, as it evaluates whether an object is in a specific state. To evaluate this statement, a UML statemachine diagram [49] is required. Representing UML statemachines in Alloy is out of the scope of this work.
oclIsNew

The oclIsNew operation return true if the object did not exist in the previous state of the system but exists in the current state. This operation can only be used in postconditions. As the oclIsNew is a property about dynamic systems it is not considered here.

oclAsType

The oclAsType statement is used for casting purposes. Its syntax is [108, Sec. 7.5.9]:

\[
\text{object} \rightarrow \text{oclAsType}(t)
\]

This operation is used to cast the object to the type \( t \) (as long as the object conforms to the type \( t \)). The oclAsType operation is useful, when we want to access a property that is redefined and is available in a superclass. In that case we need to use oclAsType to cast the subclass to the superclass and access the property. In Alloy there is no need for casting and thus no need to transform the oclAsType operation, since there is no support for property redefinition.

5.7 Collections

UML has inherent support for a number of collection constructs. In particular the OCL standard defines Sets, Bags, OrderedSets and Sequences) [108, Sec. 7.5.11]. On the other hand Alloy only supports Sets. Recently support for sequences have been introduced to the Alloy language. Here we show that it is possible to express the UML concepts of Bags and OrderedSets using the notion of Sequences in Alloy. An Alloy Sequence is defined as a relation between an integer value and a user defined signature. The integer value denotes the index of the element in the Sequence.

In UML a Bag is collection like a Set, but allows elements to appear more than once in the collection. A Bag can be represented as an Alloy Sequence, ignoring the integer value, which represents the index of the element in the Sequence. Similarly a UML OrderedSet is a Set,
whose elements are ordered. Again this can be easily expressed using an Alloy Sequence, with an additional constraint that no two distinct elements in the Sequence can be the same.

Currently our transformation rules only support the translation of sets. Representing the rest of the UML collection constructs in Alloy remains for future research.

**Remark : Nested Collections**

The UML standard allows for nested collections (i.e. Collections of Collection) [108, Sec. 7.5.12]. In Alloy on the other hand all collections are flat and it is not possible to express higher-order relations [77, p. 41]. As a result if a collection of a collection is defined in OCL, it is rejected by our transformation.

### 5.7.1 Predefined Collection Operations

The OCL standard defines a number of predefined operations on collections. In this section we discuss how those operation can be transformed to Alloy.

**forAll**

The following provides the general syntax of OCL’s universal quantifier expressions [108, Section 7.6.3].

```
collection -> forAll(var:Type | boolean-expression)
```

The expression is evaluated for every item in the `collection`. The declared variable `var` of type `Type` binds each element of the `collection`, with `var` and then the `boolean-expression` is evaluated. If `boolean-expression` is true for every binding of `var` the whole expression is true, otherwise the expression if false.

Alloy, as a language based on first-order predicate logic, directly supports universal and existential quantifiers. The Alloy syntax for universal quantification is the following:

```
all var:collection | boolean-expression
```
Alloy evaluates this expression by binding `var` for every element in `collection` and evaluating `boolean-expression`. If `boolean-expression` is true for every binding of `var`, the whole expression evaluates to true, otherwise it evaluates to false.

It is therefore clear that there is a direct mapping between the OCL `forAll` construct and the Alloy `all` statement. Our transformation of `forAll` OCL expressions is the following:

```
all TRANSFORM(var) : TRANSFORM(collection) |
  TRANSFORM(boolean-expression)
```

**exists**

An approach similar to the approach for transforming `forAll` OCL statements to Alloy, can be followed when dealing with OCL statements expressing existential quantification. The following provides the syntax of OCL’s existential quantifier expressions [108, Section 7.6.4]:

```
collection -> exists(var:Type | boolean-expression)
```

The evaluation of this expression is similar to the evaluation of the `forAll` expression. More specifically an `exists` expression evaluates to true if there is at least one element in `collection` that makes the `boolean-expression` to evaluate to true.

Similarly in Alloy the syntax for existential quantification is the following:

```
some var:collection | boolean-expression
```

This expression evaluates to true if there is at least one element in the `collection` that makes `boolean-expression` to evaluate to true.

Again there is a direct mapping between the OCL `exists` and the Alloy `some` constructs. Our transformation of `exists` OCL expressions is the following:

```
some TRANSFORM(var) : TRANSFORM(collection) |
  TRANSFORM(boolean-expression)
```
The OCL select statement is used to select members of a collection that satisfy a boolean expression and return a new collection that contains only those members [108, Sec. 7.6.1]. The general OCL syntax of the statement is:

```
collection -> select(var:Type | boolean-expression)
```

Such a statement selects the elements of `collection`, which satisfy the `boolean-expression` and returns a new collection with only those elements.

A similar concept in Alloy is a comprehension expression [77, p. 280]. A comprehension expression is of the form:

```
{ x1: e1 | F }
```

and returns a set of `x1` from the set `e1`, where the condition `F` is true. For example, assume the following OCL statement on the class diagram of Figure 5.1:

```
-- The company respects equal opportunities
-- At least two of the employees are female
context Company
inv: self.employee -> select(e:Person |
    e.gender = Gender::female) ->
    size() >= 2
```

This statement can be expressed in Alloy using the following formula:

```
pred invariant(self:Company){
  #{e:self.employee | e.gender = Female } >= 2
}
```
As a result an OCL \textit{select} statement with the syntax:
\[
\text{collection} \rightarrow \text{select}(\text{var:Type } \mid \text{boolean-expression})
\]
is mapped to Alloy using the following rule:
\[
\{ \text{TRANSFORM}(\text{var}): \text{TRANSFORM}(\text{collection}) \\
| \text{TRANSFORM(\text{boolean-expression})} \}
\]
\textbf{reject}

A \textit{reject} OCL statement has the opposite effect of \textit{select} \cite[Sec. 7.6.1]{[108]}. Its syntax is:
\[
\text{collection} \rightarrow \text{reject}(\text{var:Type } \mid \text{boolean-expression})
\]

The reject expression returns a new collection, that does not contain the elements that satisfy the \textit{boolean-expression}. A reject statement is equivalent to the following statement \cite{[35]}:
\[
\text{collection} \rightarrow \text{select}(\text{var:Type } \mid \text{not boolean-expression})
\]

As a result when translating a reject operation to Alloy, we use the translation for the \textit{select} statement as presented in the previous section and negate the \textit{boolean-expression}.

\textbf{size}

If the \textit{size} operation is applied on a collection it returns its cardinality (i.e. the number of elements in the collection). This is an integer number with a value greater or equal to zero. This operation can only be applied to countable sets.

In Alloy the cardinality of a set is given by using the hash symbol (\#).

Consequently the \textit{size} OCL operation of the form:
\[
\text{collection} \rightarrow \text{size()}
\]
is transformed to Alloy using the following method:
\[
\# \text{ TRANSFORM}(\text{collection})
\]
**isEmpty**

The `isEmpty` operation returns true if the collection on which it is applied is empty (i.e. has no elements). Otherwise it returns false.

```
collection -> isEmpty()
```

The Alloy keyword ‘`no`’ can be used to check whether the collection is empty. Consequently, the `isEmpty` OCL operation is transformed to the equivalent Alloy expression:

```
no (TRANSFORM(collection))
```

**notEmpty**

Similarly to the `isEmpty()` operation, the `notEmpty` operation is used to denote that a collection contains at least one element (i.e. it is not empty). The syntax of the statement is:

```
collection -> notEmpty()
```

This operation can be expressed in Alloy with the ‘`some`’ Alloy keyword as in the following:

```
some (TRANSFORM(collection))
```

**count**

The `count` operation is used to return the number of times a specific object exists in a collection. Its syntax is:

```
collection -> count(elem)
```
If the \textit{elem} does not exist in \textit{collection}, the operation returns zero\footnote{The UML specification [108, A.2.5.5] indicates that \textit{count} should return 2, if \textit{elem} does not exist in \textit{collection}, but this is a typo.}, otherwise it returns the number of times the \textit{elem} exists in the collection.

In Alloy it is possible to create a simple polymorphic function that accepts as parameters the collection and the element and returns an integer value, which is how many times the element appears in the collection.

In Alloy the predominant collection type is \textit{set}, where an element cannot appear more than once. Therefore if an object belongs to the set, the \textit{count()} method should return 1, otherwise it should return 0. The following function represents the concept of \textit{count} on sets in Alloy:

\footnotesize
\begin{verbatim}
fun count(collection: set univ, elem: univ): Int{
  elem in collection=> 1
  else
    0}
\end{verbatim}
\normalsize

If the \textit{count()} operation is applied on a collection of type sequence, instead of a set, the result can be greater than 1. In such a case, the following Alloy function is used to represent the \textit{count()} operation:

\footnotesize
\begin{verbatim}
fun countSeq(sequence: seq univ, elem: univ): Int{
  // sequence.indsOf[elem] returns a set of the
  // indexes of the sequence, where elem appears
  #sequence.indsOf[elem] }
\end{verbatim}
\normalsize

This statement returns the cardinality of the set that contains the indexes of the sequence, where \textit{elem} appears.
includes

The *includes* operation can be used on a collection to check if an element belongs to the collection. The syntax of includes expression is:

\[
\text{collection} \rightarrow \text{includes}(\text{elem})
\]

In Alloy the *in* keyword can be used to check for if an element belongs to a set. As a result an *includes* OCL statement will be transformed to Alloy using the following rule:

\[
\text{TRANSFORM}(\text{elem}) \in \text{TRANSFORM}(\text{collection})
\]

**Remark : Type Checking**

In this Alloy formula, *elem* can be a set of any cardinality. It can be a set containing more than one elements. On the other hand the OCL standard states that *elem* has to be a scalar. As a result, using this approach it is not feasible to identify a case where a set is erroneously used as a parameter to *includes*. In such a case the OCL operation *includesAll* should be used.

includesAll

The *includesAll* operation is similar to *includes*, but is used to check if a collection *collElem* belongs to another collection. The syntax of the operation is the following:

\[
\text{collection} \rightarrow \text{includesAll}(\text{collElem})
\]

In Alloy, to check for set inclusion it is possible to use the *in* keyword. The *includesAll()* operation is transformed to Alloy in the following manner:

\[
\text{TRANSFORM}(\text{collElem}) \in \text{TRANSFORM}(\text{collection})
\]

Alloy typechecks this statement following its standard typechecking rules [77] and it makes sure *collElem* and *collection* are not disjoint sets and have the same arity.
excludes

The *excludes* operation checks if an object does not belong to a collection and is the opposite of the *includes* operation. The OCL syntax is:

\[\text{collection} \rightarrow \text{excludes}(\text{elem})\]

This is transformed to Alloy using the following pattern:

\[\text{TRANSFORM}(\text{elem}) \text{ not in TRANSFORM}(\text{collection})\]

excludesAll

Similarly to the *includesAll* operation, the `excludesAll` is used to check if all the elements of a collection do not belong to another collection. The OCL syntax is:

\[\text{collection} \rightarrow \text{excludesAll}(\text{collElem})\]

Similarly to the `includesAll`, the `excludesAll` operation is transformed to Alloy using the following rule:

\[\text{TRANSFORM}(\text{collElem}) \text{ not in TRANSFORM}(\text{collection})\]

iterate

Iterate is a construct with an imperative flavour. To the best of our knowledge there is no equivalent statement in Alloy. Consequently, our transformation rules do not support the iterate construct. If an *iterate* statement can be equivalently expressed using OCL constructs supported by our transformation, the modeller can manually refactor the iterate statement using those constructs and use our approach to transform the statements to Alloy. For example, the OCL *reject, select, forAll, exists* and *collect* statements can be expressed using the *iterate* OCL statement [108, Sec. 7.6.5]. It is therefore expected that in certain occasions the reverse is possible (i.e. the *iterate* construct can be expressed with the help of the *reject, select, forAll, exists* and *collect* expressions).
collect

The *collect* operation can be used on a collection and returns all the objects on a collection that is derived from the original one [108, Sec. 7.6.2]. The syntax of the collect operation is:

\[ \text{collection} \rightarrow \text{collect( } v: \text{Type | expression-with-v) } \]

If *collection* is of type Set, then the new collection that will be returned will be of type Bag. Since the transformation presented in this work only supports Sets, the collect operation is supported, but will always return a Set (the same issue has been discussed on the navigation over multiple associations in Section 5.1).

Alloy’s uniform semantics on scalars and collections does not require an operation, similar to *collect*. Instead, the standard dot join operator can be used. Consequently the OCL *collect* operation will be transformed into Alloy using the following rule:

\[ \text{TRANSFORM(collection)}.\text{TRANSFORM(expression-with-v)} \]

product

Another operation on collections, defined by the OCL standard is the *product* [108, Sec. 11.7.1], which represents the well known cartesian product operation of set theory. The syntax of the operation is:

\[ \text{collection1} \rightarrow \text{product(collection2) } \]

and the result of the operation is a set of ordered pairs, where the first coordinate is populated by elements of *collection1* and the second coordinate is populated by elements of *collection2*.

In Alloy the relational arrow (\(\rightarrow\)) is used to represent the product operation. The transformation of the *product* OCL operation is carried our using the following rule:

\[ \text{TRANSFORM(collection1) } \rightarrow \text{TRANSFORM(collection2) } \]
isUnique

The isUnique operation returns true if an element appears exactly once in a collection [108, Sec. A.3.1.3]. The syntax of the isUnique expression is:

\[
\text{collection} \rightarrow \text{isUnique} (\text{element})
\]

The isUnique operation can be expressed, using the count operation as follows:

\[
\text{collection} \rightarrow \text{count} (\text{element}) = 1
\]

On page 105, we have already presented how the count operation can be represented in Alloy. We therefore represent the isUnique OCL operation with the help of count and transform it to Alloy, using our existing transformation rules for the count operation.

5.7.2 Predefined Set Operations

Section 11.7.2 of the OCL standard [108] specifies a number of operations, which may be applied on sets. These operations represent well known concepts of set theory, such as the union and intersection of sets. Here we present how these operation may be transformed to Alloy.

union

The union OCL operation, represents the well known union operation of set theory. The OCL syntax of the operation is:

\[
\text{set1} \rightarrow \text{union} (\text{set2})
\]

The result of the operation is a third set, which contains the union of sets, set1 and set2. In Alloy the union operator is the plus (+) symbol. Consequently the union OCL operation is represented in Alloy, using the following rule:

\[
\text{TRANSFORM(set1)} + \text{TRANSFORM(set2)}
\]
intersection

The syntax of the OCL operation for intersection is the following:

\[
\text{set1} \rightarrow \text{intersection(set2)}
\]

The result of the expression is a third set, which contains the intersection of sets, \textit{set1} and \textit{set2}. This operation is transformed to Alloy using the following rule:

\[
\text{TRANSFORM(set1) \& TRANSFORM(set2)}
\]

including

If the \textit{including} OCL operation is applied to a set, it results in another set, including the item passed as a parameter to the operation. The syntax of the operation is:

\[
\text{set} \rightarrow \text{including(element)}
\]

In Alloy we can use the union operator to represent the inclusion of the \textit{element} in the \textit{set}. As a result the including operation is transformed to:

\[
\text{TRANSFORM(set) + TRANSFORM(element)}
\]

excluding

The \textit{excluding} OCL operation has the opposite effect of the \textit{including} operation. Its syntax is:

\[
\text{set} \rightarrow \text{excluding(element)}
\]

In Alloy we can express the \textit{excluding} operation using set difference. The following rule is use to transform the excluding operation.

\[
\text{TRANSFORM(set) - TRANSFORM(element)}
\]
symmetricDifference

The syntax of the \textit{symmetricDifference} operation is the following:

\[
\text{set1} \rightarrow \text{symmetricDifference(set2)}
\]

The operation returns the elements in \textit{set1} or \textit{set2}, but not both [108, p. 147]. The symmetricDifference can be expressed with the help of the union and intersection operations in Alloy and is transformed as follows:

\[
(\text{TRANSFORM(set1)} + \text{TRANSFORM(set2)})
- (\text{TRANSFORM(set1)} \& \text{TRANSFORM(set2)})
\]

\textbf{Unsupported Operations}

A number of operations can convert a set to other collection structures (i.e. bags, sequences, orderedsets). As explained in Section 5.7 our transformation does not support sequences, bags and orderedsets. Consequently the OCL operations \textit{asSequence}, \textit{asBag} and \textit{asOrderedSet} are not supported. If they are found in a UML model the transformation quits with an error.

\textbf{5.8 Integer Expressions}

An important difference between OCL and Alloy is the dealing of Integer expressions. To illustrate the difference assume the following simple Alloy model, which depicts a \textit{Person} with a field called ‘\textit{age}’:

\[
\text{sig Person\{ age: one Int\}}
\]

In this Alloy model \textit{age} is a relation between \textit{Person} and \textit{Int} atoms. Therefore if the following expression is applied on a Person: ‘\textit{this.age}’, it will return an \textit{Int} atom that corresponds to the age of the Person. However, if this statement took part in an integer expression, we would
have to cast the ‘Int’ atoms to the ‘int’ primitive types. For example, the following expression: ‘this.age > 18’, needs to convert the Int atom of the left hand side of the rule to primitive integer value, so as to carry out the comparison. Fortunately this is automatically done by the Alloy Analyzer, thus it does not bring any additional complications in our transformation and the transformation of integer expressions into Alloy is rather trivial.

It has to be noted however that integer operations such as multiplication and division were not supported by the Alloy language at the time of specifying the transformation rules. However, recently support for those operation has been added to the language. Incorporating this development into our transformation framework remains to be investigated in the future.

**Supported Operations**

OCL comparison operations on integer numbers (>,<,>=,=<,=,<>)) map exactly to Alloy’s comparison operations for greater, less, greater or equal, less or equal, equal and not equal. In both the OCL and Alloy the plus (+) and minus (-) symbols are used to denote addition and subtraction operations on integer numbers.

**Sum**

The OCL specifies a *sum* operation on collections of integer numbers. The operation returns the sum of all numbers in the collection. Similarly in Alloy the *sum* operation can be used on sets of Int atoms to return a primitive integer value, which is the sum of all the numbers in the set. As a result an OCL operation:

```
collection -> sum()
```

is transformed using the following rule:

```
sum {TRANSFORM(collection)}
```
**Remark**

Previously we referred to the implicit casting between ‘Int’ atoms and primitive integer values carried out by the Alloy Analyzer. This can potentially create problems. Assume the following Alloy expression in the context of a Company, on the equivalent Alloy model of Figure 5.1:

\[
\text{this.employee.age} > 18
\]

The left hand side part of the expression will give a set of the ages of all the employees. Assume there are three employees in the Company where the expression is evaluated and the ages of each employee is 10, 11 and 12 years old. Then the left hand side of the expression will return the set \{10, 11, 12\}. The Int atoms to primitive integer values conversion, will convert the sum of the set of Int atoms to primitive values. Consequently the left hand side of the expression, when cast to primitive integer values will return the value 33 and the whole statement will return true.

Unlike Alloy, this expression in OCL should not type check because the left hand side is a Bag, while the right hand side is an integer primitive value and comparing a Bag of integers with a primitive integer value should raise a type error.

### 5.9 Transitive Closure

A useful operation on relations between classes is the transitive closure operation.

**Definition 5.1.** A binary relation \( R \) in a set \( A \) is called transitive, if for all \( a, b, c \) in \( A \), \( aRb \) and \( bRc \), imply \( aRc \) [71].

**Definition 5.2.** If \( R \) and \( R^+ \) are binary relations in a set \( X \), then \( R^+ \) is called the transitive closure of \( R \) if:
1. $R^+$ is a transitive relation

2. $R \subseteq R^+$

3. if $R'$ is a binary relation on $X$ satisfying 1 and 2, then $R^+ \subseteq R'$

OCL does not provide an operation which denotes the transitive closure of a relation. Instead the UML standard demonstrates a method to calculate the transitive closure. More specifically a well-formedness constraint in the UML metamodel expresses that a generalisation relation has to be acyclic [111, Sec. 7.3.8]. To express this notion in OCL, the standard defines an operation ($allParents()$), which gives all the direct and indirect ancestors of a generalised Class [111, p. 49]. The $allParents()$ operation is defined as follows:

$$allParents = self.parents() \rightarrow \text{union}(self.parents() \rightarrow \text{collect}(p | p.allParents()))$$

It is evident that the $allParents()$ operation is recursive and calculates the transitive closure of all the parents (i.e. direct and indirect) of a class. Baar [25] has proved that the $allParents()$ operation in the UML standard is correct, as long as the relation is acyclic. So, the $allParents()$ operation, that is used in the UML standard to forbid generalization relations from being cyclic is correct only when the the generalization relation is already acyclic! Apart from recursion, other methods of expressing transitive closure in OCL have been proposed in the literature. Mandel and Cengarle [95] and Richters [120, p.112] express transitive closure with the help of the iterate OCL operation.

Alloy, as a fully declarative language, does not directly support either recursion or the iterate OCL operation. So it is not possible to express the transitive closure in Alloy, in the same way it is expressed in the UML community. Fortunately Alloy’s relational logic provides inherent operators for expressing the transitive closure of a binary relation.
Formally Alloy defines the transitive closure of a relation as [77, p. 292]:

\[ E[^\wedge p]_i = \{ (x, y) \mid \exists p_1, \ldots, p_n \mid (x, p_1), (p_1, p_2), \ldots, (p_n, y) \in E[p]_i \} \]

This definition follows the standard definition of transitive closure in relational databases [39, p. 178]. Informally the transitive closure definition of Alloy specifies that a relation between \( x \) and \( y \) is the transitive closure if there is a path that connects \( x \) to \( y \).

Since our work uses Alloy as the semantic domain for UML, it is possible to express the transitive closure in OCL. We extend OCL by introducing a \( tclosure() \) operation, that when transformed to Alloy, denotes the Alloy’s notion of transitive closure.

### 5.10 Let Expressions

The OCL allows the definition of \( let \) expressions [108, Sec. 7.4.3], with syntax:

\[
let \ var: \text{type} = \ expr \ in \ expression-with-var
\]

Let expressions are used for convenience as a shorthand to referring to an expression. A \( let \) expression binds the variable \( \text{var} \) to the \( \text{expr} \), so that \( \text{var} \) can be used in the place of \( \text{expr} \). In Alloy there is a similar notion of let declarations, with the syntax:

\[
let \ \text{varId} = \ expr \ | \ {expression-with-varId}
\]

The mapping of let expressions is therefore quite trivial, using the following rule:

\[
let \ \text{TRANSFORM(var)} = \ \text{TRANSFORM(expr)} \ | \\
\{\ \text{TRANSFORM(expression-with-var)}\}
\]
5.11 Tuples

In OCL 2.0 the ability to specify tuple structures has been added [108, Sec. 7.5.15]. However, custom OCL expressions can be used to define the tuple elements. For example, assume the following OCL expression, taken from [108, p. 24]. The statement refers to elements of the class diagram of Figure 5.1.

```ocl
class Person

attr statistics : Set(TupleType(company: Company, numEmployees: Integer, wellpaidEmployees: Set(Person), totalSalary: Integer)) =
managedCompanies->collect(c |
Tuple { company: Company = c,
numEmployees: Integer = c.employee->size(),
wellpaidEmployees: Set(Person) = c.job->select(salary>10000).employee->asSet(),
totalSalary: Integer = c.job.salary->sum()
}
)
```

This statement declares a tuple of 4 coordinates and uses OCL to specify how the elements of each coordinate will be populated. In Alloy support for tuples exists and the previous OCL could have been defined as:

```alloy
statistics: Company -> Integer
            -> set Person -> Integer
```

This Alloy statement makes it more difficult to refer to a specific coordinate. For example, if we wanted to refer to the `numEmployees` coordinate, we would have to write the following statement: `Company.statistics.Int.Person`. Moreover if we want to transform the
OCL statements that specify the values of the coordinates, we would have to use universal or existential quantifiers to refer to the specific elements of the tuple. To avoid such issues, our transformation does not deal with tuples. Instead, if a tuple needs to be specified, it can be done with the help of a class diagram.

5.12 Undefinedness and Errors

The OCL standard specifies a number of OCL operations on all types to represent lack of value or a runtime error. These operations are presented in this section.

5.12.1 OclIsUndefined

The syntax of the oclIsUndefined expression is [108, Sec. 11.2.5]:

\[
\text{expr\text{.oclIsUndefined()} \quad \text{This statement returns true, if the expression } \text{`expr`} \text{ is undefined, otherwise it returns false.}
\]

As described in Section 5.14, we assume that the notion of undefinedness in OCL is similar to the notion of an empty set in Alloy. Consequently the iclIsUndefined operation is transformed to:

\[
\#{\text{TRANSFORM(\text{expr})}} = 0
\]

5.12.2 Other Errors

In addition to the oclIsUndefined() operation the OCL provides additional types and operations to check for potential errors that might occur in the evaluation of an expression.

OclVoid

An OCL expression can evaluate to the OclVoid type. The OclVoid type has exactly one instance called ‘null’, which represents the absence of a value [108, p. 36]. The null value can be present in collections.
**OclInvalid**

In addition to the *OclVoid* type the OCL standard also defined an *oclInvalid* type on expressions. The *OclInvalid* type has a single instance called *invalid* [108, p. 138]. ‘*Any property call applied on invalid, results in oclInvalid, except for the operations oclIsUndefined and oclIsInvalid.*’ [108, p. 138].

Using the above description of *oclVoid* and *oclInvalid* types, taken from the UML specification, we can assume that while *oclVoid* is the type of expressions, which represents lack of value, *oclInvalid* is the type for all other kinds of errors (for example division by zero). Moreover the OCL standard defines the operation ‘*oclIsInvalid*’, which can be used to check if the type of an expression is *oclInvalid*.

Using the *null* value to denote the absence of a value is closer to programming languages (e.g. Java), when the absence of a value is indicated by a special *null* value. However, in a language based on set-theoretic semantics, used for the high level modelling of systems, the use of the *null* value seems to complicate things. For example, the empty set could be used to denote the absence of a value. In particular assume the following two expressions:

\[
\begin{align*}
\text{Set}\{\text{null}\} & \rightarrow \text{size}() \\
\text{Set}\{} & \rightarrow \text{size}()
\end{align*}
\]

Both expressions can potential represent the absence of value (navigating over an association end with a lower multiplicity of zero and upper multiplicity of greater than one, will return an empty set) and we would assume both statements should return the same result. However, the first expression will return 1, while the second expression will return zero as expected. The use of many types of expressions (i.e. *undefined*, *invalid* and *null*) to represent the concept of the absence of values and the confusion this could cause to the modeller, is also discussed by Brucker et al. [33].
In order to avoid such issues and considering the fact that the explicit representation of the \texttt{null} value is useful for modelling source code, but is not very often used when modelling high level abstract systems, our transformation does not support the \texttt{oclVoid} and \texttt{oclIsInvalid} types. If it is required to explicitly model the absence of value, using a special \texttt{null} value, it is feasible to do so on the model level. For example, the type of an attribute can be an enumeration, with an enumeration literal \texttt{null} representing explicitly on the model level the absence of value.

\section*{5.13 Operations Precedence}

An important issue when transforming OCL into Alloy, is to ensure that operations precedence is maintained. Appendix G shows the operation precedence rules for both OCL and Alloy. From a close inspection of the rules, it is obvious that equivalent notions in UML and Alloy have different priority. For example, assume the expression:

\begin{equation*}
\text{expr1 or expr2 implies expr3}
\end{equation*}

In OCL it will be parsed as:

\begin{equation*}
(\text{expr1 or expr2}) \text{ implies expr3}
\end{equation*}

However, in Alloy where the \textit{implies} operation has a higher precedence, the expression will be parsed as:

\begin{equation*}
\text{expr1 or (expr2 implies expr3)}
\end{equation*}

This difference in the operation precedence rules can potentially change the result of an OCL expression when evaluated in Alloy. In order to avoid this situation, when transforming an automatically generated instance of the Alloy metamodel to text, we use parentheses.
For example, the OCL parser, which takes into consideration the OCL operation precedence rules, will create the tree of Figure 5.4 from the expression. Using our OCL to Alloy model transformation this tree will be converted to an equivalent Alloy syntax tree. Finally, the MOF2Text rules, which are responsible for generating the actual textual Alloy model, use parentheses on every branch of the tree. As a result the original OCL expression:

\[ \text{expr1 or expr2 implies expr3} \]

will be transformed to the following Alloy expression:

\[ ((\text{expr1 or expr2}) \text{ implies } \text{expr3}) \]

The parentheses ensure that when the Alloy Analyzer parses the textual Alloy model, it creates a parse tree equivalent to the original OCL parse tree.

5.14 Undefined Expressions

An important issue, when transforming OCL to Alloy, is that while the former is based on three-valued logic (i.e. there is a special *undefined* value to flag that the result of an operation cannot be determined), the latter is based on two-valued logic.
In order to transform OCL’s three-valued logic into Alloy’s two-valued logic, we make a counter intuitive, but necessary assumption, that the empty set evaluates to the same as the *undefined* value. This is due to OCL’s implicit conversion of scalars to sets. In particular according to the OCL specification [108, p. 81], if a collection operation is applied on a scalar, the scalar is automatically converted to a set. Assume the following two statements in the context of the Company, in the example shown in Figure 5.1:

```plaintext
inv collSize: self.employee -> size() = 0
inv undef: self.employee.oclIsUndefined()
```

Both *collSize* and *undef* invariants evaluate to *true* if no Person is related to the Company. Considering the empty set equal to the undefined value, is an important assumption that allows us to transform OCL statements to Alloy. Such an assumption has also been made by Akehurst et al. [18] in their translation of UML into Java 5.

An important remark is that in OCL equality is defined as a strict operation [108, Sec. A.2.2]. This means that the statement: ‘*expr1 = expr2*’, will return an undefined value if either *expr1* or *expr2*, or both are undefined. However, using our assumption, that the undefined value equals to the empty set, such an equality expression will always return true! It is recommended that in such cases the modeller uses the *oclIsUndefined()* OCL operation to query whether the left or the right hand side of an equality expression denotes an undefined value.

In general the *undefined* value in OCL is used to denote the absence of a value or a run time error [108, Ap. A]. If it is important to explicitly model the absence of a value from a model element, it is possible to do so on the model level.
Remark

Another issue related to the treatment of undefined values by OCL, is the fact that the OCL standard enforces implicit conversion between scalars and sets, when a set operation is applied on a scalar [108, p. 81]. More specifically, assume the following OCL statement fragment evaluated in the context of a Company that does not have a VAT number: ‘self.vatNumber’. This will evaluate to an undefined value. However, if a collection operation was applied on it:

\[
\text{self.vatNumber} \rightarrow \forall \text{v: String } | \text{ v = '123'}
\]

the statement would return true. Using Alloy’s uniform semantics, we do not encounter this problem; instead it can be considered that we treat all OCL statements as if a collection operation is used.
CHAPTER 6

TRANSFORMATION OF OPERATIONS

Alloy is a language ideal for modelling static systems involving complex logical constraints [74]. We consider static systems, as systems which are evaluated over a single state (i.e. the values of the elements of the system do not change over time). Alloy does not have an inherent notion of state machines to capture the evolution of the system from one state to another. According to the developers of the language the reason for this, is that they did not want to the modeller to tie a specific modelling pattern of dynamic systems [77, p. 173]. In this chapter existing idioms and methods to model the dynamics of systems directly in Alloy are presented. We also discuss how the dynamics of a system can be modelled using OCL pre and post conditions, so that they can be parsed by UML2Alloy.

6.1 Static vs. Dynamic Systems

This section briefly introduces the difference between modelling static and dynamic systems in Alloy and the different stances of UML and Alloy on this issue. We refer to models as ‘static’, when they evaluate over a single state (i.e. the objects of the system as well as the values of the attributes and association ends do not change over time). Conversely we consider as ‘dynamic’ the systems where either the objects or the values of the properties of
the objects of the system change over time. Assume the UML model of Figure 6.1, taken from Gogolla et al. [63]. Using the transformation rules presented in the previous chapters, this model can be transformed to an equivalent Alloy model.

In this model, initially a Person ‘p1’ is not married. Therefore the marital status of the person (marstatus attribute in Figure 6.1) is set to SINGLE. This can be expressed with the Alloy statement: ‘p1.marstatus = SINGLE’. However, when a Person gets married the status changes to MARRIED, which can be expressed with the Alloy statement: ‘p1.marstatus = MARRIED’. Since the Alloy language does not have any inherent notion of states and Alloy fields are immutable [77, p. 173], the two previous statements would lead to a logical inconsistency, because they specify that a person is single and married at the same time.

In contrast to Alloy, a UML class diagram has inherent support for states [108, Section A.1.2.4]. In particular, an instance of a class diagram represents the state of the system at a particular point in time and is provided by the structure: $\sigma(M) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATTR}}, \sigma_{\text{ASSOC}})$ [108, Section A.1.2.4], where:

- $\sigma_{\text{CLASS}}$ is the set of object ids of the objects that make up the system state.
- $\sigma_{\text{ATTR}}$ is the value of the attributes of the classes participating in the model.
- $\sigma_{\text{ASSOC}}$ are the instances of the associations that connect the objects.

In order to express the changes in the model over time, the UML standard provides the ability to specify pre and post conditions. Pre and post conditions are OCL expressions and they specify in a declarative way the effect of a UML operation. More specifically, if the precondition of an operation is satisfied on the invocation of an operation, then the postcondition of the operation must be satisfied at the end of the operation.

For example, to express the fact that a person is initially single, but then gets married in the class diagram of Figure 6.1, we could specify an operation marry:
context Person::marry():void

pre mPre: self.marstatus = Status:SINGLE

post mPost: self.marstatus = Status::MARRIED

According to the UML specification, this operation will evaluate over two states, one state for the pre condition and one state for the post condition [108, Section A.1.2.4]. The inherent notion of states in the UML specification makes it easy to express the change of values of attributes in UML class diagrams, through OCL pre and post conditions. On the other hand, the lack of inherent support of dynamics in the Alloy language requires the modeller to specify the notion of state on the model level, in an Alloy model. A number of approaches on how to specify dynamics in Alloy have been proposed in the literature [77, 144, 132, 131, 54]. The following section presents those approaches and discusses how our UML to Alloy model transformation can take advantage of such methods.

6.2 Existing Approaches

6.2.1 Global State

The global state idiom presented by Jackson [77] (also see [144]) proposes the specification of the notion of a global state on the model level. More specifically, the UML class diagram of Figure 6.1, would be represented with the UML class diagram of Figure 6.2. In this model a State class has been added on the model level and an association that relates the Person
class with the State class was inserted. This is also the approach we followed in some of our earlier work [30, 29]. To represent the class diagram of Figure 6.2 in Alloy, we use the transformation rules presented in the previous chapters, which produce the Alloy model of Figure 6.3.

Line1 of the produced Alloy model shows that the Alloy ordering library [77] is used. The ordering library is an Alloy module that comes with the Alloy Analyzer distribution and enforces ordering over the atoms of the signature on which it is applied (in this example the State signature). The ordering library also provides a number of methods to query the next and previous state of the system. Lines 2-12 of the model are produced according to the transformation rules presented in the previous chapters. Lines 13-15 depict the specification of the marry operation. This operation has been specified so that it accepts two State parameters, s and s'. The first parameter will represent the state of the system before the execution of the marry operation (i.e. when the preconditions of the operation are satisfied) and the second parameter will represent the state of the system after the execution of the operation (when the postcondition is satisfied)\(^1\). Finally lines 16-17 of the produced Alloy model ask the Alloy Analyzer to produce a random instance of the system, when the specification of the marry operation is executed. In particular, the simulation command requires that for every state in the system there is a next state, where the marry operation is invoked. The simulation command is specified for a scope of 2, which means that the analyser will attempt to simulate the marry operation on over two states only.

Figure 6.5a depicts a random instance of the model, produced by the Alloy Analyzer. State0 is the state before the execution of the marry operation and is related to a Person. State1 is the state after the execution of the marry operation. It can be seen that the person of

\(^1\)The the names of the parameters (s and s’) were randomly chosen. In Alloy a primed variable does not necessarily denote the value of a variable in a postcondition as in other languages (for example in Z [149])
State0 is single, while the person of State1 is married.

However, modelling dynamics using the global state pattern can be problematic. More specifically, using this pattern the ‘marstatus’ field of the Person signature is still immutable (i.e. its properties cannot change over time). As a result, the Alloy Analyzer uses two different person atoms to represent the same person (as seen from the instance shown in Figure 6.5a). In particular, Person1 represents the person before the execution of the marry operation, while Person0 is used to represent the person after the marry operation. It is clear that in this case our assumption made in Section 4.1.2, that a UML object id is equal to an Alloy atom is no longer valid, because the same person object is represented by two completely different Alloy atoms. Moreover, using this approach the transformation of the association multiplicities presented in Section 4.1.3, needs to be changed. For example, the multiplicity facts for the relations wife and husband (lines 10-12) should be changed, to express that a Person may be married to at most one Person, in each state of the system, not necessarily in the whole lifetime of the system (since someone can marry, divorce and re-marry another person).

![Figure 6.2: Modelling Global State on Class Diagrams](image)

### 6.2.2 Local Time

To overcome those issues with the global state pattern, Taghdiri and Jackson [132] have developed another pattern, based on local time. The local time pattern (or Tick-based mod-
open util/ordering[State] as ord

sig State( person: one Person )

sig Person( marstatus: one Status,
gender: one Gender,
alive: one Bool,
name: one String,
wife: lone Person,
husband: lone Person )

....

fact{wife = ~husband}

fact{wife in Person lone -> lone Person}

fact{husband in Person lone -> lone Person}

pred marry(s,s’:State){
(s.person.marstatus=SINGLE) &&
(s’.person.marstatus=MARRIED) }

run { all s:ord/first[],s’:ord/next[s] |
marry[s,s’] }

Figure 6.3: Alloy Model of Figure 6.2

elling [131]), proposes that all dynamic Alloy fields are related to a signature representing the time. The Alloy Analyzer ordering module is used to apply ordering on the atoms of the signature representing the time. This pattern allows for better modularity in the specification [132]. Moreover as we will explain in the following, using this pattern we can conveniently represent UML concepts in Alloy.

Using this pattern the attributes or association ends which may change over time are related to a Time signature. Total ordering is applied on the atoms of the Time signature, using the Alloy Analyzer ordering module. To show this, consider the UML class diagram of Figure 6.1, which will be transformed to the Alloy model of Figure 6.4. Line 1 shows that total ordering will be applied over the instances of time. This is required to ensure that time $T0$ represents the time before time $T1$. Line 2 declares a signature Time, while lines 3-9 show the Person signature with its fields. Notice that the fields, which might change over time, are related to the Time signature. For example, line 4 shows that the marstatus field is made of
the triple: \(\{ (\text{Person}, \text{Status}, \text{Time}) \}\). Multiplicity keywords are used to restrict that a Person can have one status for every Time. Moreover all constraints of the model need to hold for all instances of the Time signature. As a result, the multiplicity constraints (see lines 11-13) are evaluated for each instance of the time.

Lines 14-16 show the specification of the marry operation. Class operations are transformed into Alloy in a similar way that class invariants are transformed. In this example, an instance of Person is passed as a parameter to the operation to express the notion of context (the parameter self). Moreover two instances of the time signature are passed as parameters to the operation (i.e. \(t\) and \(t'\)), to express the notion of time before the operation is executed (when evaluating the precondition) and after the operation is executed (when evaluating the postcondition). Line 15 expresses that on time \(t\) the value of the marstatus field of the person self is SINGLE, while line 16 expresses that on time \(t'\) the value of the field will be MARRIED.

Finally lines 17-20 depict the command, which will be used for simulation. The command expresses that on the first (i.e. time \(t\)) and second (i.e. time \(t'\)) ticks of the clock, there is exactly one person \(p\). The marry operation is invoked on the person \(p\), using time \(t\) as the time when the precondition of marry is evaluated and time \(t'\) as the time when the postcondition is evaluated. The scope is set to 2, which means that two time instances will be used to evaluate the operation.

Figure 6.5b shows an instance of the model, produced by the Alloy Analyzer. It is clear that the marstatus field has two values, the SINGLE at time \(T0\) and MARRIED at time \(T1\). The application of the ordering module on the Time signature (line 1 of the model) guarantees that \(T0\) occurs before \(T1\).

This idiom of modelling the dynamics has certain benefits over the global state idiom. First, one Person atom is used to represent the Person over time, instead of two in the global
state pattern. Consequently our assumption described in Section 4.1.2, that the notion of object id corresponds to the Alloy notion of an atom, still holds. Additionally, as shown in this example (lines 11-13), using tick-based modelling pattern it is easy to express multiplicity constraints.

6.2.3 Extensions to the Language

The lack of inherent support for the notion of states in the Alloy language provides the modeller with the flexibility of choosing how to model the dynamics [77]. On the other hand, as demonstrated above modelling dynamics in Alloy requires a considerable amount of overhead. To overcome this issue Frias et al. [54] [52] have extended the Alloy language with actions of the form:

precondition

Figure 6.4: Tick-Based Alloy Model of Figure 6.1
This means that if the precondition formula is satisfied, and action is executed and the operation terminates, then the postcondition formula needs to be satisfied. The extension, which is based on Dynamic Logic [65], is powerful enough to allow sequential composition, iteration and non-deterministic choice of actions. A tool called DynAlloy has also been developed by Frias et al. [53], to support the language.

This approach seems appropriate for the analysis of an action language, with the help of Alloy. However, OCL is a side effect free language and it specifies the effect of an operation on the system state purely with the use of pre and post conditions. Consequently this approach is not well-suited for the OCL to Alloy transformation we are employing here. In the next section we show how our transformation treats OCL pre and postcondition specifications in Alloy.
6.3 Modelling Dynamic Systems\(^2\) in UML2Alloy

A review of the existing approaches on modelling dynamics in Alloy, presented in the previous section, reveals that tick-based modelling is the most well-suited existing approach for the representation of OCL pre and post conditions in Alloy. In this section we present how we can transform OCL pre and post conditions in Alloy, with the help of a small example. It is important to note that the rules are presented in this section informally (i.e. we have not defined the rules in QVT or another similar notation) and the rules have not been implemented in the UML2Alloy tool presented in Chapter 7 yet. The implementation of the rules remains for a future release of the tool.

The UML profile for Alloy we have described in Section 3.1 specifies that by default all class attributes and association ends are immutable (i.e. they are considered read-only; once the value is set it cannot be changed). In order to express that the value of an attribute or an association end might change over time, the modeller needs to use the ‘dynamic’ stereotype introduced in Section 3.1.2. The use of this stereotype ensures that the Alloy field generated from the class attribute or association end, will be related to the \textit{Time} signature.

\textbf{An Example Class Diagram with Pre/Post Conditions}

Assume the toy example of Figure 6.7 taken from Gogolla et al. [63]. The example shows a person who has a \textit{Status} (e.g. single or married), may have a wife or a husband. The \textit{marstatus} attribute and the husband and wife association ends are stereotyped as ‘dynamic’, because their value may change over time (i.e. a person may get married or divorced), unlike the \textit{gender} attribute of the Person, which cannot change over time. The \textit{prevSpouse} association end denotes a collection of the people that a person was previously married to.

\(^2\)As explained in Section 6.1 we consider as ‘dynamic’ the systems where either the objects or the values of the properties of the objects of the system change over time.
class defines two operations, the `marry` operation, which specifies what happens when a person gets married and the `divorce` operation, which specifies what happens when a person is divorced.

The specifications of the two operations are shown in Figure 6.6. More specifically line 1 specifies the context of the OCL statements, which is the operation `marry`. The `marry` operation is of type `Boolean`. This is necessary in order to be able to reference the operation from within other operations, as explained in page 138. Lines 2-7 express the preconditions that need to be satisfied for the operation to be executed. In particular the person on whom the operation is applied cannot have already been married and his/her gender must be different from the gender of the person (s)he is marrying. After the `marry` operation has finished (lines 8-14), we require that the `marstatus` of the person is set to `MARRIED` and the person is linked to the husband or wife (s)he married. Finally, line 15 adds the new husband or wife to the collection of the previous spouses of the person and line 16 does the same for the spouse.
Figure 6.7: Application of UML2Alloy Profile on Dynamics

Now we want to simulate the application of the *marry* operation on a person. Recall from Section 3.1.2 that a simulation command is an OCL invariant stereotyped as *simulation*. Consequently, we need to reference the *marry* operation from within an OCL invariant stereotyped as ‘*simulation*’. This is done with the OCL statement of lines 17-21 in Figure 6.7.

This OCL statement, when transformed to an Alloy simulation command requests the Alloy Analyzer to find two random persons, *p1* and *p2*, where *p1* marries *p2*, using the specification of the *marry* operation. In the following we describe an algorithm of how operation pre and post conditions are transformed to Alloy, with the help of the simple example presented in this section.

**Algorithm for the Transformation to Alloy**

The example UML class diagram of Figure 6.7 is transformed to an Alloy model, using the local time pattern explained above. The transformation is carried out using the following algorithm:

1. Using the rules presented in the previous chapters of this thesis we transform all model
elements of the class diagram, except for class attributes and association ends stereotyped as ‘dynamic’ and operation specifications.

2. All class attributes and association ends stereotyped as dynamic are related to a Time signature. Ordering is applied on the atoms of the Time signature, using the Alloy Analyzer ordering module.

3. Every UML class operation of type void or Boolean is transformed to an Alloy predicate. Similarly the parameters of the UML operation are transformed to predicate parameters.

4. An additional parameter is added to the list of the parameters of the generated Alloy predicate. This parameter is used to represent the context of the operation and is called self.

5. In addition to the context parameter, two additional parameters are added to the generated Alloy predicate. The parameters called t and t’ and represent the time when the pre condition and post conditions are evaluated respectively.

6. Class invariants where attributes or association ends stereotyped as ‘dynamic’ take part, are quantified over all instances of the Time signature.

7. Invariants stereotyped as ‘simulation’ or ‘assertion’ can reference an operation. These are transformed to Alloy as ‘run’ or ‘assertion’ commands respectively. Moreover two time instances, representing the first and second ticks of the clock are passed as parameters to the operation being referenced.

8. The scope for the Time signature is set according to the ‘time’ tagged value (for more information on the ‘time’ tagged value please refer to Section 3.1.2).
open util/ordering[Time] as ord
sig Time{}
...
sig Person{
  marstatus: Status one -> Time,
  gender: one Gender,
  husband: Person lone -> Time,
  wife: Person lone -> Time,
  prevSpouse: Person set -> Time}

fact{all t:Time | wife.t in Person lone -> lone Person}
...
pred marry(spouse:Person, self:Person,t:Time,t’:Time) {
  self.marstatus.t != MARRIED and
  self.gender != spouse.gender and
  spouse ! in self.prevSpouse.t and
  #self.husband.t = 0 and #self.wife.t=0
  #spouse.husband.t =0 and #spouse.wife.t=0

  //POSTCONDITIONS
  self.marstatus.t’ = MARRIED and
  self.prevSpouse.t’ = self.prevSpouse.t + spouse and
  spouse.prevSpouse.t’ = spouse.prevSpouse.t + self and
  ((self.gender = FEMALE) =>
   (self.husband.t’ = spouse and self.husband.t’.marstatus.t’ = MARRIED))
  else
   (self.wife.t’ = spouse and self.wife.t’.marstatus.t’ = MARRIED))
}
run {let t=ord/first[] |
  {some p1,p2:Person | p1!=p2 and init[p1,p2,t,ord/next[t]} for 2 but 2 Time}

Figure 6.8: Pre/Post Conditions of Figure 6.6 in Alloy

Using this algorithm the UML class diagram of Figure 6.7 and the pre and post conditions of Figure 6.6 have been transformed to the Alloy model of Figure 6.8. Following Step 1 of the algorithm, all UML classes and datatypes are transformed to Alloy signatures. Using Step 2 of the algorithm, lines 5 and 7-9 of the Alloy model are generated. According to Step 3 of the rule, the predicate of line 12 is generated. Using Step 4 of the rule, the self parameter of the predicate is created, while using Step 5 of the rule, t and t’ are added to the list of parameters of the predicate. Step 6 of the rule is responsible for creating the multiplicity facts of the dynamic elements of the model, such as those of line 10. Using Step 7 we transform the simulate invariant to the Alloy run command shown in lines 26-27 of the model. Finally, using Step 8 of the algorithm a scope of 2 is set for the Time signature (line 27 of the model).
Analysing the Example

The Alloy model generated using the algorithm presented above, can now be analysed using the Alloy Analyzer. Figure 6.9a depicts a simulation of the model, showing a random instance before the marry operation is applied and Figure 6.9b shows a random instance of the model after the marry operation. In this simulation there is a divorced woman (Person0) and a windowed man (Person1) that get married. In the following we show how to chain operations (i.e. simulate the execution of one operation after the other).

![Figure 6.9: Simulation of the marry Operation](image)

Chaining Operations

Here we will show how we can simulate the effect of two or operations executing one after the other, or in parallel. In the OCL to Alloy transformation algorithm for pre and post conditions, presented above, we discussed that every operation is evaluated over two time instances $t$ and $t'$. Time $t$ refers to the time when the precondition is evaluated, while time $t'$ refers to the time when the postcondition is evaluated. Using this pattern, it is possible to chain operations and call one after the other. For example, assume we want to simulate that a Person gets married and then gets a divorce in the example UML class diagram of Figure 6.7. To achieve this simulation, all we have to do is add a call to the `divorce()` operation at the end of the postcondition of the `marry()` operation. By doing this we specify that, $t$ is the time when the
precondition of the marry operation is evaluated and \( t' \) is the time when the postcondition of the operation is evaluated. Moreover, since the divorce operation is now referenced in the postcondition of the marry operation, the time when the precondition of the divorce operation is evaluated is \( t' \) and the time when the postcondition of the divorce operation is evaluated is \( t'' \) (i.e. two ticks of the clock since time \( t \)). Using this pattern it is possible to model the case of two operations being executed sequentially (i.e. one after the other). The whole marriage example with the chaining of the marry and divorce operations are shown in Appendix H.

**Remarks**

As explained in Section 3.1, the UML profile for Alloy, enforces that class attributes and association ends are read only, unless they are stereotyped as ‘dynamic’. This ensures that the modeller will have to specify frame conditions [31] only for those attributes and association ends that may change over time.

An interesting remark is about the interpretation of OCL pre and post conditions. In particular, OCL pre and post conditions can be interpreted in two ways, the \( \text{precondition} \Rightarrow \text{postcondition} \) or the \( \text{precondition} \land \text{postcondition} \) [69]. Using the \( \text{precondition} \Rightarrow \text{postcondition} \) interpretation, if the precondition holds then the postcondition needs to hold at the end of an operation. However, if the precondition does not hold the postcondition may be either true or false. Using the second interpretation (\( \text{precondition} \land \text{postcondition} \)), if the precondition is not satisfied, there is no state that satisfies the postcondition. The OCL specification adopts the second interpretation [108, Sec. A.3.2.2]. Consequently, the transformation to Alloy presented in this chapter adopts the same interpretation. In particular, if the precondition of an operation is not satisfied, then there is no instance where the postcondition is satisfied. In such a case (i.e. if we try to simulate an operation where the preconditions are not satisfied) the Alloy Analyzer will not produce an instance of the model and using the Unsat Core [126] facility of the tool, it is possible to inspect why the precondition is not satisfied.
In the example presented in this section we have shown how to transform operations of type Boolean. However, operations of other types can be transformed to Alloy similarly. In particular, in OCL the special variable result [108, Sec. 7.3.4] is used to represent the result of an operation (in a similar manner the return statement is used in Java to represent the value the operation will return). Such operations can be transformed to Alloy functions [77, Sec. 4.5.2]. The only difference between an Alloy predicate and a function is that while the former is a formula that denotes a boolean expression, the latter denotes a formula of a specific type.

Another important remark is that the OCL standard forbids the referencing of non-query operations from within an OCL statement. This is too restrictive for our approach, since it does not allow us to reference operations one after the other as described in page 138. In order to be able to simulate and reason about UML models we have to extend OCL to be able to reference other non-query OCL specifications from within a pre or postcondition. To achieve this, we use the approach of Nunes [103], which makes this extension to OCL and provides its formal semantics. The main benefit of the extension is that it makes OCL more expressive. Additionally this extension is compliant with the way Alloy treats predicates.

In this chapter we have discussed how to reference operations from the pre or post conditions of other operations in order to chain operations. This method has a shortcoming if we want to reference the same operation twice. For example, assume that we want to execute sequentially the operations \emph{a()}, \emph{b()} and \emph{a()}. This means that we have to reference \emph{b()} in the postcondition of \emph{a()} and \emph{a()} again in the postcondition of \emph{b()}. In such a case, the operation \emph{a()} is called recursively, which is not allowed by Alloy. To overcome this problem, we would have to be able to express that the operation \emph{a()} will evaluate on time \emph{t} and \emph{t'}, the operation \emph{b()} will evaluate on time \emph{t'} and \emph{t''} and the second call to the operation \emph{a()} will be evaluated on time \emph{t''} and \emph{t'''}. However, this is not possible with the method presented in this chapter. Alloy provides the ability to analyse various aspects of the dynamics of systems (e.g. whether two
operations commute as demonstrated by Dennis et al. [41]). Investigating the expressiveness of the approach presented in this chapter and extending it, remains for future investigation.
CHAPTER 7

UML2Alloy: Implementation of a Transformation Framework

UML2Alloy is our implementation of the transformation rules presented in the previous chapters. This chapter introduces the implementation of UML2Alloy and a number of case studies conducted with the help of the tool. In addition to the case studies presented in this chapter, UML2Alloy has also been used by Babkin [26] and Pons and Garcia [118] for the automated transformation of UML class diagrams and OCL constraints into Alloy.

7.1 Architecture

Figure 7.1 depicts an overview of the approach as described in the previous chapters. In particular, the UML and OCL metamodels are mapped into the Alloy metamodel using a number of QVT rules. A MOF2Text transformation is then used to map an instance of the Alloy metamodel to the Alloy textual representation, so that it can be used by the Alloy Analyzer.

Figure 7.2 depicts an overview of the implementation of the approach. More specifically, the Kent Modelling Framework (KMF) [4] XMI parser and the Kent OCL library, which is
part of KMF were used in order to read a UML class diagram with its corresponding OCL statements. The KMF contains a set of Java classes that represent the UML and OCL metamodels. At the time of the implementation of UML2Alloy there were no available tools fully compliant with the QVT standard. As a result, we used the Simple Transformer (SiTra) [17] transformation engine to define the transformation rules. The Alloy metamodel was implemented in the Java programming language. This is in essence a representation of the Abstract Syntax Tree (AST) of the Alloy language in Java. Finally an implementation following the visitor pattern [55] was used to generate the textual Alloy model. The rest of this section describes the implementation of UML2Alloy in more detail.

UML2Alloy works by reading a UML class diagram with OCL constraints, which it maps to an Alloy model. The UML model is input in XMI 1.2 format [104] and the Alloy model is output as a text file. Earlier versions of UML2Alloy required the user to interact with the
Alloy Analyzer in order to carry out the analysis. Recently we have added the facility to analyse the generated Alloy model, from within UML2Alloy, by interacting with the Alloy API.

Figure 7.3 shows a high level view of the components used by UML2Alloy. The tool uses of the Kent Modelling Framework (KMF) [4] XMI reader and OCL parser. The KMF XMI reader is used to read the XMI file input to the tool and populate the Java classes that represent the UML and OCL metamodels. The core of UML2Alloy defines the transformation rules, by using the Simple Transformer (SiTra) [17] transformation engine. The transformation rules map the UML/OCL metamodel elements to instances of the the Alloy metamodel. Any errors and warnings that occur during the transformation, are logged using the Apache Log4j component.

In order to implement the transformation it was necessary to extend the SiTra transformation engine. The following section describes the motivation behind the extension of SiTra and the details of the implementation.
7.2 Extending SiTra for UML2Alloy

Simple Transformer (SiTra) has been developed by Akehurst et al. [17] and is a simple and lightweight implementation of an extensible transformation engine. Figure 7.4 provides a conceptual outline of the framework. SiTra consists of two interfaces, the Rule interface, which user defined mapping rules have to implement and the Transformer interface, which provides the skeleton of the methods that carry out the transformation. The authors of SiTra provide a simple implementation of the Transformer interface and to use SiTra for simple transformations the modeller only needs to define the transformation rules by implementing the Rule interface, which consists of three methods as depicted in Figure 7.4. If the rule is applicable for the source element in question, the check() method of the rule interface returns true and the build() method is executed. The build() method generates the target model element. The setProperties() is used to set the attributes and links of the newly created target element. For further details on SiTra please refer to [17].

In our implementation it was necessary to extend the SiTra distribution with some additional functionality, in order to take full advantage of the features the Alloy Analyzer provides. More specifically one of the most powerful features of the Alloy Analyzer, is the Unsat Core [126] facility. In the case of inconsistent models (i.e. models for which the Analyzer cannot provide an instance), the Unsat Core functionality can locate the conflicting statements.
that cause the inconsistency. In order to take advantage of this feature we have implemented Model to Model (M2M) and Model to Text (M2Text) tracing.

Figure 7.5 illustrates a scenario of how the Alloy’s Unsat Core functionality can be used in our UML2Alloy transformation framework. Initially the UML/OCL model is transformed to an instance of the Alloy metamodel. This is a Model to Model (M2M) transformation. The instance of the Alloy metamodel is then transformed to the Alloy textual notation, through a Model to Text (M2Text) transformation. The Alloy Analyzer API is then used to analyse the Alloy model. If the analyzer cannot produce an instance, it returns the positions of the conflicting statements in the Alloy textual representation. Using the M2Text tracing, we can trace back those positions to the Alloy metamodel elements responsible for the inconsistency. Finally, using M2M tracing we can trace back the original UML/OCL model elements responsible for the conflict. In the following we explain how this tracing extension was implemented in UML2Alloy, by extending the SiTra framework.

**Traceability**

Traceability is a feature in model transformations and has recently received considerable attention [80, 88, 47, 114]. In model transformations traceability is typically used during the execution of a model transformation to keep track of which element(s) of the source model(s)
has been transformed to which element(s) of the target model(s) and the reverse. Traceability can have many applications in the context of model transformations. For example, when executing a model transformation for a second time after the source model changes slightly, traceability links can be used to update only the necessary elements of the target model without having to execute the whole transformation again. For an extended classification of traceability approaches please refer to [114]. Traceability is also a standard feature in state of the art model transformation frameworks, such as ATL [80], Epsilon [88] and Kermeta [47].

Traceability is also an important feature in Model to Text transformation frameworks, such as MOFScript [106] and the Epsilon Generation Language [122]. These frameworks generate trace links to identify which model elements have been transformed to what text. Traceability in Model to Text transformations is particularly important to keep a model and the text to which it corresponds synchronised. As discussed in detail in this section, our work generates trace links to identify which part of the generated Alloy textual model has been created from what Alloy metamodel element.

The Queries Views and Transformations (QVT) specification [107] defines a tracing mechanism that can be used to trace which source metamodel elements are mapped to which target metamodel elements and the inverse. Figure 7.6 depicts the tracing model we have developed, to trace the model to model transformation between the UML/OCL metamodel and the Alloy metamodel.
We have defined an interface $I\text{Trace}$, which holds a collection of $Trace\text{Instances}$ ($ts$). Each $Trace\text{Instance}$ represents a mapping between a source and a target model element, through a SiTra rule. The $I\text{Trace}$ interface implementation, should provide a number of methods to query the $ts$ collection. More specifically the $\text{resolve}$ method, should query the $ts$ collection, and return all target instances that have been created during the transformation, from the src instance. Likewise, $\text{resolveone}$ should return only the first instance of the target element that has been created during the transformation from the src instance. The method names preceded with $\text{inv}$ (i.e. $\text{invresolve}$), perform the inverse (i.e. return the source elements that have been mapped to the target element passed as a parameter). The QVT specification also defines a number of additional methods that can be used to query the trace model. For more information please refer to [107, Section 8.1.10].

A similar tracing model has been defined for our model to text transformation ($M2\text{Text}$), which transforms the Alloy metamodel elements to the Alloy textual notation. Each $Trace\text{-Instance}$ of the $M2\text{Text}$ trace model maps an Alloy metamodel element to a range in the text file. The range is identified by the rows and columns it occupies.

Figure 7.7 shows an example of how M2M and M2Text tracing work. The top left part depicts a simple UML class ($Person$), with an attribute ($age$ of type $Integer$). The top right part of the figure depicts an equivalent Alloy textual representation. This transformation is executed in three steps. In the first step, the class and its attribute are represented as an instance of the UML metamodel (as an object diagram). In the second step the UML class and its attribute is represented as an instance of the Alloy metamodel, after executing the M2M transformation. Finally, in the third step the instance of the Alloy metamodel is transformed to the Alloy textual notation using the MOF2Text transformation.

The bottom part of Figure 7.7 represents the tracing information recorded during the transformation. In particular, the $Person$ class is transformed to a $Person$ signature in Alloy.
sequently, the trace information recorded has as a source element the \textit{Person} class and as a target element the \textit{PersonSig}, which is of type \textit{ExtendsSigDecl} (i.e. a signature declaration). Similarly, for the \textit{age} attribute the trace information recorded has as a source element the \textit{Age} property and as a target element the \textit{AgeDecl}, which is of type \textit{Decl} (i.e. an Alloy field). Finally, M2Text tracing records how the \textit{PersonSig} model element is transformed to text. More specifically, the \textit{PersonSig} model element is transformed to the ‘\textit{sig Person}’ string, which occupies a certain range in the text file. Similarly M2Text tracing is used to trace how the \textit{AgeDecl} model element is transformed to text. The populated tracing model can then be queried to identify for example to which range of the Alloy text the \textit{age} attributed was transformed.

\textbf{Extending the trace model to non-binary transformation rules}

A close inspection of the \textit{TraceInstance} class of Figure 7.6, shows that we can use our mechanism only for binary rules (a \textit{TraceInstance} relates \textit{one} source element to \textit{one} target element). As a result, we cannot use this tracing mechanism to trace transformations with ternary rules (i.e. one source element mapping to two target elements). However, as we have demonstrated with our transformation, it may be possible to express ternary transformations, as two binary ones. More specifically as discussed in Section 4.1.4 a bidirectional binary UML Association is transformed to two separate Alloy facts. In particular, it is mapped to a multiplicity fact in Alloy constraining the number of instances of the Signatures taking part in the relation and an additional fact to show that the generated relation is symmetric. In our transformation we have implemented this ternary mapping, using two binary mappings, one that maps the association to the multiplicity fact and another mapping that maps the association to the symmetry fact. However, in cases where such an approach is not feasible, the tracing mechanism presented here needs to be extended.

Additional technical details, screenshots and detailed documentation on how to use the
Figure 7.7: Example of Model to Model and Model to Text Transformation
UML2Alloy implementation to automatically transform and analyse UML class diagrams with OCL constraints can be found in the UML2Alloy user manual, which is available on the UML2Alloy project website [8]. In the following section we present three studies on which UML2Alloy was applied.

7.3 UML2Alloy in Practice

Interfacing with the Alloy Analyzer the UML2Alloy implementation is able to simulate UML class diagrams with OCL constrains, reason on the validity of assertion statements and locate inconsistencies. In this section we present a number of case studies where UML2Alloy has been used, show the results and discuss our experiences.

The studies presented in this section demonstrate the feasibility of the proposed approach and serve as examples of the domains where the approach can be used. In particular in Section 7.3.1 we present how the approach can be used to provide a solution to a sudoku puzzle, taking advantage of the simulation capabilities of our approach. A similar study was conducted by Torlak and Dennis [136], where a sudoku puzzle was modelled directly in Alloy and a solution was produced. However in [136] the puzzle was developed with the analysis capabilities of the Alloy Analyzer in mind. In particular, the puzzle was modelled as a triple, where the first coordinate represented the row of the puzzle, the second coordinate represented the column, while the third coordinate represented the actual number of the cell. The study we present here uses UML class diagrams and models the row, column and the numbers of the puzzle as integer attributes. We believe this way of modelling the puzzle is closer to the way a software developer would model it with the implementation of the system in mind.

In Section 7.3.2 we also demonstrate that the proposed approach can be used in the domain of analysis of model transformations. In this study we analysed a transformation that
deals with a domain specific language used for business process modelling, utilised by IBM’s WebSphere Business Modeler [73]. This study demonstrates a method to analyse certain aspects of model transformations. It is possible to apply the method presented in Section 7.3.2 to analyse similar properties of other model transformations.

Finally in Section 7.3.3 we demonstrate how the approach can be used to check that a user requirement is satisfied by the design of the system. In particular the study presents the design of the ACTIVE [42] e-commerce platform using UML class diagrams and OCL constraints. The design considers a client and server communicating over a secure connection and a man-in-the-middle intercepting the communication between the client and the server. In this scenario a user requirement is that the man-in-the-middle does not have access to confidential information exchanged between the client and the server.

### 7.3.1 Precise UML for Sudoku

Sudoku is a popular puzzle game and it is an ideal toy application for our approach as the puzzle rules can be expressed in UML and OCL. Moreover it is considered as a benchmark application by the constraint solving community [94] and has been directly applied to the Alloy paradigm [136]. Here we demonstrate how such a puzzle can be solved with the use of UML and OCL.

A sudoku puzzle consists of a square grid. Each cell of the grid has to be filled with a number and a number has to appear only once in a row, column or region. Initially the grid is partially complete and the player is required to complete the rest of the puzzle. In this study we investigate how such a puzzle can be solved using precise UML with the help of UML2Alloy.

Figure 7.8a depicts a 6x6 sample sudoku puzzle and Figure 7.8b shows the UML class diagram that represents the puzzle. This puzzle consists of 36 cells. Each cell has a rowIndex,
which represents the row of the cell, a *columnIndex*, which represents the column of the cell and a *value*, which represents the contents of the cell. Finally, a cell belongs to a *region* and each region is represented by an integer number.

OCL is used to specify the constraints of the puzzle and the numbers that exist already in the puzzle. The following OCL statement depicts the rules of the puzzle:

```ocl
context Cell
inv rules :
Cell.allInstances() -> forAll (c1, c2: Cell |
( ( c1.rowIndex = c2.rowIndex or
c1.columnIndex = c2.columnIndex or
c1.region = c2.region) and
( c1 <> c2 )) implies
 c1.value <> c2.value)
```

The OCL statement requires that for every two cells, *c1* and *c2*, if they belong to the same row or column or region, and they are not the same, then the numbers of *c1* and *c2* are different. The complete UML model and OCL constraints of this puzzle and the generated Alloy model can be found in appendix B.

UML2Alloy was used to transform the UML representation of the puzzle to Alloy. The Analyzer provided a solution to the puzzle. The solution the Analyzer found is illustrated in Figure 7.9.

An interesting observation is the Analyzer required 213.660ms (around 3.5 minutes) to solve this simple 6x6 puzzle. This is probably due to the extensive use of integer types in the model. We developed another model of the same puzzle without integer types. Instead,

---

1The sample puzzle of Figure 7.8a has 6 regions. The region borders are designated by the bold lines.
we used an enumerator type with enumeration literals to represent integer values (i.e. *ONE*, *TWO*, *THREE* . . . ). It took the Alloy Analyzer less than 5 seconds to produce a solution for the version of the puzzle without integer numbers. It is therefore recommended that integer types are not used in a model to be analysed by UML2Alloy unless it absolutely necessary.

7.3.2 Analysis of Model Transformations

An important application of our approach is in the domain of the analysis of model transformation specifications. In [22] we present a systematic method of representing declarative model transformations in Alloy and demonstrate how the Alloy Analyzer can be used to conduct fully automated analysis of a model transformation specification.

Figure 7.10 depicts an outline of our approach, which is comprised of two steps. A model transformation can be seen as a relation between the elements of the source and the target metamodels [16, 107]. The first step of our approach is to convert the model transformation specification to an equivalent specification expressed in the Alloy language. The second step is to use the Alloy Analyzer to analyse the produced Alloy model. In the following we explain the first step of the approach in more detail.

A model transformation specification consists of a MOF compliant representation of the source metamodel, a MOF representation of the target metamodel and the transformation

![Figure 7.8: A Sample Sudoku Puzzle and its Class Diagram Representation](image)

(a) A Sample Sudoku Puzzle
(b) A Class Diagram Representing the Sudoku Puzzle
rules, which define the mappings between the source and target metamodels.

A MOF metamodel is usually accompanied by constraints, which define syntactic and semantic properties of the language (well-formedness rules). The OCL is commonly used to specify the well-formedness rules. As a result, it is possible to use the UML2Alloy approach presented in this thesis to transform the source and target metamodels to Alloy.

However, the transformation rules specification, which defines how elements of the source metamodel map to elements of the target metamodel needs to be expressed in Alloy manually. In [22] we show an example of how to represent declarative transformation rules to Alloy.\(^2\)

In order to assess the possibility of systematically representing model transformation rules in Alloy, a comparative study between the QVT specification and Alloy needs to be conducted. Such a study is outside the

Figure 7.10: An Outline of our Approach
When both the source and target metamodels, as well as the transformation rules are represented in Alloy the transformation can be analysed by the Alloy Analyzer. More specifically the simulation capabilities of the Alloy Analyzer can be used to create a random instance of the transformation. If the transformation specification is inconsistent, the Unsat Core [126] functionality of the Analyzer can be used to detect the rules that cause the inconsistency. Another advantage of this approach is the ability to reason about certain properties of the transformation. The property of the transformation that needs to be checked, can be formulated as an Alloy assertion. If the property is not satisfied by the transformation, the Analyzer presents a counterexample, which is an instance of the target model that violates the property. The counterexample can be inspected to deduce the flaw in the definition of the transformation.

Moreover our method can be used to simulate a model transformation by specifying an instance of the source metamodel in Alloy. Using this approach the Alloy Analyzer can produce an instance of the target metamodel. For more details on how Alloy can be used as a model transformation execution environment, please refer to [20].

We have applied the proposed method to the transformation presented in [89] by Küster and Abd-El-Razik. This transformation deals with a domain specific language used for business process modelling, utilised by IBM’s WebSphere Business Modeler [73]. The language is similar to UML Activity Diagrams [111].

The transformation [89] removes control actions (i.e. Decision, Fork, Join and Merge nodes) and replaces them with implicit control actions expressed with the help of pinsets. In [20] we transformed automatically the MOF source and target metamodels in to Alloy, while the transformation rules were manually translated to Alloy. Analysing the produced Alloy model we exposed that it is possible that the model transformation produces ill formed
Since with this approach we need to represent the source and target metamodels as well as the transformation rules in Alloy, it is expected that it will not scale well, when large metamodels and complex transformation rules are involved. In [20] we present some preliminary performance benchmarks using this approach on a case study.

A number of languages used for the definition of model transformations are imperative [17] and hybrid [82] (i.e. provide support for both imperative and declarative specifications). Alloy is a fully declarative language. As a result, it is not possible to directly map imperative model transformations to Alloy and analyse them. However, it may be possible to abstract an imperative transformation to a declarative one by removing the computational details of the transformation, which are not of interest.

Additionally the Alloy language has a simple type system. The only primitive types supported are Integers. As a consequence it is not possible to use Alloy to analyse complex properties involving certain types (i.e. String, Real numbers).

Finally, models in Alloy are static, i.e. they capture the entities of a system, their relationships and constrains about the system. An Alloy model defines an instance of a system where the constraints are satisfied. More specifically, Alloy does not have any built in notion of state machine [77, Ap. B.5.1]. As a result, our approach can only be used to reason about static properties of the transformation. For example, currently it is not possible to use our approach to reason whether applying a rule r1 before a rule r2 in a model, will have the same effect as applying r2 before r1. It is however possible to model dynamic systems in Alloy [77]. Extending our approach to reason about dynamic properties of transformations remains for further research.
7.3.3 Applying UML2Alloy to Secure Systems

UML2Alloy has also been successfully applied in the domain of modelling secure applications. In [58] we carried out a study in collaboration with leading researchers on the domain of Aspect-Oriented Modeling (AOM) and secure applications.\(^3\)

In this study a methodology is illustrated, which utilizes AOM techniques and UML2Alloy for the design of secure systems. More specifically the proposed methodology combines an application model (called the primary model) with an aspect model representing the security properties of the application. The primary and the aspect model are combined to create the misuse model. The misuse model, can be transformed by UML2Alloy to Alloy and then analysed by the Alloy Analyzer, to show whether the security of vital resources is compromised by an attack. This methodology has been applied to the ACTIVE [42] e-commerce platform.

The ACTIVE system allows a client to purchase products on the internet. A number of security attacks can be launched on such a system. The man-in-the-middle attack is one of them. In such an attack an attacker can intercept confidential information exchanged between the client and the server dealing with the purchase request. In order to avoid such attacks the security of the login service of the ACTIVE system, has been enhanced with the Transport Layer Security (TLS) [133] protocol, which is an improved version of the Secure Sockets Layer (SSL) [134] protocol. The TLS protocol utilises asymmetric cryptography, which means that the server needs to have a certificate signed by an authority. For the purposes of this study, certificate creation and all public and private keys are assumed to be obtained in a secure manner. The client must have the certificate authority’s (CA) public key, and the server must have a certificate, signed by the certificate authority, of its name and public key. Other assumptions include the fact that both unique identifier numbers called nonces, and session

\(^3\)Our contribution in this work is in the automated translation of the UML model to Alloy and the analysis of the system of the study, which demonstrates that the system preserves some security properties.
keys must change each time the protocol is initiated.

Figure 7.11 illustrates the abstract sequence diagram that represents the login sequence of the misuse model of the ACTIVE system\(^4\). In the misuse model an Attacker intercepts the messages exchanged between the client (ActiveClient) and the server (LoginManager). In the following we describe the login sequence.

**Login Sequence**

The sequence begins with the ActiveClient sending a login message that contains a nonce (iNonce), its public key (cPublicKey), and an encrypted version of its public key (encryptedCPublickey). The encryptedCPublickey, is encrypted by the client using its private key. The message is intercepted by the Attacker, which decrypts it using the client’s public key. The Attacker then re-creates the login message using the iNonce that existed in the login message it received (recINonce), its own public key (aPublicKey) and its own encrypted public key (encryptedAPublicKey). The Attacker then forwards the message to the LoginManager. The LoginManager receives the message and creates a new message containing the session key (sessionKey) that needs to be used for encryption once the login sequence is complete, the server name (lmName), the nonce received in the original message, and the public key extracted from the message it received (this is the attacker’s public key, as the attacker has inserted its own public key in the message). Then the LoginManager encrypts the message using the public key in the original message (i.e. the Attacker’s public key), signs the message using its own private key and sends it back to the Attacker. This message is decrypted by the Attacker, using its private key. After it accesses the information required, the Attacker re-encrypts the message, using the ActiveClient’s public key and sends it to the ActiveClient. The ActiveClient checks if the server name in the message it received is the same as

\(^{4}\)Here we only present our analysis of the misuse model. For more details on the proposed methodology of how the misuse model can be obtained from the application model, please refer to [58].
Figure 7.11: Abstract Sequence Diagram Depicting the Login Sequence
the server name contained in the server certificate. This check is carried out in checkAC1() method. If the names do not much the login procedure is aborted. Otherwise the ActiveClient extracts its public key from the signed portion of the message it received (encryptedPublicKey) and compares it to the signed version of the public key it sent (encryptedCPublicKey). This is reflected in method checkAC2(). If the comparison fails the login is aborted. Otherwise the ActiveClient sends a continueWithLogin() message, which once received by the LoginManager, results in the home page being returned to the ActiveClient. At this point the Attacker has access to sensitive information and if the login sequence reaches at this stage, it is compromised by the Attacker.

The Misuse Model

In order to be able to formally analyse the login sequence of the system using UML2Alloy, we need to represent it in a UML class diagram, enriched with OCL constraints. Figure 7.12 depicts the UML class diagram of the misuse model of the ACTIVE system. The messages of the sequence diagram have been transformed to equivalent OCL statements manually, but systematically. In particular every message that belongs to a different object lifeline has the potential to become a method in the OCL specification of the receiving object, if the object performs some computation of interest as a result of receiving the message. If the receiving object just passes the message through to another object lifeline, the method will exist in the final receiving object. Additionally every alt box of the sequence diagram of the misuse model can be represented by an if-then-else OCL constraint in the specification.

The following shows the OCL definition of the main() method of the ActiveClient class. The method specifies that the receiveLoginFromClient() method of the Attacker class related to the ActiveClient, should be invoked.

context ActiveClient :: main ( ) : Boolean
Figure 7.12: Misuse Model of the ACTIVE System

-- The main operation imposes that the receiveLoginFromClient() operation will apply to all of the ActiveClients;

ActiveClient.allInstances() -> forAll(ac:ActiveClient | ac.at.receiveLoginFromClient())

The rest of the methods in the model have similar specifications, in that they specify the values of the attributes of the classes, and invoke class methods. For example the following depicts the OCL specification of the receiveLoginFromClient() method of the Attacker.

context Attacker::receiveLoginFromClient():Boolean
post receiveLoginFromClient:

-- The Attacker reads the iNonce, the ActiveClient -- public key and the encrypted public key of the -- Client and calls the receiveLoginFromAttacker
-- method on the server

self.recINonce = self.ac1.iNonce and
self.recCPublicKey = self.ac1.cPublicKey and
self.recEncryptedCPublicKey = self.ac1.encryptedCPublicKey and
self.lm.receiveLoginFromAttacker ( )

In the following we present the results of our analysis.

**Analysis**

Our objective is to analyse the misuse model of the ACTIVE system. In particular we want to reason whether the presence of a man-in-the-middle can compromise any sensitive information on the messages exchanged between the client and the server. We assume sensitive information is not compromised, if the protocol always aborts in the presence of the man-in-the-middle. The following OCL statement is used as an assertion.

context ActiveClient
alwaysExits :
ActiveClient.allInstances() ->
forall (ac:ActiveClient | ac.loginAborted = ResultType::r_True )

Using our transformation rules, this assertion is mapped to the equivalent Alloy formula:

all ac: ActiveClient | ac.loginAborted = r_True

We ran this assertion for a scope of 6. The Alloy Analyzer came up with a counterexample (i.e. an instance where the protocol does not abort in the presence of the man-in-the-middle)!
After careful inspection of this counterexample, it became evident that the instance is produced when the encrypted public key of the client \((encryptedCPublicKey)\) is the same as the encrypted public key of the attacker \((encryptedAPublicKey)\). In practice this can never happen, unless the attacker has access to the client’s private key. So in order to avoid this situation, we included the following invariant in the model:

context ActiveClient

\[
\text{inv: ActiveClient.allInstances()} \rightarrow \\forall \text{ac:ActiveClient} \mid \text{ac.at.encryptedAPublicKey} \neq \text{ac.encryptedCPublicKey}
\]

After inserting this assertion the Analyser did not produce a counterexample for a scope of up to 20. The full UML and OCL specification can be found in Appendix C.

It is important to emphasise that the representation of the login sequence of the TLS protocol presented in this section is at a high conceptual level. For example it does not deal with the details of the encryption and decryption algorithms and the computations carried out during these phases. Here we analyse the high level conceptual model of the system.

In order to avoid the complication of having to define frame conditions [31], we have not introduced the notion of ordering in the produced Alloy model. Each participant in the protocol has a set of attributes that hold the values of the messages exchanged. A different attribute is used for the value of the message received and the message sent (for example \(encryptedPublicKey\) represents the encrypted public the \textit{Attacker} receives from the client and the \(recEncryptedCPublicKey\) represents the encrypted public key the \textit{Attacker} receives from the server). The tradeoff of this approach is that we cannot reason whether the sequence of messages exchanged between the entities posses a vulnerability to the system.

It is also essential to remember the fact that though Alloy did not find a counterexample for the scope of 20; it does not mean that a counterexample does not exists for a larger scope.
However if a counterexample is not found for such a large scope, is unlikely (though possible) to get a counterexample for a larger scope.

**Related Work on Analysis of Secure Systems**

A few languages based on UML that allow for the specification and analysis of system and security properties have been proposed. SecureUML [92] is such a language that specifies system and security models using an UML profile. Model transformations are used to generate system code from the models. The interactive theorem prover Isabelle is used to analyse and verify the model prior to code generation. Unlike our work, SecureUML is specifically meant for access control and authorisation properties. Our work is more generalised to check to any sort of security property as long as it can be expressed in first-order logic. Moreover our work is based on automated analysis. We make use of the Alloy Analyzer, which is not a theorem prover and does not require any user guidance to generate the result of the analysis.

UMLsec [83] is a UML profile that allows a modeller to specify security requirements using UML stereotypes and tagged values. It has an accompanying toolset that allows a design conforming to the profile to be verified. Such verification can be done using a theorem prover such as e-SETHEO [98]. The tool also provides the ability to generate an implementation of the routines that check the security properties. As demonstrated with the studies presented in the previous sections, our approach is not confined to analysis security properties.

Torlak et al. [139] have used Alloy to analyse man-in-the-middle attacks. They introduce the idea of *knowledge flow*, which models how knowledge (information) is exchanged between the participants of a protocol handshake. In particular they focus on the information that the man-in-the-middle can possess, irrespective of the order the messages are exchanged between the participants. The protocol under investigation was modelled directly in the Alloy language and analysed. They used their method on the Needham and Schroeder [100] protocol and verified the existence of a flow already discovered by Lowe [93]. Their method
was developed to prove certain properties of security protocols, thus they formally prove the completeness and soundness of their method. On the other hand, our approach was intended as a proof of concept for our transformation and makes use of notations such as sequence diagrams.

Alloy has also been applied to other protocols and systems. Jackson and Sullivan [78] have analysed the COM architecture and Khurshid and Jackson [84] have analysed the consistency of the International Naming Scheme (INS). There are also a number of systems originally modelled in UML, which have been manually transformed to Alloy for the purpose of analysis. Alloy has also been used for partially analysing the run-time configuration management of an Asynchronous Transfer Mode/Internet Protocol (ATM/IP) Network Monitoring System [59]. Dennis et al. [41] have used Alloy to analyse a radiation therapy machine, exposing major flaws in the original UML design of the system. Unlike these case studies, our work supports fully automated transformation of UML models enriched with OCL constraints, to Alloy, for the purpose of analysis.
CHAPTER 8

CONCLUSIONS

This chapter concludes the work presented in this thesis. Section 8.1 presents a synopsis of the thesis. In section 8.2 we draw some conclusions and present directions for future work.

8.1 Thesis Summary

This thesis presents a model driven approach for the automated analysis of UML models. In particular, the popular modelling notation of UML class diagrams enriched with OCL constraints, is transformed to Alloy for the purpose of analysis. The proposed approach uses the MDA technology to achieve fully automated analysis of UML class diagrams with OCL constraints.

More precisely, we first identified a number of UML class diagram and OCL notions expressible in Alloy. A UML profile for Alloy was then developed for two purposes. Firstly, the profile was designed to forbid the definition of UML notions not expressible in Alloy. Secondly, the profile was developed to allow the representation of Alloy concepts, such as the notion of ‘scope’ in UML, to facilitate fully automated analysis of UML class diagrams. Moreover an Alloy MOF metamodel (i.e. the abstract syntax of the Alloy language in MOF representation) was developed from the Alloy grammar.
A number of transformation rules have been developed to map elements of the UML and OCL metamodels to elements of the Alloy metamodel. The mapping rules are described using natural language, while the QVT specification language is used to provide a formal description of the mapping. Reasoning on why we mapped a specific UML element to a corresponding Alloy element is provided for each mapping rule. How a number of UML concepts, such as n-ary associations, can be expressed using simpler UML notions, which are supported by our transformation are also presented. Moreover, the mapping rules are extended to transform the dynamics of UML in Alloy. In the approach presented in this thesis the dynamics of UML are expressed using OCL pre and post conditions.

The details of the implementation of the rules in a tool called UML2Alloy are also presented. The tool is built on the SiTra lightweight model transformation engine. Babkin [26] and Pons and Garcia [118] report that they have successfully used an early version of the implementation to transform UML to Alloy. Finally, the feasibility of the approach is demonstrated with a number of case studies.

In the next section we discuss our experiences from the model transformation between UML/OCL and Alloy. We believe that our experiences can be generalised and applied in the domain of model transformations between languages with substantial differences.

8.2 Final Discussion and Future Work

Maintaining model transformations with evolving language specifications and metamodels is an important issue. More precisely, when this project was launched UML 2.0 [109, 111] was the latest version of the UML specification. Since then a new version of the UML specification has been released (UML 2.1.2 [110, 112] is the latest version). Additionally the Alloy language has moved from version 3.0 to version 4.0 with slight syntactic changes and new features. Updating the existing transformation rules to the latest versions of the languages
presents unique challenges to the model transformation community. In particular, it is possible to update the transformation rules by inspecting only the updated parts of the UML and Alloy specifications. However, tool support with relevant capabilities is necessary to carry out this delicate task. For example, it is important to be able to apply regression testing on the model transformation to ensure the updated language metamodels and the new transformation rules do not introduce any bugs in the model transformation implementation. We plan to evaluate whether existing model transformation frameworks have such capabilities in order to update UML2Alloy to the latest versions of the languages.

Often a model transformation is used to produce a model of a target language as an intermediate step in a process. For example, a model transformation may be used to transform a model from a source language $A$ to a target language $X$, to take advantage of better tool support for $X$ (for example, our transformation from UML/OCL to Alloy). In such a case, the toolset of the target language is used to process (e.g. to analyse or refactor) the produced model and the results of this process need to be interpreted in the domain/paradigm of the original language $A$. If the transformation between the source and the target languages is not bidirectional this is not a simple task. In such a case the traceability information recorded during the model transformation can be used to interpret the results of the analysis. In UML2Alloy, such traceability information is used to interpret the Unsat Core results provided by the Alloy Analyzer in the UML/OCL domain, as explained in Section 7.2.

An issue that requires special attention when transforming textual languages is the operator precedence rules. For instance, as discussed in Section 5.13 in OCL the operation ‘or’ has a higher precedence from the ‘implies’ operation, while in Alloy it is the other way around. This difference in the operation precedence rules can potentially change the result of an OCL expression when evaluated in Alloy. In order to avoid such situations, when transforming textual languages, extra parentheses need to be used to ensure that the parser that will parse
the target model will create a parse tree equivalent to the original model parse tree.

Another interesting remark is related to the expressiveness of our approach, depicted in Figure 8.1. As discussed in this thesis OCL has a number of notions (such as the iterate and casting operations), which cannot be expressed in Alloy and are thus not supported by our work. On the other hand, Alloy’s relational logic provides a number of operators to directly manipulate relations (for example it is easy to express the transitive closure or the transpose of a binary relation [77]). OCL does not directly support such operations. Therefore it is not possible to take advantage of the full expressive power of Alloy, without extending OCL. Consequently the expressiveness of the expressions supported by our work is the intersection of the expressions supported by OCL and Alloy.

Additionally the method presented in this thesis can also be used in heterogeneous specifications [113]. In particular, the UML standard allows a language other than OCL to be used for the specification of constraints in UML class diagrams. It is therefore possible to use UML class diagrams to depict the entities of a system and the relationships between them, and Alloy to specify constraints on those entities. Using this approach it is possible to use UML class diagrams as a graphical notation of Alloy’s textual models. The full expressive power of Alloy can then be used (i.e. transitive closure operators, relational expressions) to express constraints on the UML class diagram elements.

In this work the transformation rules from class diagram concepts to Alloy have been defined in the QVT language in Chapter 4, but they have been implemented in Java using
the SiTra engine as explained in Section 7.1. The main reason for this is that when we developed our implementation there was no tool available, which supported the QVT language. Since then a number of model transformation frameworks, which support the QVT or a QVT-like language have been made available. Most notably, the Epsilon transformation framework [88], SmartQVT [7] and the Atlas language (ATL) [81] amongst others, provide integrated solutions to a number of issues (such as tracing capabilities for the model transformation), which we had to implement in SiTra from scratch. Evaluating the advantages of those frameworks over SiTra in the context of UML2Alloy and assessing the possibility of using them in future releases of UML2Alloy remains to be investigated.

As discussed in Section 2.1.2 Alloy carries out bounded analysis, based on a user specified scope. Even when Alloy fails to provide a counterexample to an assertion for a given scope, an instance of the model at a higher scope that violates the assertion may exist. Currently there are no clear guidelines on how to chose the ‘right’ scope. The amount of confidence a certain scope inspires is a subjective issue and depends on the problem under analysis. However, the method presented in this thesis, which relies on the analysis capabilities of the Alloy Analyzer and is fully automated, can be used as a first line of defence to discover flaws in the design of a system. If no counterexample is produced by our method, other techniques based on theorem provers can be utilised to ensure a property is not violated. Such techniques are more time consuming and require human intervention and expertise. Using our method before such techniques, will save time and resources by rapidly discovering a number of flaws that would otherwise require more time and resources to uncover.

Moreover, a number of issues remain open for future research. This work does not study the transformation rules for association classes, n-ary and qualified associations. In particular, the UML profile in Appendix E, does not allow expressing a class diagram with association classes or n-ary/qualified associations. To apply the method suggested in this work, such
constructs must be manually refactored to binary associations with OCL to represent the additional semantics as explained in Section 4.1. Transforming association classes, qualified and n-ary associations directly to Alloy remains for future research.

Another interesting direction for further research is to apply emerging model transformation testing techniques [48, 89] on the method presented in this thesis. In particular, it is important to validate that our transformation rules from UML/OCL to Alloy are correct. As a minimum requirement, it is essential to validate that the UML/OCL to Alloy transformation will never generate a syntactically incorrect Alloy model. Applying such model transformation testing techniques, we can test if the model transformation implementation satisfies certain criteria, such as coverage, termination and syntactic correctness [89].

Moreover, as discussed a number of UML and OCL notions are not supported in Alloy and are therefore not supported by our approach. We chose to transform UML/OCL to Alloy, rather than building a UML/OCL Analyzer based on SAT solvers from scratch for a number of reasons. Alloy has already solved the problem of how to represent a high level model into a single boolean formula that can be parsed by SAT solvers. Additionally the Alloy Analyzer works by transforming an Alloy model into the KodKod language [138]. KodKod is an intermediate language between Alloy and SAT-solvers, which comes with a tool that automatically translates a KodKod model into a format that can be used as input in off-the-shelf SAT-solvers, such as Berkmin [64], MiniSat [43] and SAT4J [1]. KodKod provides certain performance improvements. In particular, Torlak and Jackson [138] explain how their tool utilises advanced symmetry detection algorithms and takes advantage of partial instances information in the model to achieve better performance (for detailed benchmarks please refer to [138] and [137]). By transforming UML/OCL into Alloy, which utilises the KodKod engine, our approach merits from all those performance enhancements of the KodKod engine. It is true that this decision to transform class diagrams and OCL to Alloy affects the expressive-
ness of our approach. Exploring the possibilities to alleviate this limitation by transforming UML class diagrams and OCL constraints to KodKod (or directly to SAT solvers, incorporating the performance optimisations of KodKod) remains for future research.

Another issue that needs to be addressed is how UML2Alloy fits into the software development lifecycle. We expect that it can be used to uncover flaws in the early stages of the design of a system. In particular, the simulation capabilities of UML2Alloy make it an appropriate solution for the incremental design of systems. However, case studies with the industry are required to confirm this hypothesis. Moreover, the ability of UML2Alloy to simulate class diagrams with OCL constraints, can be used in the classroom to teach students UML and OCL. We plan to evaluate how suitable the tool is for this task with the help of an empirical study involving students.

The UML defines a number of diagrams, such as statemachines and activity diagrams to express the dynamics of systems. In this work we assume the dynamics of a system are expressed using OCL pre and post conditions. However, the diagrammatic notations the UML offers to model the dynamics are also popular in modelling different views of systems. Transforming such UML diagrammatic notations to Alloy remains for future investigation. Nevertheless, it is expected that petri-nets or CSP algebras are more suitable than Alloy to formalise such notations.

Additionally, since the approach presented in this thesis is based on the analysis capabilities of the Alloy Analyzer, it is natural that our approach is affected by the idiosyncrasies of Alloy. In particular, Alloy’s performance is decreased by the extensive use of integer types in a model (for more details see Section 4.1.6). Additionally Alloy is used for the analysis of software abstractions [77]. Attempting to analyse a model of a system with low level implementation details may be problematic. We are currently in the process of evaluating the performance of the Alloy Analyzer by transforming existing UML models of systems and
developing a number of guidelines for better scalability.

Better scalability and performance can also be achieved by extending and optimising the transformation rules from UML and OCL to Alloy presented in this work. For example, assume the following OCL statement: ‘collection -> size() > 0’. It is expected that if this statement is expressed using the equivalent notEmpty() operation [35] the analysis results will be faster, since the Alloy Analyzer will not have to count all the elements in the collection to evaluate the expression. Instead the Alloy Analyzer will have to check if there is at least one element in the collection. We plan to incorporate such optimisations in the transformation rules, by refactoring expressions in the future.

In [22] and [20] we demonstrated that it is feasible to define both a UML class diagram with OCL constraints and an object diagram on the same Alloy model. It is possible to use this approach to check if a UML object diagram is a valid instance of a model. More precisely, if the object diagram is not a valid instance, a logical inconsistency will exist in the generated Alloy model (for example, if an association end has upper multiplicity of one, but the object diagram has two objects attached to the association end). Therefore, the Unsat Core functionality of the Alloy Analyzer can be used to highlight which parts of the object diagram are not a valid instance of the model. Taking this approach one step further it is possible to check if a model is a valid instance of a metamodel. Defining the transformation rules for object diagram to Alloy remains a subject for future work.

Finally, in this work we presented how to automate the transformation from UML and OCL to Alloy. It would be interesting to investigate the reverse (i.e. the transformation from Alloy to UML and OCL). In particular, certain UML concepts such as association end navigability do not exist in Alloy (i.e. Alloy fields are relations and using the transpose operator they can be navigated in both directions). By defining the transformation from Alloy to UML and OCL we could achieve two goals. First, we would be able to inspect if there are
any bidirectional rules in the transformation from UML and OCL to Alloy. Second, we will be able to represent Alloy models using the more popular UML notation. Representing the textual notation of the signatures and fields of an Alloy model, in the graphical UML class diagram notation aids the modeller to get the “big picture” of the model elements and their relationships. A similar approach is proposed by Kim et al. [86], where the authors present transformation rules from Object-Z to UML class diagrams.
In this Section we present some experimental code for simulating String types with Alloy. The following is a small module that simulates some of the functionality of String types. The purpose of this Section is to demonstrate the feasibility of the approach; we do not intend to provide a library for String types.
Line 3 declares an abstract Signature that represents a character. Lines 4 - 11 declare the
characters in our alphabet (‘a’-‘h’). Lines 13-15 declare the signature String, which has a
value field that denotes a sequence of characters.

Lines 17-18 represent the String “abc”. Lines 20-21 represent the empty string (“”). Lines
24-25 declare a signature Person with a field name. Lines 27-28 specify a constraint that there
is no Person, whose name is empty or “abc”. Lines 31-32 represent the concat method that
concatenates two Strings. Lines 34-39 represent a method that returns the substring of a
String. It accepts as a parameter the index of first and last characters where the concatenation
is going to take place. If the from and to are within the bounds of the String (lines 35-36), the
concatenated String is returned, otherwise the original String is returned (line 39).

Line 42 is used to test the concat method and lines 47-49 are used to test the subString
method and 53-57 are used to test the subString method where the to parameter is greater
than the length of the String.

The first run command (lines 47-49) was used to simulate the concat method with the
SAT4J SAT solver. The result is depicted in Figure A.1. More specifically the value of String
s is ‘ab’ and the value of the String s′ is ‘c’. The value of the concatenated string s″ is as
expected ‘abc’. Simulating the subString method provides similar results.

In this example we also showed how specific Strings (e.g. the String “abc”) can be represen-
ted. We can take advantage of this representation in our translation from UML to Alloy.
We could scan all OCL statements in a model and for every String representation a specific
Alloy Signature will be generated, like the one depicted in lines 17-18. This will make it
possible to translate OCL statements with String types to Alloy. For example assume the
following OCL statement:

context Person
inv: self.name <> "" and self.name <> "abc"

This OCL constraint can be translated to the equivalent Alloy statement depicted in lines 27-28 in our Alloy model.

Remark: This way of representing Strings takes advantage of a latest feature added to the Alloy language, the inherent ability to represent sequences. A sequence in Alloy is a mapping from Integer atoms to the atoms of the type of the sequence. As it can be depicted in Figure A.1 the value of String $s$ is a set of mappings from Integers to Characters. More specifically the value of String $s$, is the set of tuples: $s = \{(0, a), (1, b)\}$

The predefined sequence type in Alloy provides a number of functions to manipulate the sequence (i.e. append or delete items). For more information the interested reader is referred to [6].
APPENDIX B
COMPLETE UML MODEL OF SUDOKU PUZZLE

B.1 UML and OCL Model

This section presents the complete UML model of the sudoku puzzle presented in Section 7.3.1. Figure B.1a shows the puzzle and Figure B.1b shows the class diagram representing it.

![A Sample Sudoku Puzzle](image1)

![A Class Diagram Representing the Sudoku Puzzle](image2)

Figure B.1: A Sample Sudoku Puzzle and its Class Diagram Representation

OCL was used to specify the rules of the puzzle. The following OCL statement specifies the size of the puzzle.

```ocl
class Cell {
  numberOfRow : Integer;
  numberOfColumn : Integer;
}
context Cell
inv puzzleSize : Cell.allInstances() ->forAll (c : Cell | c.numberOfRow > 0 and c.numberOfRow <= 6 and c.numberOfColumn > 0 and c.numberOfColumn <= 6 )
```

The following OCL statement constraints that the values of the cells to be in the range of one to six (since it is a 6x6 puzzle).
The following OCL excerpt illustrates the constraints that define which cells belong to which region (notice the puzzle is split up in six regions as indicated by the bold lines in Figure B.1a.

```ocl
context Cell
inv regions : Cell.allInstances() -> forAll (c:Cell |
  ((c.rowIndex <= 2 and c.columnIndex <=3) implies c.region = 1)
and
  ((c.rowIndex <= 2 and c.columnIndex >3) implies c.region = 2)
and
  ((c.rowIndex > 2 and c.rowIndex <=4 and c.columnIndex <=3)
  implies c.region = 3)
and
  ((c.rowIndex > 2 and c.rowIndex <=4 and c.columnIndex >3)
  implies c.region = 4)
and
  ((c.rowIndex > 4 and c.columnIndex <=3) implies c.region = 5)
and
  ((c.rowIndex > 4 and c.columnIndex > 3) implies c.region = 6)
)
```

The following OCL fragment depicts specifies the existing numbers in the puzzle (for example the cell, whose row is 1 and column is 3, has a value of 6).

```ocl
context Cell
inv numbers_exist : Cell.allInstances() -> forAll (c: Cell |
  ((c.rowIndex = 1 and c.columnIndex = 3) implies c.value = 6)
and
  ((c.rowIndex = 1 and c.columnIndex = 4) implies c.value = 3)
and
  ((c.rowIndex = 1 and c.columnIndex = 6) implies c.value = 2)
and
  ((c.rowIndex = 2 and c.columnIndex = 1) implies c.value = 3)
and
  ((c.rowIndex = 2 and c.columnIndex = 2) implies c.value = 2)
and
  ((c.rowIndex = 2 and c.columnIndex = 6) implies c.value = 6)
and
  ((c.rowIndex = 3 and c.columnIndex = 5) implies c.value = 2)
and
  ((c.rowIndex = 3 and c.columnIndex = 6) implies c.value = 3)
and
  ((c.rowIndex = 4 and c.columnIndex = 1) implies c.value = 2)
and
  ((c.rowIndex = 4 and c.columnIndex = 2) implies c.value = 5)
and
  ((c.rowIndex = 5 and c.columnIndex = 1) implies c.value = 5)
and
  ((c.rowIndex = 5 and c.columnIndex = 5) implies c.value = 3)
and
  ((c.rowIndex = 5 and c.columnIndex = 6) implies c.value = 1)
and
  ((c.rowIndex = 6 and c.columnIndex = 1) implies c.value = 1)
and
  ((c.rowIndex = 6 and c.columnIndex = 3) implies c.value = 2)
and
  ((c.rowIndex = 6 and c.columnIndex = 4) implies c.value = 4))
```

The following OCL statement defines that all cells have different row and column indexes (if two cells have the same row and column index, then they are the same cell).

```ocl
context Cell
inv differentCells : Cell.allInstances() -> forAll (c1, c2 : Cell |
  (c1.rowIndex = c2.rowIndex and c1.columnIndex = c2.columnIndex)
implies c1 = c2)
```
The following OCL fragment illustrates the rules of the puzzle. More specifically if two different cells belong to the same row or column or region, they have to have a different value.

```plaintext
context Cell
inv rules : Cell.allInstances()->forAll(c1,c2:Cell | (c1.rowIndex = c2.rowIndex or c1.columnIndex = c2.columnIndex
or c1.region = c2.region)
and (c1 <> c2 ) implies c1 . value <> c2 . value)
```

### B.2 Generated Alloy Model

The following listing is the Alloy code generated by the application of UML2Alloy on the UML model presented in the previous section.

```plaintext
module EmptyModule
some sig Puzzle{
cells : set Cell}
some sig Cell{
rowIndex : one Int,
columnIndex : one Int,
value : one Int,
region : one Int}
fact { Puzzle <: cells in ( Puzzle) one->some ( Cell)
{ all var : Puzzle | # var . cells = 36 } }
fact { Cell_puzzleSize[ ] }
fact { Cell_values[ ] }
fact { Cell_regions[ ] }
fact { Cell_numbers_exist[ ] }
fact { Cell_differentCells[ ] }
fact { Cell_rules[ ] }

pred Cell_puzzleSize ( ){
all c : Cell | (((int c . rowIndex > 0) &&
(int c . rowIndex <= 6)) &&
(int c . columnIndex > 0)) &&
(int c . columnIndex <= 6))
}

pred Cell_values ( ){
all c : Cell | (int c . value > 0) &&
(int c . value <= 6))
}

pred Cell_regions ( ){
all c : Cell | ((((((int c . rowIndex <= 2) &&
(int c . columnIndex <= 3)) =>
(int c . region = 1)) &&
((int c . rowIndex <= 2) &&
(int c . columnIndex > 3)) =>
(int c . region = 2 ))))
&& (((int c . rowIndex > 2) &&
```
( int c . rowIndex <= 4) &&
( int c . columnIndex <= 3) => ( int c . region = 3 )
}) &&
((
( int c . rowIndex > 2) &&
( int c . columnIndex <= 3) =>
( int c . region = 4)
)) &&
((
( int c . rowIndex > 4) &&
( int c . columnIndex <= 3) =>
( int c . region = 5)
)) &&
((
( int c . rowIndex > 4) &&
( int c . columnIndex > 3)) =>
( int c . region = 6))}

pred Cell_numbers_exist ( ){
all c : Cell | (((((((((((((
( int c . rowIndex = 1) &&
( int c . columnIndex = 3) =>
( int c . value = 6)) &&
((
( int c . rowIndex = 1) &&
( int c . columnIndex = 4) =>
( int c . value = 3)
)) &&
((
( int c . rowIndex = 1) &&
( int c . columnIndex = 6) =>
( int c . value = 2)
)) &&
((
( int c . rowIndex = 2) &&
( int c . columnIndex = 3)) =>
( int c . value = 3)
)) &&
((
( int c . rowIndex = 2) &&
( int c . columnIndex = 3)) =>
( int c . value = 2)
)) &&
((
( int c . rowIndex = 2) &&
( int c . columnIndex = 6)) =>
( int c . value = 6)
)) &&
((
( int c . rowIndex = 3) &&
( int c . columnIndex = 5)) =>
( int c . value = 2)
)) &&
((
( int c . rowIndex = 3) &&
( int c . columnIndex = 6)) =>
( int c . value = 3)
)) &&
((
( int c . rowIndex = 4) &&
( int c . columnIndex = 4)) =>
( int c . value = 3)
)) &&
((
( int c . rowIndex = 4) &&
( int c . columnIndex = 6) =>
( int c . value = 6)
)) &&
((
( int c . rowIndex = 5) &&
( int c . columnIndex = 5)) =>
( int c . value = 6)
)) &&
((
( int c . rowIndex = 5) &&
( int c . columnIndex = 6)) =>
( int c . value = 3)
)) &&
((
( int c . rowIndex = 6) &&
( int c . columnIndex = 3)) =>
( int c . value = 3)
)) &&
((
( int c . rowIndex = 6) &&
( int c . columnIndex = 5)) =>
( int c . value = 6)
))

182
(int c . columnIndex = 1)) =>
(int c . value = 2)
)) &&
((
(int c . rowIndex = 4) &&
(int c . columnIndex = 2)) =>
(int c . value = 5)
)) &&
((
(int c . rowIndex = 5) &&
(int c . columnIndex = 1)) =>
(int c . value = 5)
)) &&
((
(int c . rowIndex = 5) &&
(int c . columnIndex = 5)) =>
(int c . value = 3)
)) &&
((
(int c . rowIndex = 5) &&
(int c . columnIndex = 6)) =>
(int c . value = 1)
)) &&
((
(int c . rowIndex = 6) &&
(int c . columnIndex = 1)) =>
(int c . value = 1)
)) &&
((
(int c . rowIndex = 6) &&
(int c . columnIndex = 3)) =>
(int c . value = 2)
)) &&
((
(int c . rowIndex = 6) &&
(int c . columnIndex = 4)) =>
(int c . value = 4)
))

pred Cell_differentCells ( )
all c1, c2 : Cell | (c1.rowIndex = c2.rowIndex && c1.columnIndex = c2.columnIndex) => (c1 = c2)
}

pred Cell_rules ( )
all c1, c2 : Cell | ((c1.rowIndex = c2.rowIndex) ||
(c1.columnIndex = c2.columnIndex) ||
(c1.region = c2.region)) &&
(c1 != c2) =>
(c1.value != c2.value)
}

pred Puzzle_run ( )
all p : Puzzle | # p . cells > 0
}

pred defaultRunCommand ( )
run Puzzle_run for 4 but 1 Puzzle, 36 Cell, 7 int
run defaultRunCommand for 3
APPENDIX C
COMPLETE UML MODEL OF SECURE SYSTEMS STUDY

C.1 Class Diagram

Figure C.1 depicts the class diagram of the secure system study presented and described in Section 7.3.3.

![Class Diagram of ACTIVE System](image)

Figure C.1: Class Diagram of ACTIVE System

C.2 OCL of the Active Client

```
-- Main Method
context ActiveClient :: main ()
post main : ActiveClient.allInstances ->
forall(ac : ActiveClient | ...
```
ac.at.receiveLoginFromClient()  

-- receiveMsgFromAttacker Method  
context ActiveClient::receiveMsgFromAttacker():Boolean  
post receiveBackLogin_From_Attacker:  
self.encryptedPublicKey = self.at.encryptedPublicKey and  
self.recSessionKey = self.at.recSessionKey and  
self.recINonce = self.at.recINonce and  
self.lmName = self.at.lmName and  
self.checkAC1()  

-- checkAC1 Method  
context ActiveClient :: checkAC1 ( ) : Boolean  
post checkAC1:  
if self.lmName = self.sName then  
  self.checkAC2()  
else  
  self.abortLogin()  
endif  

-- checkAC2() Method  
context ActiveClient :: checkAC2 ( ) : Boolean  
post checkAC2:  
if self.iNonce = self.recINonce then  
  if self.encryptedPublicKey = self.encryptedCPublicKey then  
    self.at.lm.continueWithLogin()  
  else  
    self.abortLogin()  
  endif  
else  
  self.abortLogin()  
endif  

-- AbortLogin Method  
context ActiveClient :: abortLogin ( ) : Boolean  
post abortLogin: self.loginAborted = ResultType::r_True  

-- receiveLoginPage Method  
context ActiveClient::receiveLoginPage (): Boolean  
post receivedLoginPage :  
self.resultPage = ResultPageType::homePage and  
self.loginAborted = ResultType:: r_False  

-- Encrypted Keys are Different Invariant  
context ActiveClient  
inv: ActiveClient.allInstances ->  
forall(ac:ActiveClient |  
ac.at.encryptedAPublicKey <> ac.encryptedCPublicKey)  

C.3 OCL of the Attacker  

-- receiveMsgFromServer Method  
context Attacker::receiveMsgFromServer(): Boolean  
post login_from_server:  
self. lmName = self.lm.lmName and  
self. recSessionKey = self.lm.sessionKey and  
self. recINonce = self.lm.recINonce and  
self. encryptedPublicKey = self.at.encryptedPublicKey and  
self.at.receiveLoginFromClient()
self.encryptedPublicKey = self.lm.encryptedPublicKey and
self.ac1.receiveMsgFromAttacker()
-- receiveLoginFromClient Method

context Attacker::receiveLoginFromClient():Boolean
post receiveLoginFromClient:
self.recINonce = self.ac1.iNonce and
self.recCPublicKey = self.ac1.cPublicKey and
self.recEncryptedCPublicKey = self.ac1.encryptedCPublicKey and
self.lm.receiveLoginFromAttacker()

C.4 OCL of the LoginManager

-- continueWithLogin Method

context LoginManager::continueWithLogin():Boolean
post continueWithLogin:
self.at1.ac1.receiveLoginPage()

C.5 Assertion

The assertion, which we checked the model against is the following:

context ActiveClient
alwaysExits:
ActiveClient.allInstances -> forAll (ac:ActiveClient |
    ac.loginAborted = ResultType::r_True)
Appendix D
Alloy Metamodel

This section presents the MOF representation of the Alloy metamodel we have generated using the method presented in Section 3.2 of this thesis.

D.1 Notation

The notation of the Alloy grammar presented in the following is described in [77, Appendix B.3].

More specifically,

- \( x^* \) represents zero or more repetitions of \( x \).
- \( x^+ \) represents one or more repetitions of \( x \).
- \( x \mid y \) represents a choice between \( x \) and \( y \).
- \([x]\) represents that \( x \) is optional.
- \( x^* \) represents zero or more \textit{comma separated} repetitions of \( x \)
- \( x^+, \) represents one or more \textit{comma separated} repetitions of \( x \)

Terminals appear in \textbf{bold} typeface. Moreover all symbols with name ending in \textit{Id} is a \textit{terminal} and identifier.

D.2 MOF Alloy Metamodel

In this section we present parts of the Alloy grammar (in EBNF) and their equivalent MOF representation. The generated Alloy metamodel is presented in four parts, the \textit{signatures}, \textit{expressions}, \textit{integer expressions} and \textit{constrains} parts. A number of well-formedness rules expressed in OCL are also provided for a number of elements. The rules presented here do not capture all the syntactic constraints of the Alloy language. We plan to extend them in the future, so as to apply model transformation testing techniques as explained in Section 8.2. For more information on the metaelements please refer to Alloy language specification [77, Appendix B.3].
Signatures

Here we describe the part of the Alloy grammar for the signatures and the its MOF compliant representation. Figure D.1 depicts the part of the Alloy grammar for the signatures and the MOF representation we generated from it.

\[
\text{module} ::= \text{header import}^* \text{ paragraph}^* \\
\text{header} ::= \text{module} [\text{path}] \text{ moduleId} / [\text{sigId},^+ ] / \\
\text{paragraph} ::= \text{sigDecl | factDecl | funDecl | predDecl | assertDecl | runCmd | checkCmd} \\
\text{sigDecl} ::= /\text{abstract} /\text{mult} / \text{sig} \text{ sigId},^+ /\text{extends} \text{ sigRef / sigBody} | / \text{mult} / \text{sig} \text{ sigId},^+ \text{ in} \\
\text{sigBody} ::= \{\text{decl},^* \} /\text{constraintSeq} / \\
\text{decl} ::= \text{varId},^+ :\text{declExpr} \\
\text{declExpr} ::= \text{declSetExpr | declRelExpr} \\
\text{declSetExpr} ::= /\text{mult} /\text{expr} \\
\text{declRelExpr} ::= \text{declExp2} /\text{mult} /\text{some} / \text{declExp2} \\
\text{declExp2} ::= \text{declRelExpr} /\text{mult} \\
\text{mult} ::= \text{lone} | \text{one} | \text{some}
\]

Figure D.1: EBNF Representation and MOF Metamodel of the Signatures Part of the Alloy Grammar

Well-formedness constraints in OCL for this part of the Alloy metamodel are the following:

--An element (i.e. signature, field) has to have a name:
D.2.1 Expressions

Figure D.2 depicts the part of the Alloy grammar for the expressions and its MOF representation.

D.2.2 Integer Expressions

Figure D.3 depicts the part of the Alloy grammar for the integer expressions and its MOF representation.

D.2.3 Constraints

Figure D.4 depicts the part of the Alloy grammar for the constraint formulas and its MOF representation.
expr ::= varId | sigRef | unOp expr | expr binOp expr | expr [ expr ] | dec;,* | /constraint/ | let letDecl,+ | expr | Int intExpr | funRef ( expr,* ) | ( expr )

Figure D.2: EBNF Representation and MOF Metamodel of the Expressions Part of the Alloy Grammar
intExpr::=number | # expr | sum expr | int expr | intExpr intOp intExpr | let letDecl,... | intExpr | sum decl,* | intExpr | (intExpr)
intOp::=+ | -

Figure D.3: EBNF Representation and MOF Metamodel of the Integer Expressions Part of the Alloy Grammar
constraintBody ::= constraintSeq | constraint
constraintSeq ::= constraint*
constraint ::= expr / neg / compOp expr | quantifier expr | intExpr / neg / intCompOp intExpr
| neg constraint | constraint logicOp constraint | constraint thenOp constraint / elseOp constraint/
| quantifier decl, + constraintBody | let letDecl, + constraintBody | predRef ( expr, * )
thenOp ::= implies | ⇒
elseOp ::= else |

neg ::= not | !
logicOp ::= and | or | ⇔
quantifier ::= all | no | mult
binOp ::= + | - | & | . | →
unOp ::= ~ | * ^
compOp ::= in | =
APPENDIX E
OVERVIEW OF THE UML METAMODEL OF THE UML PROFILE FOR ALLOY

This chapter presents a detailed description of the UML metamodel used in this work. For every metamodel element, we also describe in detail any additional restrictions imposed by our profile\(^1\).

**Root Diagram of the Kernel Package.**

Figure E.1 depicts the Root diagram of the Kernel package, from [111, p. 23].

*Description of the metaclasses of the Root Diagram:*

**Element**

An Element is an abstract metaclass of the UML metamodel. It is the common metaclass of all UML metamodel elements. An Element can own other elements and a number of comments.

*Additional Constraints Imposed by our Profile*

In our approach we ignore the comments of an Element.

**Relationship**

A Relationship is an Element, which represents the fact that one or more Elements can be related to each other.

**DirectedRelationship**

A DirectedRelationship is a Relationship that relates one or more source Elements to one or more target Elements.

---

\(^1\)This chapter is intended to describe in detail the additional constraints imposed to the UML metamodel elements by our profile. In order to do so we also provide a gentle introduction to the metaclasses of the UML superstructure that our profile is using. For a complete reference and a more in depth description of the metamodel elements the reader is referred to *Part I* of the UML superstructure specification [111].

194
Namespaces Diagram.

Figure E.2 depicts the Namespaces diagram of the Kernel package, from [111, p. 23].

Description of the metaclasses of the Namespaces Diagram:

NamedElement

A NamedElement is an abstract metaclass. It represents an Element, which may have a name. A NamedElement can also have a visibility, which indicates the scope (i.e. public, private, protected or package) of the element. A NamedElement may also have a qualifiedName, which is used to identify the element when there are nested hierarchies.

*Additional Constraints Imposed by our Profile

1. The UML standard allows for a NamedElement to have no name. For example a parameter of an Operation can be unnamed and referenced only by its position [111, Chapter 7.3.41] In our profile the NamedElement has to have a name. This is depicted by the following OCL constraint:

   context NamedElement inv: self.name -> size() = 1

2. The NamedElement also needs to have a qualifiedName. This is depicted by the following OCL constraint:

   context NamedElement inv: self.qualifiedName -> size() = 1
3. The **NamedElement** cannot have a name, which is an Alloy keyword. In OCL:

   ```ocl
class NamedElement {
   inv: Set{'abstract', 'all', 'and', 'as',
   'assert', 'but', 'check', 'disj', 'else',
   'exactly', 'extends', 'fact', 'for', 'fun', 'iden',
   'if', 'iff', 'implies', 'in', 'let', 'lone',
   'module', 'no', 'none', 'not', 'one', 'open', 'or', 'part',
   'pred', 'run', 'set', 'sig', 'some', 'sum', 'then', 'univ'}
   -> excludes(self.name)
   }
   ```

4. The **NamedElement** is one word (i.e. the String does not have any spaces. Underscore and dash can be used, where a space is required.)

5. The **visibility** attribute of the NamedElement is ignored, because there is no equivalent concept in Alloy\(^2\).

\(^2\)Recently the notion of visibility has been added to the Alloy language, so it might be possible to map UML visibility to Alloy. However this remains for future research.
Figure E.3: Multiplicities Diagram of the Kernel Package, from [111, p.24]

**PackageableElement**
A PackageableElement is a NamedElement, which can be owned by a Package.

**Namespace**
Each NamedElement can be owned by zero or one Namespace. A Namespace can own a number of constraints (see Figure E.5). A Namespace can import a number of elements or other packages through ElementImports or PackageImports.

*Additional Constraints Imposed by our Profile*
A Namespace, may not have an elementImport or packageImport. The reason for this is that we have not defined transformation rules for package and element import. It is possible to create modules in Alloy and import elements of a module to another module, however this has various implication as discussed in Section 8.2.

**Multiplicities Diagram.**
Figure E.3 depicts the Multiplicities diagram of the Kernel package, from [111, p. 24].

**Description of the metaclasses of the Multiplicities Diagram:**

**MultiplicityElement**
A MultiplicityElement is an Element, which defines a multiplicity.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isOrdered</td>
<td>Indicates whether the values of the instances of this element are ordered.</td>
</tr>
<tr>
<td>isUnique</td>
<td>Indicates whether the values of the instances of this element are unique.</td>
</tr>
</tbody>
</table>

*Additional Constraints Imposed by our Profile*

1. The value of isOrdered is always false. The reason for this is that we have not defined transformation rules for ordered association ends as discussed in Section 4.1.3.
2. The value of `isUnique` is always true. The reasoning for this restriction is that in Alloy it is not possible to have non-unique association ends (for more information see the discussion on the difference between object ids and atoms in Section 4.1.2).

3. As depicted in Figure E.3 the UML standard allows the `upperValue` and `lowerValue` of an attribute to be any `ValueSpecification`. A `ValueSpecification` can be any OCL expression. The UML standard requires that the OCL expression be constant. Since we have no way of checking whether the OCL expression will denote a constant or not, we constraint ourselves to Integer literals. As a result in our profile, the `upperValue` and `lowerValue` can only be integer literals. This is depicted by the following OCL constraint:

```ocl
class MultiplicityElement

inv: self.upperValue -> forAll(u:ValueSpecification | u.oclIsTypeOf(LiteralInteger) or u.oclIsTypeOf(LiteralUnlimitedNatural))
```

**TypedElement**

A `TypedElement` is a `NamedElement`, which may have a type.

*Additional Constraints Imposed by our Profile*

1. According to the UML specification a `TypedElement` may not have a type. This means that an `Attribute` may be defined without a type. If this occurs the model is probably incomplete and as such it cannot be analysed. As a result our profile requires that the `TypedElement` has a type.

   This is depicted by the following OCL statement:

   ```ocl
class TypedElement

inv: self.type -> notEmpty()
```

**Type**

A `Type` is a `PackageableElement` and is used to constraint the values a `TypedElement` can have.

**Expressions Diagram.**

Figure E.4 depicts the Expressions diagram of the Kernel package, from [111, p. 24].
Description of the metaclasses of the Expressions Diagram:

**ValueSpecification**

A *ValueSpecification* is a *TypedElement* which can also belong to a *Package* (*PackageableElement*).

*Additional Constraints Imposed by our Profile*

1. We only support OpaqueExpressions whose language is OCL. The reason for this is that we have defined transformation rules from OCL to Alloy. We also allow *LiteralSpecifications* whose type is either *LiteralInteger* or *LiteralBoolean* or *LiteralNull*. Our transformation does not currently deal with *String* types, though we provide some guidelines in Appendix A on how String types could be dealt. This constraint expressed in OCL is formulated as:

   ```
   context ValueSpecification
   inv: self.oclIsTypeOf(LiteralInteger) or
   self.oclIsTypeOf(LiteralNull) or
   self.oclIsTypeOf(LiteralBoolean) or
   ((self.oclIsTypeOf(OpaqueExpression) and
     self.language='OCL'))
   ```

**Constrains Diagram.**

Figure E.5 depicts the Expressions diagram of the Kernel package, from [111, p. 25].
Description of the metaclasses of the Constraints Diagram:

Constraint

A Constraint is a PackageableElement. It is defined in a context, which is the NameSpace, under which will be evaluated. A Constraint can reference a number of model elements (constrainedElement). The Constraint also has a specification, which denotes a ValueSpecification. The specification is an expression whose condition has to be true for the Constraint to hold.

*Additional Constraints Imposed by our Profile

1. The UML specification defines that a Constraint may or may not appear in a context. In our work we require a each constraint to be defined in exactly one Namespace. This is depicted by the following OCL constraint:

   context Constraint
   inv: self.namespace -> notEmpty()

2. Our transformation, only supports constraints defined in the context of a Class. Constraints defined in the context of a namespace, other than a Class (such as a Package), may be represented in Alloy, but it is out of the scope of our work.

   context Constraint
   inv: self.context.oclIsTypeOf(Class)

Classifiers Diagram

Figure E.6 depicts the Classifiers diagram of the Kernel package, from [111, p. 26].
Description of the metaclasses of the Classifiers Diagram:

RedefinableElement

A RedefinableElement is an abstract metaclass. It extends the NamedElement metaclass and represents an element that can be redefined through a Generalization hierarchy.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isLeaf</td>
<td>Indicates whether the element can be specialised further by other elements or not. If the value of the attribute is true, this RedefinableElement cannot be specialised further.</td>
</tr>
</tbody>
</table>

A RedefinableElement has a number of redefinedElements. RedefinedElements is the set that is made of the union of the elements that are being redefined by this element.

A RedefinableElement also has a number of redefinitionContexts. A redefinitionContext is the context from which the RedefinableElement can be redefined.

*Additional Constraints Imposed by our Profile

1. Alloy does not support the concept of redefinition for attributes or operations. Consequently, if the RedefinableElement is not a class it cannot be redefined further. More formally:

   ```
   context RedefinableElement
   inv: (not self.oclIsTypeOf(Class)) implies (self.isLeaf = True and self.redefinedElement -> isEmpty() and self.redefinitionContext -> isEmpty())
   ```
**Classifier**

A *Classifier* is an abstract metaclass. It extends the *RedefinableElement*, *Namespace* and *Type* metaclasses. A *Classifier* can be related to a number of *Properties*. Properties denote the attributes of the Classifier. A *Classifier* may be related to a number of *general* classifiers. A *general Classifier* is a *Classifier* related with a *Generalisation* association with another Classifier. A *Classifier* may have a number of *inheritedmembers*. An *inheritedMember* represents the *NamedElements* inherited from the *general* Classifiers. A *Classifier* may also be related to a number of *Generalizations*.

<table>
<thead>
<tr>
<th>Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>isAbstract</em></td>
</tr>
</tbody>
</table>

*Additional Constraints Imposed by our Profile*

1. A specific Classifier may only be related to *one general*. This constraint prohibits multiple inheritance. The reason for this is explained in Section 4.1.2. Formally this constraint is represented by the following OCL constraint:

   ```
   context Classifier
   inv: self.general -> size() <= 1
   ```

**Generalization**

A *Generalization* denotes a *DirectedRelationship* that is used to specify a taxonomy between a general and a specific *Classifier*.

**Features Diagram.**

Figure E.7 depicts the Features diagram of the Kernel package, from [111, p. 27].

**Description of the metaclasses of the Features Diagram:**

**Feature**

A *Feature* is an abstract metaclass. It denotes a *RedefinableElement* that represents *Structural* or *Behavioural* features of a *Classifier*.

<table>
<thead>
<tr>
<th>Attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>isStatic</em></td>
</tr>
</tbody>
</table>

202
**StructuralFeature**

A *StructuralFeature* is an abstract metaclass. It has a type and a *Multiplicity* (as it extends both the *TypedElement* and *Multiplicity* classes). It denotes the structural elements of a *Classifier*.

**Attributes:**

- **isReadOnly** Indicates whether the feature can be modified or not.

**BehavioralFeature**

A *BehavioralFeature* is an abstract metaclass. It denotes a *Feature* of the *Classifier* that represents some kind of behaviour. A *BehavioralFeature* can own some *Parameters*. A *BehavioralFeature* can also declare a number of *raisedException* that can be used indicate the exceptions that can be raised when the *BehavioralFeature* is executed.

*Additional Constraints Imposed by our Profile*

1. The *raisedExceptions* of a *BehavioralFeature* are ignored. Indeed a raisedException is usually used for low level modelling of the system (i.e. code generation). When dealing with the abstract specification of a system, it is usually not necessary to deal with the exceptions that a *BehavioralFeature* may raise. Consequently a *BehavioralFeature* may not be related to any *raisedExceptions*. More formally:

\[\text{RaisedExceptions of a BehavioralFeature are ignored.} \]

---

\[^3\text{It is possible to explicitly model a raisedException on the model level, but this is out of the scope of this}\]
context BehavioralFeature
inv: self.raisedException -> isEmpty()

Parameter
A Parameter is a TypedElement and a MultiplicityElement. As a result the parameter of a BehavioralFeature has a type and can be multivalued.

Attributes:

<table>
<thead>
<tr>
<th>direction</th>
<th>Indicates the direction of the parameter (in, inout, out, return). An in parameter is only used as input in the BehavioralFeature. An inout parameter is used as input and may be modified by the BehavioralFeature. An out Parameter is used to pass values out of the BehavioralFeature. Finally a return Parameter indicates that the value is returned from the BehavioralFeature. Indicates the default value of the Parameter in the BehavioralFeature, if a value has not been assigned to the Parameter by the client when calling the feature.</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td></td>
</tr>
</tbody>
</table>

*Additional Constraints Imposed by our Profile*

1. The direction of a Parameter can only be inout or return. This is expressed by the following OCL constraint:

```plaintext
context BehavioralFeature
inv: self.ownedParameter.direction = ParameterDirectionKind::inout or self.ownedParameter.direction = ParameterDirectionKind::return
```

Operations Diagram.

Figure E.8 depicts the Operations diagram of the Kernel package, from [111, p. 28].

*Description of the metaclasses of the Operations Diagram:*

An Operation is a concrete metaclass that represents a BehavioralFeature that may own a number of Constraints, and may have a Type and may redefine one or more Operations. The Operation may have a number of Parameters, like any other BehavioralFeature (as explained in page 203).
An Operation may also have a number of Precondition, Postcondition or bodyCondition Constraints. Preconditions are Constraints that have to be satisfied before the invocation of the Operation. Similarly Postconditions are Constraints that need to be satisfied after the Operation has completed. BodyConditions are Constraints on the result values of Operations. Bodyconditions can be seen as a similar concept as PostConditions, but bodyConditions may be overridden by other Operations, while Postconditions may only be amended [111, Section 7.3.36].

An Operation may also have a type, which indicates the type of the return parameter if it exists.

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isQuery</td>
<td>Indicates whether the Operation does not change the system state. If it is true the Operation does not change the system state. If it is false the state might change.</td>
</tr>
<tr>
<td>isOrdered</td>
<td>Indicates whether the return Parameter of the Operation isOrdered.</td>
</tr>
<tr>
<td>isUnique</td>
<td>Indicates whether the return Parameter is unique or not.</td>
</tr>
<tr>
<td>lower</td>
<td>Represents the lower Multiplicity of the return Parameter.</td>
</tr>
<tr>
<td>upper</td>
<td>Represents the upper Multiplicity of the return Parameter.</td>
</tr>
</tbody>
</table>

*Additional Constraints Imposed by our Profile*

1. The bodyCondition is ignored by our transformation, since we do not allow overriding of Operations.

2. As with BehavioralFeatures we do not support raisedExceptions of Operations.
3. For reasons explained in Section 4.1.3, we do not support overriding of Operations. Consequently we cannot support redefinedOperations. The is expressed by the following OCL invariant:

```ocl
class Operation {
  inv: self.redefinedOperation -> size() = 0
}
```

4. The value of isQuery is ignored. IsQuery is usually used to indicate Operations that query an attribute from the UML model and perform a calculation based on the value of the attribute queried. Our translation is not affected by whether the Operation is a query Operation or not.

Classes Diagram.

Figure E.9 depicts the Classes diagram of the Kernel package, from [111, p. 29]. The Classes diagram defines the main metamodel elements used for modelling Class Diagrams.

**Description of the metaclasses of the Classes Diagram:**

**Class**

A Class is a concrete metaclass, which is used on the model level to represent the well-known concept of a class. The metaclass Class extends the abstract metaclass Classifier. A Class has a number of ownedAttributes. Each ownedAttribute denotes the Attributes and navigable Association Ends of the Class.
A Class may also be related to a number of `superClasses`. A superClass denotes a parent child association between two classes, through a `Generalization` relation.

A Class may also be related to a number of `nestedClassifiers`. A nestedClassifier is a Classifier defined within the Class. This provides the ability to model inner Classes.

A Class may also own a number of `ownedOperations`. Each ownedOperation denotes an Operation of the Class.

*Additional Constraints Imposed by our Profile*

1. We do not allow for a Class to be related with `nestedClassifiers`. The reason for this is that there is no direct support for inner classes in Alloy. This constraint is formally defined in OCL:

   context Class
   inv: self.nestedClassifier -> size() = 0

2. Abstract classes, which are not extended by other classes, are not treated by Alloy abstract, as explained in page 52. The following OCL constraint depicts this rule:

   context Class
   inv: self.isAbstract implies
   (Class.allInstances() -> exists(c:Class | c.superClass -> includes(self)))

**Property**

A `Property` is a concrete metaclass that is an extension of a `StructuralFeature`. It can represent either an `Attribute` or a navigable `Association End` of a class.

A `Property` can be related to a number of `subSettedProperties`. A `subSettedProperty` represents a `Property` of which this `Property` is a subset.

A `Property` may also be related to a number of `redefinedProperties`. A `redefinedProperty` represents a `Property` which is redefined by this `Property`.

A `Property` may have an opposite `Property`. This is a derived attribute. If this `Property` is a navigable association end and belongs to a binary association and the other association end is navigable, `opposite` gives the opposite association end.

A `defaultValue` denotes a `ValueSpecification`, which gives the default value that the Property will have when the Class to which the Property belong, will be instantiated.

A `Property` may also be related to an `Association`. If it is related to an `Association`, the `Property` denotes an `Association End`.  

207
A Property may also be related to an owningAssociation, in which case the Property
denotes a non-navigable Association End.

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isDerived</td>
<td>Indicates whether the value of the Property can be computed from the values of other elements of the model.</td>
</tr>
<tr>
<td>isReadOnly</td>
<td>Indicates whether the Property value can be changed or not.</td>
</tr>
<tr>
<td>isDerivedUnion</td>
<td>Indicates whether the value of the Property can be computed as a union of all the Properties that subset the Property.</td>
</tr>
<tr>
<td>default</td>
<td>Represents the default value that the Property will have when the Class to which the Property belongs is instantiated. This is calculated from the defaultValue ValueSpecification related to the Property.</td>
</tr>
<tr>
<td>aggregation</td>
<td>Represents the Aggregation type of the Property (i.e. if it is a normal Property, an aggregation or a composition).</td>
</tr>
<tr>
<td>isComposite</td>
<td>This attribute is derived from the value of the aggregation attribute and denotes whether the Property is an aggregation or a composition.</td>
</tr>
</tbody>
</table>

*Additional Constraints Imposed by our Profile*

1. If an attribute is derived OCL needs to be used to express how the value of the attribute is computed.

**Association**

An Association is an element, which extends the Relationship and the Classifier metaclasses. An Association has two or more memberEnds. Each memberEnd denotes an association end. An Association may also have a number of ownedEnds. These represent the association ends owned by the Association. An Association may also have a navigableOwnedEnds, which denote association ends, which are navigable. Finally an Association may have or more endTypes. The endTypes of an Association is a collection of the types of its the association ends.

*Additional Constraints Imposed by our Profile*

1. Our transformation only supports binary associations. Higher order associations needs to be refactored to binary. More formally:

```oclm
context Association
inv: self.memberEnd -> size() = 2
```
AggregationKind

AggregationKind specifies the kind of Aggregation of a property. This can be either none, shared or composite.

*Additional Constraints Imposed by our Profile*

1. Our transformation does not support aggregation or composition. More formally: context Association inv: self.aggregation = AggregationKind::none and self.isComposite = False

DataTypes Diagram.

Figure E.10 depicts the DataTypes diagram of the Kernel package, from [111, p. 30].

Description of the metaclasses of the Datatypes Diagram:

**DataType**

A **DataType** is a special kind of **Classifier**. It may have a number of **ownedAttributes** and **ownedOperations**. The main difference between a **DataType** and a **Class**, is that a **DataType** is identified only by its values, while a Class is identified by the **Class** identifier [111, Section 7.3.11].

**PrimitiveType**

A **PrimitiveType** denotes the predefined UML primitive types. Those are Boolean, Integers, UnlimitedNatural and String [111, Chapter 7.3.43].
**Additional Constraints Imposed by our Profile**

1. Alloy does not have any inherent notion of Primitive types, apart from Integers. Moreover it does not provide full support on Integers. For example, an operation on integer numbers is multiplication. Alloy does not support such operations. It only supports addition and substraction. As a result our profile only supports integer numbers and addition and substraction. For a more detailed discussion on how our transformation deals with integer numbers, please refer to Section 4.1.6.

2. For the reason discussed above (i.e. Alloy has inherent support only for integer types), the only Primitive types allowed in our profile Integers.

**Enumeration**

An *Enumeration* is a *DataType*, which consists of a number of Enumeration literals (*ownedLiteral*).

**EnumerationLiteral**

An *EnumerationLiteral* is an *InstanceSpecification* that represents a user defined value.

**Packages Diagram.**

Figure E.11 depicts the Packages diagram of the kernel package, from [111, p. 31].

---

4The latest version of the Alloy language has added support for multiplication and division between integers, but providing the transformation rules for such operations remains for future work.
Description of the metaclasses of the Packages Diagram:

Package

A Package is a Namespace and a PackageableElement, which provides the ability to group a number of elements. It can own a number of ownedMembers, which are other PackageableElements. A Package may be related to a number of ownedType, which are the Types owned by the Package (for example user defined datatypes). A Package may also own a number of packageMerges. Finally, a Package may own a number of other Packages (called nestedPackages).

*Additional Constraints Imposed by our Profile

1. The notion of PackageMerge is not formally defined by the UML standard. Zito et al. [151] demonstrate that the concept of PackageMerge as defined by the UML standard is not commutative (i.e. the result of the merge depends on the order of the packages being merged). In order to avoid such complications we do not support PackageMerge in our transformation. This constraint is depicted by the following OCL statement:

   context Package
   inv: self.packageMerge -> size() = 0

   Defining transformation rules for PackageMerge remains to be investigated in the future.

2. Package nesting is also not supported. The following OCL statement depicts this constraint.

   context Package
   inv: self.nestedPackage -> size() = 0

3. In our profile a Package can only own a Class, an Association, a Datatype, a Generalization or a GeneralizationSet. We do not allow other types of PackageableElements (i.e. a Package) because this would generate a Package hierarchy. Our transformation rules do not deal with package hierarchy, because this may provide additional complications as discussed in Section 8.2. As a result the only PackageableElements that a Package may own is Classes, Associations, Generalizations, GeneralizationSets or DataTypes:

   context Package
inv: self.ownedMember -> forAll(om:OwnedMember | om.oclIsKindOf(Class) or om.oclIsKindOf(Association) or om.oclIsKindOf(Generalization) or om.oclIsKindOf(GeneralizationSet) or om.oclIsKindOf(DataType) )

**PackageMerge**  A PackageMerge is a DirectedRelationship, which is used to denote that a Package is merged by another Package. As noted previously (see description of the Package concept in page 211), we do not support PackageMerge.

**AssociationClasses Package.**

Figure E.12 depicts the AssociationClasses diagram of the AssociationClasses package, from [111, p. 34].

**Description of the metaclasses of the AssociationClasses Diagram:**

**AssociationClass**

An AssociationClass is an element that is both a Class and an Association. An AssociationClass is used to connect a number of Classifiers, but also to declare a set of attributes that belong to the association itself and not to any of the connected Classifiers.

**Additional Constraints Imposed by our Profile**

1. As discussed in Section 4.1.8, our profile requires that all AssociationClasses are refactored to binary associations. More formally:

   ```
   context AssociationClass
   inv: AssociationClass.allInstances()
       -> size() = 0
   ```
Property
The AssociationClasses package defines that a property can be related to zero or more other properties that denote qualifiers [111, Section 7.3.44].

*Additional Constraints Imposed by our Profile

1. As discussed in Section 4.1.9, the UML profile for Alloy requires that qualified properties (i.e. association ends) need to be expressed as normal binary associations. More formally:

   context AssociationClasses::Property
   inv: self.qualifier -> size = 0

PowerTypes Package.
Figure E.13 depicts the PowerTypes diagram of the PowerTypes package, from [111, p. 34].

Description of the metaclasses of the PowerTypes Diagram:
GeneralizationSet
A GeneralizationSet expresses how a Generalization relation between two Classifiers, partitions the instances of the General Classifier, in the instances of the specific Classifier.
Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isCovering</td>
<td>Indicates whether the specific Classifiers partition totally the instances of the general Classifier(s).</td>
</tr>
<tr>
<td>isDisjoint</td>
<td>Indicates whether the specific Classifiers do not share any instances in common.</td>
</tr>
</tbody>
</table>
1. The UML metamodel allows a Generalization to belong to more than one GeneralizationSet. Our profile specifies that a Generalization can belong to at most one GeneralizationSet:

\[
\text{context Generalization} \\
\text{inv: self.generalizationSet -> size() =< 1}
\]

For more information on the GeneralizationSet, please see the relevant section of the transformation rules (Section 4.1.10).
APPENDIX F
ALLOY MODEL OF THE UML CLASS DIAGRAM OF SECTION 4.2

This section presents the complete Alloy textual code automatically generated from the example UML model of Figure 4.17, using the transformation rules presented in Section 4.1. Some comments have been manually inserted to illustrate, which part of the model came from which part of the UML class diagram.

```alloy
// Module declaration
// Corresponds to the Example
// Package
module Example

// Corresponds to the Element class
abstract sig Element{
  dr1 : one DirectedRelationShip,
  relationship : lone RelationShip,
  dr2 : one DirectedRelationShip}

// Corresponds to the NamedElement class
sig NamedElement extends Element{
  name : one String,
  visibility : one VisibilityKind}

// Corresponds to the String datatype
sig String{}

// Corresponds to the VisibilityKind enumeration
abstract sig VisibilityKind{

  // Correspond to the VisibilityKind
  // enumeration literals
  one sig PRIVATE extends VisibilityKind{}
  one sig PUBLIC extends VisibilityKind{}

  // Corresponds to the RelationShip class
  // Note the 'some' keyword that
  // enforces at least one instance of this signature
  // to exist in the simulation
  some sig RelationShip extends Element{
    relatedElement : some Element}
```

215
// Corresponds to DirectedRelationShip class
sig DirectedRelationShip extends RelationShip{
  numOfElements : one Int,
  source : some Element,
  target : some Element}

// Bidirectionality fact for the first association
// between the DirectedRelationShip and the
// Element
fact dr1_source { dr1 = ~source }

// Multiplicity fact for the first association
// between the DirectedRelationShip and the
// Element
fact { Element <: dr1 in { Element} set->one { DirectedRelationShip}
  DirectedRelationShip <: source in { DirectedRelationShip} one->set { Element} }

// Bidirectionality fact for the association between the
// RelationShip and the Element
fact relationship_relatedElement { relationship = ~relatedElement }

// Multiplicity fact for the first association
// between the RelationShip and the Element
fact { Element <: relationship in { Element} set->lone { RelationShip}
  RelationShip <: relatedElement in { RelationShip} lone->set { Element} }

// Bidirectionality fact for the second association
// between the DirectedRelationShip and the
// Element
fact dr2_target { dr2 = ~target }

// Multiplicity fact for the second association
// between the RelationShip and the Element
fact { Element <: dr2 in { Element} set->one { DirectedRelationShip}
  DirectedRelationShip <: target in { DirectedRelationShip} one->set { Element} }

// Union fact for the relatedElement
// association end
fact(relatedElement = source+target)

// Default empty simulation command
// since no OCL statement with the
// 'simulation' stereotype has been specified
pred defaultRunCommand { }{}

// The scope for the run command. Follows
// the defaultScope, intScope and scopedElement
// tagged values specified in the model

run defaultRunCommand for 6 but 5 int, 1 RelationShip
APPENDIX G
OCL AND ALLOY OPERATIONS PRECEDENCE

This section present the operation precedence rules for both OCL and Alloy, starting from the lowest precedence to the highest.

G.1 OCL Precedence Rules
The OCL precedence rules are shown in the following list [108, Sec. 7.4.7].

- “implies”
- “and”, “or”
- “=”, “<>”
- “<”, “>”, “<=”, “>=”
- “if-then-else-endif”
- “+”, “-”
- “@”’, “？”
- “not” and unary minus
- dot and arrow operations (“.”) and “->”
- @pre
G.2 Alloy Precedence Rules

The Alloy precedence rules are shown in the following list [77, Sec. B.4]:

- “||”, “or”
- “=>”, “implies”
- “&&”, “and”
- “!”, “not”
- “+”, “-”
- “#X”
- “&”
- “->”
- “.” (dot join)
- “~”, “*”, “^”
This section presents a complete example of how to chain operations in OCL and how to represent them in Alloy, using the method presented in Section 6.3.

H.1 Example

Figure H.1 depicts the UML class diagram of the example (inspired from Gogolla et al. [63]), while Figure H.2 shows the OCL specification of the class diagram.

Using the algorithm presented in Section 6.3, we generate the following Alloy model.

```alloy
module civilstatusworld
open util/ordering[Time] as ord

sig Time{}
...

sig Person{
  marstatus: Status one -> Time,
  gender: one Gender,
  husband: Person lone -> Time,
  wife: Person lone -> Time,
  prevSpouse: Person set -> Time}

fact{all t:Time | wife.t = ~(husband.t)}

fact{all t:Time | wife.t in Person lone -> lone Person}

fact{all t:Time | husband.t in Person lone -> lone Person}

fact{all t:Time | prevSpouse.t in Person lone -> set Person}

pred marry(spouse:Person, self:Person,t:Time,t’:Time)
  // PRECONDITIONS
  self.marstatus.t != MARRIED and self.gender != spouse.gender and
  spouse ! in self.prevSpouse.t and
  #self.husband.t = 0 and #self.wife.t=0
  #spouse.husband.t =0 and #spouse.wife.t=0

  //POSTCONDITIONS
  self.marstatus.t’ = MARRIED and
  self.prevSpouse.t’ = self.prevSpouse.t + spouse and
  spouse.prevSpouse.t’ = spouse.prevSpouse.t + self and
  (self.gender = FEMALE) =>
    (self.husband.t’ = spouse and self.husband.t’.marstatus.t’ = MARRIED)
  else
    (self.wife.t’ = spouse and self.wife.t’.marstatus.t’ = MARRIED))
```
Figure H.1: Example

```
and divorce{self,t',ord/next[t']}

pred divorce(self:Person,t,t':Time){
// PRECONDITIONS
self.marstatus.t = MARRIED
//POSTCONDITIONS
self.marstatus.t' = DIVORCED
((self.gender = FEMALE => (#self.husband.t' = 0 and
self.husband.t.prevSpouse.t' = self.husband.t.prevSpouse.t
and self.husband.t.marstatus.t' = DIVORCED))
((self.gender = MALE implies (#self.wife.t' = 0 and
self.wife.t.prevSpouse.t' = self.husband.t.prevSpouse.t
and self.wife.t.marstatus.t' = DIVORCED))))
pred init(self:Person,t,t':Time){ some p:Person | {
p!= self and marry[p,self,t,t']
and #self.prevSpouse.t = 0})
pred simulate{
let t=ord/first[] |
{some p:Person | init[p,t,ord/next[t]]}
run { simulate[] } for 3 but 4 Time
fact{all self:Person,t:Time | genderok[self,t]}
pred genderok[self:Person,t:Time]{
(self.gender = FEMALE => (#self.wife=0)) &&
(self.gender = MALE implies (#self.husband=0))
context Person
inv genderok:
(self.gender = Gender::FEMALE implies self.wife->size() = 0) and
(self.gender = Gender::MALE implies self.husband->size() = 0)
```

Please note that in the postcondition of the `marry` operation the `divorce` operation is referenced. This has the following effect: while the preconditions and postconditions of the `marry` operation are evaluated on time $t$ and $t'$ respectively, the preconditions and postconditions of the `divorce` operation are evaluated on time $t'$ and $t''$ respectively.
H.2 Analysis

Simulating the example, using the Alloy Analyzer produces the three instances shown in Figures H.3. In particular, in time $t$ there are two divorced persons, $\text{Person0}$ and $\text{Person1}$. In time $t'$, $\text{Person0}$ and $\text{Person1}$ are married to each other. In particular, $\text{Person1}$ is the wife of $\text{Person0}$ and $\text{Person0}$ the husband of $\text{Person1}$. Moreover $\text{Person1}$ has been added to the collection of the previous spouses ($\text{prevSpouse}$) of $\text{Person0}$ and $\text{Person0}$ has been added to the collection of the previous spouses of $\text{Person1}$. Finally, as shown by Figure H.3c in time $t''$, both $\text{Person0}$ and $\text{Person1}$ are divorced again and also remain in the collection $\text{prevSpouse}$ of the other person.
context Person::marry(spouse:person):Boolean
pre marry_pre: self.marstatus <> Status::MARRIED and
self.gender<> spouse.gender and
self.husband -> size() = 0 and
self.wife -> size() = 0 and
spouse.wife -> size() = 0 and
spouse.husband -> size() = 0 and
self.prevSpouse -> excludes(spouse)
post marry_post:
self.marstatus = Status::MARRIED and
self.prevSpouse = self.prevSpouse@pre -> including(spouse) and
spouse.prevSpouse = spouse.prevSpouse@pre -> including(self) and
(if self.gender = Gender::Female then
  (self.husband = spouse and self.husband.marstatus = Status::MARRIED)
else
  (self.wife = spouse and self.wife.marstatus = Status::MARRIED)
and self.devorce() end)

context Person::divorce:Boolean
pre divorce_pre: self.marstatus = Status::MARRIED
post divorce_post:
self.marstatus = Status::DIVORCED and
self.prevSpouse = self@pre.prevSpouse@pre and
(if self.gender = Gender::FEMALE implies
  ((self.husband -> size() = 0) and
  (self@pre.husband@pre.prevSpouse = self@pre.husband@pre.prevSpouse@pre)
and (self@pre.husband@pre.marstatus = Status::DIVORCED)))
and (self.gender = Gender::MALE implies ((self.wife -> size() = 0) and
  (self@pre.wife@pre.marstatus = Status::DIVORCED) and
  (self@pre.wife@pre.prevSpouse = self@pre.wife@pre.prevSpouse@pre))
context Person::init():Boolean
pre: Person.allInstances() -> exists(p:Person |
p <> self and self.prevSpouse -> size() = 0 and
self.marry(p))
context Person
-- This OCL statement is stereotyped as "simulation"
inv simulate: Person.allInstances() -> exists(p:Person |
p.init())
context Person
-- The following is an invariant
inv genderok:
  (self.gender = Gender::FEMALE implies self.wife->size() = 0) and
  (self.gender = Gender::MALE implies self.husband->size() = 0)
Figure H.3: Random Instance of the marry and divorce Operations
List of References


233


