Pós-Graduação em Ciência da Computação

TARCIANA DIAS DA SILVA

VALIDATING TRANSFORMATIONS OF OO PROGRAMS USING THE ALLOY ANALYZER

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife
2017

Tarciana Dias da Silva

Validating Transformations of OO Programs using the Alloy Analyzer

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

ADVISOR: **Augusto Cezar Alves Sampaio**
CO-ADVISOR: **Alexandre Cabral Mota**

Recife
2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S586v Silva, Tarciana Dias da
Validating transformations of OO programs using the alloy analyzer / Tarciana Dias da Silva. – 2017.
149 f.: il., fig.

Orientador: Augusto Cezar Alves Sampaio.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências e apêndice.

1. Engenharia de software. 2. Linguagem de programação. I. Sampaio, Augusto Cezar Alves (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2018-104

Tarciana Dias da Silva

Validating Transformations of OO Programs using the Alloy Analyzer

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutora em Ciência da Computação

Aprovado em: 23/08/2017

_____

Orientador: Prof. Dr. Augusto Cezar Alves Sampaio

BANCA EXAMINADORA

_____

Prof. Dr. Márcio Lopes Cornélio
Centro de Informática / UFPE

_____

Prof. Dr. Juliano Manabu Iyoda
Centro de Informática / UFPE

_____

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática / UFPE

_____

Prof. Dr. Rohit Gheyi
Departamento de Sistemas e Computação / UFCG

_____

Profa. Dra Ana Cristina Vieira de Melo
Instituto de Matemática e Estatística / USP

ACKNOWLEDGEMENTS

Primeiramente, gostaria de agradecer a Deus pela vida, pela saúde, pela família e filho maravilhosos que me deu, e por ter estado sempre ao meu lado, guiando os meus caminhos e decisões. À minha mãe, Fernanda Maria, um agradecimento muitíssimo especial, por ter sido a pessoa que mais me incentivou e esteve ao meu lado, tendo uma contribuição fundamental para a conclusão desta tese. Ao meu pai, Luiz da Penha, pelos exemplos de pessoa, profissional e persistência dados ao longo de uma vida que, certamente, foram fundamentais para que eu chegasse até aqui. Ao meu marido, Bruno Gomes, por todo o carinho e apoio. Aos meus irmãos, Taíse e Luiz Fernando, por toda a ajuda, apoio e companheirismo de sempre.

Aos meus orientadores Augusto e Alexandre, por todas as reuniões, sempre regadas de incentivo, boas risadas e bom–humor. Muito obrigada por todos os ensinamentos, apoio, paciência e atenção durante todo este período. Vocês me fizeram conhecer na prática o verdadeiro significado da palavra orientador, tendo o feito com extrema maestria. Espero sempre ter a sorte de poder trabalhar com profissionais como vocês.

A Giovanny Palma, pelas discussões construtivas, pela contribuição fundamental, pelo tempo disponibilizado inclusive aos finais de semana. Muito obrigada, Giovanny, você realmente foi mais uma pessoa iluminada que cruzou o meu caminho. Aos demais amigos do formula, especialmente Sidney Nogueira, Gustavo Carvalho e Márcio Cornélio pelas valiosas discussões sobre este trabalho.

Aos colegas da Universidade Federal de Campina Grande, UFCG, em especial Rohit, Gustavo e Melina, por terem me recebido e acolhido tão bem em reuniões presenciais e pelos debates ricos em torno de Java, refactorings e principalmente, Alloy.

Aos amigos Cristina Luzia, Raphael D'Castro, Juliana Neiva, Lucas Freire, Amanda Pimentel, Suely Batista, por todo o incentivo e ajuda para a conclusão deste trabalho.

ABSTRACT

Program transformation is current practice in software development, especially refactoring. However, in general, there is no precise specification of these transformations and, when it exists, it is neither proved sound nor validated systematically, which can cause static semantic or behavioural errors in the resulting programs, after each transformation application. This work proposes a strategy to validate program transformation specifications grounded by a formal infrastructure built predominantly in Alloy. In this work we focus on transformations in languages that support OO features such as Java, ROOL, the calculus of refinement of component and object-oriented systems known as rCOS and an OO language with reference semantic. The scope of this work, concerning the strategy implementation, is a subset of Java. Complementarily to testing, formal languages provide a mathematically solid reasoning mechanism to establish the soundness of transformations. Unfortunately, a complete formal treatment for transformations in OO languages is ruled out because even for Java there is no complete formal semantics. We investigate the trustworthiness of program transformations by using an approach that combines (bounded) model finding and testing. Our Alloy formal infrastructure comprises four main Alloy models: (1) a metamodel for a subset of OO language features and a set of predicates that capture the static semantics, where the main predicate is able to determine if a program is well–formed; (2) a Transformation-Specific model for each program transformation being investigated; (3) a Static Semantics Validator model; and (4) a Dynamic Validator Model, which generates all possible instances (according to the scope specified), each one having a representation of a program before and after the transformation application. If any instances are generated in (3), this means that there is a failure in the transformation specification. So, in this case it is necessary to correct the specification and re–submit it to the Alloy Analyzer. This process is repeated until no more instances are found by the Static Semantics Validator Model. Hence, the instances generated by the Dynamic Validator model only represent well–formed programs since it is only applied *after* the Static Semantics Validator model. Afterwards, the instances generated by (4) are further submitted to a tool, called Alloy-To-Java Translator, which generates Java programs corresponding to these instances along with tests to be applied in each side of the transformation. These programs are finally validated with regard to dynamic semantic problems, based on these automatic tests generated in (4). In this way, a developer can implement the transformations with some confidence on their behavioural preservation, after validating the transformation specifications using the proposed framework. The strategy we propose seems promising since it is an alternative to validate transformations even when a complete semantics of the languages is not available. The results of the validation of a representative set of transformations found in the literature show that some transformation issues, concerning both static and dynamic behaviour, can be detected.

Keywords: Program Transformation Specification. OO. Alloy. Alloy Analyzer. Validation.

RESUMO

Transformação de programas é uma prática atual em desenvolvimento de software, especialmente *refactoring*. No entanto, em geral, não há uma especificação precisa dessas transformações e, quando existe, não é provada correta nem sistematicamente validada, o que pode causar erros de semântica estática ou comportamentais nos programas resultantes da transformação. Este trabalho propõe uma estratégia, baseada em uma infraestrutura predominantemente formal construída em Alloy, para validar especificações de transformação de programas. Neste trabalho, focamos em transformações em linguagens que suportam características orientadas a objetos (OO) tais como Java, ROOL, o cálculo de refinamento de componentes e sistemas orientados a objetos, conhecido como rCOS e uma linguagem OO com semântica de referência. O escopo deste trabalho, com relação à implementação da estratégia, é um subconjunto de Java. Complementarmente a testes, linguagens formais fornecem um mecanismo matematicamente sólido para estabelecer a consistência (*soundness*) das transformaçõoes. Infelizmente, um tratamento formal completo para transformações em linguagens OO é descartado porque, mesmo para Java, não há uma semântica formal completa. Nós investigamos a corretude de transformações de programas usando uma abordagem que combina *(bounded) model finding* e testes. Nossa infraestrutura formal é composta de quatro modelos Alloy principais: (1) um metamodelo para um subconjunto de características de linguagens OO e um conjunto de predicados que capturam a semântica estática correspondente, onde o predicado principal é capaz de determinar se um programa é bem–formado; (2) um modelo específico para cada transformação (que está sendo investigada); (3) um Validador de Semântica Estática e (4) um Validador Dinâmico, que gera instâncias possíveis da transformação (de acordo com o escopo especificado), cada uma tendo uma representação de um programa antes e depois da transformação. Se alguma instância for gerada em (3), isto significa que há uma falha (de semântica estática) na especificação da transformação. Neste caso, é necessário corrigir a especificação e re–submetê–la para o *Alloy Analyzer*. Este procedimento é repetido até nenhuma instância ser encontrada pelo modelo do Validador de Semântica Estática. Logo, as instâncias geradas pelo modelo do Validador Dinâmico (4) tipicamente somente representam programas bem formados já que este é aplicado na nossa estratégia apenas depois que o modelo em (3) não retornar instância alguma. Em seguida, as instâncias geradas em (4) são submetidas a uma ferramenta, *Alloy–To–Java Translator*, que transforma as instâncias geradas em programas Java, e também gera os testes que serão aplicados em cada lado da transformação. Estes programas são finalmente validados com relação a problemas dinâmicos com base nestes testes gerados em (4) de forma automática. Dessa forma, um desenvolvedor pode implementar as transformações com alguma segurança depois de validar a especificação das transformações usando o framework proposto. A estratégia que propomos parece promissora já que é uma alternativa para validar especificações de transformações em geral mesmo quando uma semântica completa da linguagem não está disponível. Resultados da validação de um conjunto representativo de especificações de transformações, encontrados na literatura, mostram

que tanto problemas de semântica estática quanto dinâmica podem ser detectados.

Palavras-chave: Especificações de Transformações de Programas. OO. Alloy. Alloy Analyzer. Validação.

LIST OF FIGURES

Figure 1 – Overview of our strategy. 16

Figure 2 – Rule 3 (Move Method) in (QUAN; ZONGYAN; LIU, 2008). 26
Figure 3 – Representation of Rule 3 (Move Method) in UML (QUAN; ZONGYAN; LIU, 2008). 27
Figure 4 – Rule 6 (Pull Up Method) in (QUAN; ZONGYAN; LIU, 2008). 27
Figure 5 – Representation of Rule 6 (Pull Up Method) in UML (QUAN; ZONGYAN; LIU, 2008). 27
Figure 6 – A UML class metamodel and its representation in Alloy 29
Figure 7 – Traditional Object Management Group modeling infrastructure (ATKINSON; Kï¿½HNE, 2003). 33

Figure 8 – Overview of our strategy. 36
Figure 9 – OO Metamodel embedded in Alloy 38
Figure 10 – Subpart of the JLS syntactic grammar. 40
Figure 11 – FormalParameterList element and its dismemberment. 41
Figure 12 – MethodBody element and its dismemberments. 42
Figure 13 – Statement element and its dismemberments. 43
Figure 14 – ExpressionStatement metamodel. 44
Figure 15 – Dismemberments of the StatementExpression. 45
Figure 16 – Primary element and its dismemberments. 45
Figure 17 – ClassInstanceCreationExpression element and its dismemberments. 46
Figure 18 – An overview of the main predicates in our OO Model 46
Figure 19 – Law 1 with the indication of the predicates expanded in Code 3.22 corresponding to provisos, from the right– to the left–hand side direction. 60
Figure 20 – Law 2 with the indication of the main elements and the predicates (in Code 3.28) corresponding to provisos, from the right to the left–hand side direction. 63
Figure 21 – Call graph of the main predicates and functions used in the predicate rule3LR. 79

Figure 22 – Overview of the first part of our complete strategy. 96
Figure 23 – Second part of our Validation Strategy. 98
Figure 24 – An example of program generated with behavioural problems when Law 2 is applied. 104
Figure 25 – Classes generated according to the Law 4 specification. 107
Figure 26 – Example of a program generated with behavioural problem using Rule 2.1 specification. 109

Figure 27 – Example of programs where Rule 3 was applied. 117
Figure 28 – Classes generated showing anomalies in the program transformation. 122
Figure 29 – Example of the Java program and its corresponding program graph (OVERBEY; JOHNSON, 2011). 126

LIST OF TABLES

Table 1 – Classification of main specifications analysed according to each context 101
Table 2 – Comparison of main specifications analysed according to the different errors found 103
Table 3 – Comparison of the main works presented 134

CONTENTS

1 INTRODUCTION 13
1.1 Our approach . . . 15
1.2 Main Contributions . . . 17
1.3 Thesis Structure . . . 18

2 BACKGROUND 19
2.1 Algebraic Notations . . . 19
2.1.1 Equivalence Notion . . . 21
2.1.2 Algebraic Laws . . . 22
2.1.3 Refactoring Rules . . . 24
2.1.4 rCOS . . . 26
2.2 The Alloy language and the Alloy Analyzer . . . 27
2.2.1 The Alloy language . . . 28
2.2.2 The Alloy Analyzer . . . 31
2.3 Metamodels for Java . . . 32

3 JAVA AND TRANSFORMATION–SPECIFIC MODELS 35
3.1 The OO Metamodel . . . 36
3.1.1 Correspondence between the Java Language Specification and our OO metamodel 39
3.1.2 Predicates in our OO metamodel – specifying well–formedness rules . . . 44
3.2 Transformation–Specific Models . . . 55
3.2.1 Transformation–Specific Model for Law 1 . . . 57
3.2.2 Transformation–Specific Model for Law 2 . . . 62
3.2.3 Transformation–Specific Model for Push Down Refactoring . . . 67
3.2.4 Transformation–Specific Model for Pull Up/Push Down Method Rule in (CORNELIO, 2004) . . . 70
3.2.5 Transformation–Specific Model for Rule 3 . . . 75
3.2.6 Transformation–Specific Model for Rule 6 . . . 91

4 VALIDATING TRANSFORMATION SPECIFICATIONS 95
4.1 Static Semantics Validation Step . . . 95
4.2 Dynamic Validation Step . . . 97
4.3 The Alloy–To–Java Translator . . . 99
4.4 Evaluation . . . 100
4.4.1 Experiment definition . . . 101
4.4.2 Planning . . . 101
4.4.2.1 Selection of Subjects . . . 101

4.4.2.2 Instrumentation *102*
4.4.3 Results 102
4.4.3.1 Analysis for Law 2 . . . . . *102*
4.4.3.2 Analysis for Push Down Method Refactoring *106*
4.4.3.3 Analysis for Pull Up/Push Down Method Rule in (CORNELIO, 2004) *107*
4.4.3.4 Analysis for Rule 3 in (QUAN; ZONGYAN; LIU, 2008) *116*
4.4.3.5 Analysis for Rule 6 in (QUAN; ZONGYAN; LIU, 2008) *118*
4.4.3.6 Other Analyses *121*
4.4.4 Threats and Validity 122
4.4.4.1 Construct Validity *122*
4.4.4.2 Internal Validity *122*

5 RELATED WORK 124

6 CONCLUSION AND FUTURE WORK 135
6.1 Future Work **138**

REFERENCES 140

APPENDIX A - OO METAMODEL 143

1

INTRODUCTION

Program transformation is current practice in software development, especially refactoring. In literature, refactorings is known as a program transformation that preserves behaviour. It is used for many purposes, such as to improve program readability, reduce coupling, and introduce concurrency in sequential programs. The objective of refactorings is the perfective evolution of the models to improve quality aspects without changing the observable behaviour of the system (DIAZ V., 2014). Yet, according to (DIAZ V., 2014), beyond the perfective model evolution, there are also the corrective (concerned with correcting errors in the design) and adaptive one (concerned with modifying a design model to accommodate changes in requirements and design constraints).

Usually, refactorings are available in IDEs, like Eclipse or NetBeans. Unfortunately, as commonly occurs in software engineer, people worry about in providing an implementation without a correspondent specification. Furthermore, specifying a transformation is a very challenging task. This kind of specification must consider a program as input to the transformation be applied and this program can assume different patterns and contain different source language elements. The specification should also take into consideration both static and dynamic semantics of the source language for which the transformation is specified and the necessary conditions that a starting–hand side program should fit to the transformation be applied. The tasking of specifying transformation is even more challenging if we consider that in most cases there is no formal specification of both static and dynamic semantics of the source language in question. Thus, in general, a more abstract specification of the transformation[1] itself is not available, and when it exists, it is neither proved sound nor even systematically validated. In some cases, only one or two examples are given to explain the transformation and there is no precise description of the transformation specification, which makes difficult to validate it. As a consequence, the implementation correspondent to this specification usually presents faults.

Most works focus on providing transformation (more commonly, refactorings) implementations, and are usually available as plug–ins to an IDE tool. In these works, the transformations are typically validated using test suites, provided by the tool itself or by some IDE. Due to the

[1] In this work, we will refer as *transformation* every change to a program, regardless if it is through a refactoring, an algebraic law, refactoring or rCOS rules, and so forth. Each one of these concepts will be further detailed in this thesis.

absence of transformation precise specifications, some works such as (OVERBEY; JOHNSON, 2011; OVERBEY M. J. FOTZLER; JOHNSON, 2011; SCHAFER, 2010) take the initiative to also specify these refactorings in a way to ease implementation. They provide a high-level specification of common refactorings, but in terms of pseudocode, in order to facilitate implementations. The authors also compare their refactoring engine with the Eclipse one, using the Eclipse internal test suite. They also give a formal correctness proof for one of their defined and implemented refactoring, as a case study.

On the other hand, the works in (SOARES, 2015; SABINO, 2016) focus mainly on evaluation of the implementations provided by (OVERBEY; JOHNSON, 2011; OVERBEY M. J. FOTZLER; JOHNSON, 2011; SCHAFER, 2010); but the initiatives in (SOARES, 2015; SABINO, 2016) do not address specification or implementation of refactorings. They present a technique to test refactoring engines based on test input generators using the Alloy Analyzer; in their cases, Alloy instances characterize Java programs that are used as inputs to be submitted to the refactoring engines implementations. The authors use Alloy for the generation of random instances (supported by a metamodel for a subset of Java), and translate them to Java. In (SOARES, 2015), after verifying that a generated instance is compilable, they apply a refactoring (available in some refactoring engine) on such an instance and a new program is obtained. These programs (before and after a refactoring) are subjected to a test campaign, where behavioural changes, as well as compilation errors caused by the application of the refactoring, are evaluated.

However, a validation of transformation specifications, regardless of their implementations in a source language, is not addressed. As discussed along this thesis, our strategy focused on transformation *specification* validations, considering that a transformation specification is given as input. Using Alloy and the Alloy Analyzer, we simulate all possible inputs (according to a given scope of elements) matching a specification template which enables a transformation to be applied. At the same time, all the respective instances of programs are also generated concomitantly by our Alloy infrastructure and it is checked if the transformation does not cause static semantics or behaviour (in case of transformations following a perfective or corrective model evolution) problems in them. So our main goal is transformation validation, rather than providing an transformation implementation that can be plugged into an IDE and used by developers.

Validating transformation specification is a challenge. Complementarily to testing, formal languages provide a mathematically solid reasoning mechanism to establish the soundness of transformations. There are some efforts in this direction. In (BORBA *et al.*, 2004), a set of algebraic laws is proposed for a subset of Java with copy semantics (ROOL). Soundness is proved based on a formal semantics. The algebraic approach has been adopted to provide insight into the algebraic semantics of several programming languages, proved to be useful as a basis for the definition of trustworthy transformations (DUARTE; MOTA; SAMPAIO, 2011; DUARTE, 2008). In (QUAN; ZONGYAN; LIU, 2008), it is investigated how design patterns and refactoring rules are used in a formal method by formulating and showing them as refinement laws in the calculus of refinement of component and object–oriented systems, known as rCOS. In addition, Fowler's refactoring

rules, described via examples in (FOWLER., 2002), are formulated as rCOS refinement rules for their correctness to be provable, as claimed by the authors. In (SILVA; SAMPAIO; LIU, 2008) a set of behaviour–preserving transformation laws for a sequential object-oriented language is proposed with reference semantics, in the context of the rCOS calculus. The work in (PALMA, 2015) enhances the one in (SILVA; SAMPAIO; LIU, 2008) and proposes a more comprehensive (and relatively complete) set of algebraic laws and a data refinement rule for reasoning about object oriented programs with a reference semantics. The work described in (DUARTE; MOTA; SAMPAIO, 2011; DUARTE, 2008) proposes laws in the Java context but neglects soundness proofs; the central barrier is the lack of a complete formal semantics for Java.

The specifications we validated follow an algebraic presentation style. Some refers to algebraic laws, others to refactoring rules and some others to refinement laws. Although algebraic or refinement laws can not only represent or characterize perfective evolution models, the specific ones validated in this work do. They were validated in various contexts: the one where specifications were only (1) postulated, regardless any kind of validation; the other one where they were (2) proved; and, finally, the one they were derived from provably correct ones—we validated 4 specifications in (1) context, 1 in (2) context and 2 in (3) context. In all of them, different kind of errors were found, mainly, the static semantics and behavioural ones. The former was found in 4 of the 7 specifications analysed whilst the latter, in 5 of the 7 specifications analysed.

1.1 Our approach

In this work, we propose a strategy to validate specifications of program transformations. Figure 1 shows an overview of our strategy. Firstly, we capture all the transformation elements, defined in the transformation specification, into a Transformation–Specific Alloy Model, which is built from a generic OO Alloy model that represents a subset of the Java Language. The specification elements include classes, methods and/or fields involved in the transformation, along with the provisos or premises as well as substitution (if they exist or are necessary). We use several kinds of specifications which are detailed in Chapter 2.

Secondly, another Alloy Model called the Static Semantics Validator is executed by the Alloy Analyzer. This Validator is used to check if the transformation (described in the previous Alloy model), applied to a well–formed starting program (or simply SS, for short), causes static semantics problems in the resulting program (or simply RS, for short). Basically, static semantics problems mean compilation errors. If it is possible to find ill–formedness in the resulting program, the Static Semantics Validator presents Alloy instances corresponding to each situation. This validator can be used in transformations that fit in all kind of evolution models (perfective, corrective or adaptive). We rely on the bounded exhaustive analysis (according to a given scope) (JACKSON, 2006) (see Chapter 2) provided by the Alloy Analyzer to capture all possible variations of SS well–formed programs, together with the application of the transformation for each one.

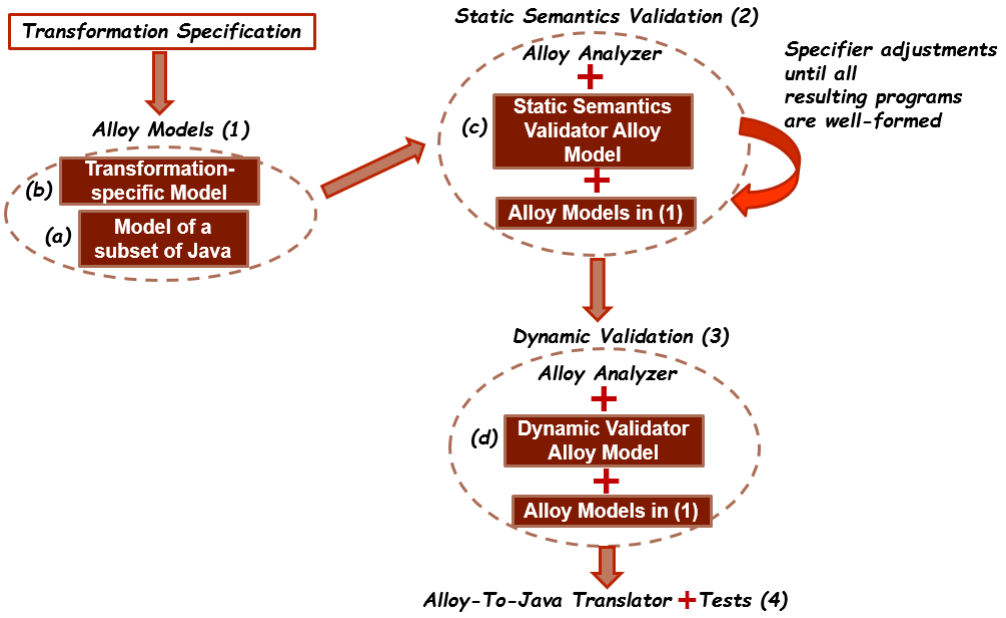


Figure 1 Overview of our strategy.

This transformation is done through a main predicate defined in the Transformation–Specific Alloy Model in (1). Thus, if the Static Semantics Validator Model generates any Alloy instance, it comprises the following pair of programs: a starting well–formed program and its corresponding *non* well–formed resulting program. The specifier can adjust the transformation specification or the Alloy Models in (1) until no more pairs of programs with a non well–formed RS are found by the Static Semantics Validator.

Thirdly, a Dynamic Validation phase starts. In this phase, the Alloy Analyzer only generates, through the Dynamic Validator Model, well–formed pairs of programs (or else, programs that do not present static semantics problems) as long as the transformation–specific model is carefully adjusted in the previous step. Both Static Semantics and Dynamic validation steps are detailed in Chapter 4.

Finally, all the instances (representing pairs of programs, before and after the application of the transformation being analysed) generated by the Dynamic Validator, are translated into Java and a test campaign, generated in Java in step (4), is performed to detect possible dynamic problems. This step can be discarded when transformations considered in the context of adaptive evolution models since as new features are added, it is difficult guaranteeing the same system behaviour.

The method we propose to validate transformations can be used in a complementary way to test tools and oracles. Our focus is on improving transformation specifications in the sense that they can be validated in a constructive way. Some experiments we have conducted have produced evidence that transformation failures can be detected during the specification analysis, without the need to implement them in a source language or submitting them to a more elaborated test campaign. Instead we use the Alloy Analyzer, with adequate models, and simple validation tests.

Despite our OO model built is based on the Java Language Specification (ORACLE, 2016), as detailed in Chapter 3, our OO Model is only a subset of Java which groups common OO constructs of various other OO languages. Because of this, it was possible to validate transformations not only specified for Java programs but for other languages as well. The program specifications we validated were written for languages such as Java (presented in (DUARTE; MOTA; SAMPAIO, 2011; DUARTE, 2008)), rCOS (shown in (QUAN; ZONGYAN; LIU, 2008)), ROOL (in (CORNELIO, 2004)), and an object–oriented language with reference semantics (presented in (PALMA, 2015)). All of them have in common OO features that are compatible with the ones supported by our OO metamodel. Thus, our strategy can detect errors in specifications related to OO features that are language independent. All of these specifications are mainly represented by either refactoring, or refinement rules or algebraic laws, detailed in Chapter 2, which are the input to our strategy.

We adopt a test–based approach with random method invocation, and with structural comparison of the results in SS and RS programs. Some experiments have uncovered several behavioural problems in the analysed transformation, as detailed in Chapter 4. As future work we consider an alternative approach that assumes a formal (behavioural) semantics for an OO language like Java in Alloy, and the semantic validation performed by the Alloy Analyser, but such a semantic model is not yet available.

1.2 Main Contributions

In summary, the main contributions of this work are as follows:

- A transformation validation engine, composed by:
  - an OO metamodel: a metamodel in Alloy that supports some of the main of OO features. This metamodel together with a transformation–specific model is able to validate transformations in many OO languages that have in common OO features such as transformations defined in Java ((DUARTE, 2008)), rCOS (QUAN; ZONGYAN; LIU, 2008), ROOL (CORNELIO, 2004), and some OO languages with reference semantics (PALMA, 2015). The predicates defined in our OO metamodel detect if a specific program is well–formed with regard to the absence of static semantics problems, considering the subset defined in this model;
  - a model in Alloy representing each transformation. This model gives support to the Validator Models (item below) and allows the generation of both sides of a transformation (in the Alloy abstract syntax notation or Alloy instances format) and can be seen as a precise transformation specification;
  - two main Alloy Validator Models: the Static Semantics and the Dynamic one;

- the Alloy-To-Java Translator, which translates the Alloy abstract syntax notation to the Java one. Hence, it generates the Java programs corresponding to the object instances generated by our Alloy formal transformation engine. The Alloy-To-Java Translator also generates test classes;
- a Java Validator tool, that receives the Java programs translated by the Alloy-To-Java Translator (after the Static Semantics or Dynamic Validator is executed), and compile them, giving the corresponding compiler error, if it exists (in the case of programs from instances generated by our Static Semantics Validator). If it is not the case, in the case of programs from instances generated by our Dynamic Validator, it executes test classes (generated earlier by the Alloy–To–Java translator) to check for dynamic problems that cannot be identified by the Alloy Analyzer but are identified in our strategy by testing.

1.3 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 provides some background. Specifically, an overview of the algebraic style for presenting the transformation laws is given (see Section 2.1.2) as well as some other formalisms used to specify transformations such as rCOS and refactoring rules; in addition, the Alloy language and the Alloy Analyzer are introduced as well as how their use contributed to our solution (see Section 2.2). Some challenges in dealing with this formal infrastructure are also pointed out.

Chapter 3 describes the Alloy models: the OO metamodel and the various transformation–specific Alloy models (for each transformation specification). The corresponding elements and predicates are detailed and, for the OO metamodel, a comparison with the elements of the Java Language Specification is done.

Chapter 4 describes our Alloy Validators and how they can detect transformation specification errors which would cause compilation or behavioural problems. In addition, the Alloy-To-Java Translator and the generation of test classes are detailed. Besides, validation results are discussed in various transformation specifications. Our strategy is also compared to existing ones in Chapter 5. Finally, Chapter 6 summarises the main contributions and presents future work.

2

BACKGROUND

In this chapter, some concepts on which our infrastructure is based on are introduced. At first, it is a challenge to express correct transformations for complex languages, mainly the ones without a formal language documented. There are some formalisms and notations used to specify transformations. The algebraic–like styles of presentation help in this direction by allowing transformations to be specified in a compositional way. The specifications following this style can be used as a basis for proving the soundness of the transformations when a formal semantics is available. Many works such as (BORBA *et al.*, 2004; SILVA; SAMPAIO; LIU, 2008; NAUMANN; SAMPAIO; SILVA, 2012; CORNELIO, 2004; QUAN; ZONGYAN; LIU, 2008; PALMA, 2015; DUARTE, 2008) illustrate the applicability of transformation specifications in algebraic–style notations. We detailed some of them in Section 2.1.

Secondly, as these kind of transformation specification establishes the equivalence between programs before and after the transformation according to the conditions stated for the transformation, we also establish the notion of equivalence in Section 2.1.1 used in our strategy with regard to the fourth step (see Figure 1). Actually, this equivalence can only be applied to a transformation that is not a refinement since a refinement does not guarantee the behaviour preservation. As a consequence, the fourth step of our strategy can not be applied to a refinement as well.

In addition, the Alloy formal language as well as the Alloy Analyzer are detailed in Section 2.2 since they are used to build our formal Alloy infrastructure which implements and validates program transformations. Finally, an overview of the use of metamodels and Model–Driven Architecture, as well as metamodels used to represent the Java language, is also given in Section 2.3.

2.1 Algebraic Notations

Some works propose constructions in the algebraic style. In (BORBA *et al.*, 2004), a set of sound and (relatively) complete laws, along with a strategy, is proposed for reducing programs to an imperative normal form. Besides, they clarify aspects of a (copy) semantics of object–oriented constructs and the major application of their laws is to formally derive more elaborate behaviour preserving program transformations, useful for optimizing or restructuring object–oriented

applications. Also, they present how laws can be used as a basis for proving refactorings. Later, the works (SILVA; SAMPAIO; LIU, 2008) and (NAUMANN; SAMPAIO; SILVA, 2012) presented laws for object–oriented languages considering a reference semantics. In (SILVA; SAMPAIO; LIU, 2008) the authors use the rCOS semantics to prove soundness of each proposed law, illustrating the applicability through a case study for improving code structure; in (NAUMANN; SAMPAIO; SILVA, 2012) the focus is on data refining class hierarchies. A common feature of all these works is that they are based on extremely simplified languages, when compared to languages like Java. The advantage is that these languages have a formal semantics and allow one to prove the soundness of the transformations.

Additionally, there is a very poor documentation regarding the most common refactoring specifications. The absence of precise refactoring descriptions is also mentioned in (SCHAFER, 2010). Even the existing refactorings in modern available IDEs are only explained in terms of one or two examples, without any guarantee of static semantics preservation of the programs, let alone their behaviour.

Some works, nonetheless, take the initiative to specify refactorings. In (SCHAFER, 2010), the authors define informal pseudocode notation for specifying refactorings that serve as the basis of an implementation. They decompose complicated refactorings into microrefactorings to make the description of the former ones more manageable. The implementation is then verified by using both correctness proofs (SCHAFER, 2010; SCHAFER; EKMAN; MOOR, 2009) and their own test suite and the one for Eclipse and IntelliJ, which are publicly available. In (BECKER *et al.*, 2011), a refactoring specification consists of a set of rules, formalized by graphs. They use an invariant checker to check that the refactoring rules do not produce any forbidden patterns.

In (OVERBEY; JOHNSON, 2011), the traditional precondition-based approach is followed. A library containing preconditions and ways to check them, for the most common refactorings, is available in different languages. In all cases, the way in which the refactoring is specified remains very close to how it is implemented, which makes it difficult to understand and is language dependent. On the other hand, an algebraic approach (represented by algebraic laws as well as refactoring or refinement rules) helps in this direction by allowing transformations to be specified in a compositional way. They provide precise, easy to understand and implementation independent refactorings.

The work in (CORNELIO, 2004) takes the initiative to formalise refactorings for a language named ROOL (an acronym for Refinement Object–oriented Language) using an algebraic approach based on what they call refactoring rules. The main objective of this work is to formalise and prove refactorings. A set of rules is given to formalize adaptations of an important set of refactorings catalogued in (FOWLER., 2002) and a few others formulated by (OPDYKE, 1992). However, ROOL takes a copy semantics defined by weakest preconditions. Without references, some important and interesting laws of OO programs do not hold for ROOL.

In (QUAN; ZONGYAN; LIU, 2008), it is investigated how design patterns and refactoring rules are used in a formal method by formulating and showing them as refinement rules in

the refinement calculus of component and object–oriented systems, known as *rCOS*. The advantage of using *rCOS* is that it takes a reference semantic model with rich OO features, including subtypes, visibility, inheritance, dynamic binding and polymorphism. In particular, some Fowler's refactoring rules are formulated as *rCOS* so their correctness can be provable. However, they were not and there is no relative completeness for the laws. They show the formulation for one rule from each of six Fowler's categories as a representative, while the others are left in their report (LONG Q., 2005). In (SILVA; SAMPAIO; LIU, 2008) a set of behaviour-preserving transformation laws for a sequential object-oriented language is proposed with reference semantics (rCOS). The work in (PALMA, 2015) enhances the one in (SILVA; SAMPAIO; LIU, 2008), and proposes a comprehensive set of algebraic laws (in the sense they can reduce a Java like program to an imperative normal form) for reasoning about object oriented programs with a reference semantics.

Inspired by the laws of ROOL (BORBA *et al.*, 2004), the work described in (DUARTE, 2008) presented laws for a significant subset of Java (ORACLE, 2016), considering Java's concrete syntax and the provisos for laws. New laws were introduced: laws involving constructors, static methods, and abstract classes (which were not considered before), as well as some parallelization algebraic laws to introduce concurrency in an original sequential program. Aiming at covering a broader range of applications, a strategy to cope with open systems was presented and incorporated to the laws. The authors of (DUARTE, 2008) also presented a reduction strategy to transform Java programs into a normal form, expressed in a small subset of Java. This allowed them to establish relative completeness of the set of laws. Their main goal is showing that there is a better performance when their parallelization strategy is applied to a program. And they show this using some case studies. In addition, a comparison between the (outcome provided by them) transformed program and the original program was done to see that the application of their strategy did not cause behavioural problems in the programs. Unfortunately, they do not provide any additional form of validation for the laws. Using our strategy, we detect some problems in their laws (the ones that reduce a program into a normal form), as will be discussed in Section 4.4.3.1, although these laws have been derived from already proved ones in (BORBA *et al.*, 2004).

2.1.1 Equivalence Notion

The term equivalence notion refers whether two programs have the same behavior (SABINO, 2016). In (BORBA *et al.*, 2004), that is, the programs (before and after the transformation) are compared with regard to the main function, similarly as done by (OPDYKE, 1992). In our strategy, our main method only invokes non–void methods, so the results of these methods are compared through a sysout command. In other words, we hope that the transformed program have the same output set of the original program for a given set of inputs. This is similar to the seventh property of the set of properties defined by Opdyke (OPDYKE, 1992) for refactorings that ensure the correctness of the transformations. The first six properties are related to well–

formedness of the programs, whereas the last property is related to semantics preservation of the program. Opdyke (OPDYKE, 1992) defines semantic equivalence between programs as: ”let the external interface of the program be the main function. If the main function is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same (p. 40)”. In this way, a refactoring may change the internal structure of the program since the mapping between input and outputs of the main method be preserved. Although for some application domain, guaranteeing that for a set of inputs the program has the same outputs after the transformation is not enough to state the transformation preserved behaviour (MENS; TOURWÉ, 2004), we used the same equivalence notion in (OPDYKE, 1992) in our work.

2.1.2 Algebraic Laws

Algebraic laws have been used to provide a formal basis for transformations, including refactorings. In the context of the refinement calculus for imperative programming, there are well established laws that can assist and form a basis for formal program development (MORGAN, 1994; PALMA, 2015). Likewise are the laws for imperative programming (AL., 1987). In (BORBA; SAMPAIO., 2000), a set of basic formal programming laws for object–oriented programming in the ROOL context are presented. These laws deal with imperative commands of ROOL as well as with medium–grain object–oriented constructs (PALMA, 2015). Borba et al.(BORBA *et al.*, 2004; BORBA; SAMPAIO; CORNELIO, 2003) present a comprehensive set of laws for object–oriented programming. They concentrate on object–oriented features, and they show that this set of laws is sufficient to transform an arbitrary program into a normal form expressed in terms of a small subset of the language operators. These laws not only clarify aspects of semantics, but also serve as a basis for deriving more elaborate laws and for practical applications of program transformations (PALMA, 2015). These laws are also used to prove rules that support compiler construction in the algebraic style proposed by Sampaio (SAMPAIO, 1997; PALMA, 2015).

Typically, in the object–oriented paradigm, an algebraic law establishes the equivalence between two programs according to a program context represented by the entire set of class declarations (*cds*), a Main class (*Main*), and also considering that some provisos must be respected. The algebraic approach has been adopted to provide insight into the algebraic semantics of several programming languages, proved to be useful as a basis for the definition of trustworthy transformations. It has been shown that algebraic laws can be successfully used in the context of program transformations (CORNELIO, 2004; GHEYI; MASSONI, 2005), as they can be composed to prove more complex transformations.

In the case of a specification written in an algebraic style, program transformations are described as conditional equations. Each equation (or algebraic law) is intended to express a semantic equivalence between the starting– and the resulting–hand side programs (SS and RS, respectively) which denote the sides before and after the transformation application, respectively. In the presentation style used in (BORBA *et al.*, 2004) and (DUARTE, 2008), conditions marked

with $(\leftrightarrow)$ must hold when performing the transformation in both directions; conditions marked with $(\rightarrow)$ must hold for the transformation from left to right, and those with $(\leftarrow)$ from right to left. In addition, *ads*, *cnds*, and *mds* represent the attribute, constructor, and method declarations, respectively; T represents an attribute type; and the symbol $\leq$ represents the subtype relation between classes.

As an example, the algebraic transformation described in Law 1 captures a refactoring that moves an attribute to a superclass and also the inverse transformation (from right to left). This inverse transformation is used as one of our running examples in this work (see Chapter 3), where the attribute of class B is moved to class C. The proviso $(\leftarrow)$ of this law states that the attribute can be moved provided it does not already belong to the set of attributes of the class C (1). Besides, there must be no access to this attribute by any subclass of B, excluding the subclasses of C (2). The proviso $(\rightarrow)$ of this law is simpler: the attribute can be pulled up to class B provided there must be no declaration of this field in its subclasses. The constraints established by this algebraic law are reflected in the transformation–specific model (see Chapter 3).

Law 1. ⟨move attribute to superclass (DUARTE, 2008)⟩

class B **extends** A {		**class** B **extends** A {
ads		**public** T f; ads
$cnds$		$cnds$
mds		mds
}		}
class C **extends** B {	$=_{cds,Main}$	**class** C **extends** B {
public T f; ads'		ads'
$cnds'$		$cnds'$
mds'		mds'
}		}

provided

(→) *The attribute name f is not declared by the subclasses of B in cds;*

(←) *(1) The attribute name f is not declared in ads'; (2) D.f, for any $D \leq B$ and $D \not\leq C$, does not appear in cds, Main, cnds, cnds', mds, or mds'*

Another example is Law 2, which captures moving an original method to its superclass and the inverse transformation. This is similar to the *push down method* refactoring, mentioned in many works such as (SCHAFER, 2010; SOARES, 2015). The proviso $(\leftrightarrow)$ states that the method can be pulled up or pushed down provided there is no access to *super* or *private* attributes in its body (1). In addition the method is not declared in any subclasses of B (2) and is not *private* (3). The proviso $(\rightarrow)$ requires that there is no other method with the same signature and arguments in class B (right) (1) and the body of the method being pulled up do not contain any uncast occurrences of the keyword *this* or expressions in the form *((C)this).a*, for any protected attribute a in the set of attributes of class C(2). Finally, proviso $(\leftarrow)$ requires there is no method declaration in C (left) and (2) is similar to the condition ($\leftarrow$ (2)) of Law 1.

As already mentioned, transformations in this style are the main input for our validation strategy. Our Alloy infrastructure was used to check if the provisos in the mentioned laws were correct (thus not causing static semantics or behavioural problems). The corresponding Alloy models for Laws 1 and 2 are shown in Sections 3.2.1 and 3.2.2, respectively, whilst the one representing the Push Down/Pull Up Method, mentioned in works such as (SCHAFER, 2010; SOARES, 2015), in Section 3.2.3. The conclusions about the provisos of these transformations are described in Sections 4.4.3.1 and 4.4.3.2, respectively.

Law 2. ⟨move original method to superclass (DUARTE, 2008)⟩

class B **extends** A $\{$ ads $cnds$ mds $\}$ **class** C **extends** B $\{$ ads' $cnds'$ $rt\ \ m(pds)\ \ \{mbody\}$ mds' $\}$	$=_{cds,Main}$	**class** B **extends** A $\{$ ads $cnds$ $rt\ \ m(pds)\ \ \{mbody\}$ mds $\}$ **class** C **extends** B $\{$ ads' $cnds'$ mds' $\}$

provided

- $(\leftrightarrow)$ *(1) super and private attributes do not appear in mbody; (2)* $m(pds)$ *is not declared in any subclass of B in cds; (3)* $m(pds)$ *is not private.*
- $(\rightarrow)$ *(1)* $m(pds)$ *is not declared in mds; (2) mbody does not contain uncast occurrences of* **this** *nor expressions in the form* $((C)\mathbf{this}).a$ *for any protected attribute a in* ads'.
- $(\leftarrow)$ *(1)* $m(pds)$ *is not declared in* mds'*; (2)* $D.m(e)$*, for any* $D \leq B$ *and* $D \not\leq C$*, does not appear in cds*, *Main, mds or* mds'

2.1.3 Refactoring Rules

Refactoring rules, illustrated in (CORNELIO, 2004), also follow an algebraic approach and are very similar to the ones explained earlier in Section 2.1.2. Refactoring Rules are described by means of two boxes written side by side, along with *where* and *provided* clauses, where the former is used to write abbreviations and the latter lists the provisos for applying a refactoring rule. The left–hand side of the rule presents the class or classes before the rule application; the right–hand side presents the classes after the rule application: the transformed classes. In addition, many of the refactoring rules are equalities and can be applied in both directions (CORNELIO, 2004).

Rule 2.1 ⟨*Pull Up/Push Down Method*⟩—Rule 4.4 in (CORNELIO, 2004)

```
1  class A extends D
2    ads_a;
3    mts_a
4  end
5  class B extends A
6    ads_b;
7    meth m ≙ (pds_m • c')
8    mts_b
9  end
10 class C extends A
11   ads_c;
12   meth m ≙ (pds_m • c')
13   mts_c
14 end
```

$=_{cds,c}$

```
1  class A extends D
2    ads_a;
3    meth m ≙ (pds_m • c')
4    mts_a
5  end
6  class B extends A
7    ads_b;
8    mts_b
9  end
10 class C extends A
11   ads_c;
12   mts_c
13 end
```

provided

($\leftrightarrow$) .1 **super** and private fields do not appear in c'.

($\rightarrow$) .1 m is not declared in mts_a, and can only be declared in a class N, for any $N \leq A$, if it has parameters pds_m;

.2 m is not declared in any superclass of A in cds.

($\leftarrow$) .1 m is not declared in mts_b or mts_c;

.2 **super**.m does not appear in mts_b or mts_c nor in any class N such that $N \leq A$ and $N \not\leq B$ or $N \not\leq C$; and

.3 $N.m$, for any $N \leq A$ and $N \not\leq B$ or $N \not\leq C$, does not appear in cds, c, mts_a , mts_b or mts_c.

The work in (CORNELIO, 2004) follows from the formal derivation of refactoring rules using programming laws (BORBA *et al.*, 2004) that deal with commands, classes and also laws for data refinement as a basis for the proofs of the program transformations (described by refactorings). Thus, based on refactoring rules and, eventually on data refinement, the author of (CORNELIO, 2004) transform a system into a structured one according to a design pattern. The correctness proof of these refactoring rules is based on the application of programming laws whose soundness is proved against the language's semantics.

Rule 2.1 formalizes the refactorings (Pull Up Method) and (Push Down Method) from Fowler's catalog (OPDYKE, 1992) and is similar to the Law 2, with the difference that in the former case the same method in different subclasses is grouped together and moved to the superclass whilst in the latter case the method in one subclass is only moved to the superclass (and vice–versa). Thus, some provisos and conditions need to be different.

However, the authors of (PALMA, 2015) corrected some provisos of this rule since they discovered that the formulations in (CORNELIO, 2004) and in (NAUMANN; SAMPAIO; SILVA, 2012) do not ensure the soundness of the transformation (PALMA, 2015). We considered Rule 2.1 as

Rule 3 (Move Method). *Let N b an attribute of class M, $ops \cup \{m()\{c\}\}$ the method set of M, where m is only used locally in M. And ops_1 the method set of N such that $m()$ is not in ops_1. If c only refers to an attribute $b.x$ of N and a method $b.n()$ of b for theoretical neatness[2]. Define*

$$\begin{array}{lcl} ops' & \widehat{=} & ops[b.m()/m()] - \{m()\} \\ c' & \widehat{=} & c[x/b.x, n()/b.n()] \end{array}$$

where $F[a/b]$ stands for the substitution of all occurrences of b. We have

$$\begin{array}{l} cdecls; M[N\ b, ops \cup \{m()\{c\}\}]; N[ops_1] \\ \sqsubseteq\ cdecls; M[N\ b, ops']; N[ops_1 \cup \{m()\{c'\}\}] \end{array}$$

provided that $m()$ is not called from outside of M on the left hand side of the rule.

[1] This means local variables declared outside c.
[2] It can be the case that c refers to a number of attributes and a number of methods of N.

Figure 2 Rule 3 (Move Method) in (QUAN; ZONGYAN; LIU, 2008).

one of the inputs to our Alloy infrastructure and conclude the same as the work in (PALMA, 2015). The corresponding Alloy model is shown in Section 3.2.4 and the conclusions about the provisos missing and added are described in Section 4.4.3.3, along with the corresponding modified specification depicted in Rule 4.1 (the same shown in (PALMA, 2015)). Besides, the additional predicates, corresponding to the new provisos, are also described in Section 4.4.3.3.

We also translated another refactoring rule in (CORNELIO, 2004) –the rule 5.8, *Encapsulate Field*– and discovered that some adjustments in the provisos need to be done.

2.1.4 rCOS

As already mentioned, refactoring rules are formulated as rCOS refinement laws of OO specifications and programs in (QUAN; ZONGYAN; LIU, 2008). A rule for each of the six Fowler's refactoring rule categories is defined as an rCOS refinement law, while the others are presented in an extended report (LONG Q., 2005). *rCOS* supports typical OO constructs as well as some special statements for specification and refinement. In this section, we present some examples of these refinement laws; the ones that we evaluate using our Alloy infrastructure.

A refactoring rule is formalized as a refinement law (QUAN; ZONGYAN; LIU, 2008) of the form $cdecls_0 \bullet P_0 \sqsubseteq cdecls_1 \bullet P_1$, where the left–hand side is the original program and the right–hand side is the refactored one. Observe that the notation is similar to the other formalisms already mentioned, with the difference that in the case of a refinement law only one direction (from left to right) of the transformation is allowed (since it is a refinement). We can see that the notation of the rules is easy to understand as depicted in Figure 2. These rules can also be presented with UML diagrams (see Figure 3).

The rCOS refinement law, depicted in Figure 2, states that if a method of a class *M* only refers to attributes of another class *N*, one can move the method to class *N*. This rule is also depicted as a UML diagram (see Figure 3). The other rule we have analysed is Rule 6, *Pull*

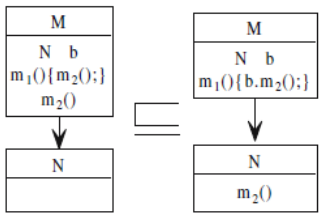


Figure 3 Representation of Rule 3 (Move Method) in UML (QUAN; ZONGYAN; LIU, 2008).

Rule 6 (Pull Up Method). *Assume M is the super class of N_1 and N_2, method $m()$ are declared with the same definition in both N_1 and N_2, and all attributes used in $m()$ are in M. Let ops be the operations declared in M which does not include $m()$. Then*

$$cdecls; M[ops]; N_1[M, \{m()\} \cup ops_1]; N_2[M, \{m()\} \cup ops_2]$$
$$\sqsubseteq\ cdecls; M[ops \cup \{m()\}]; N_1[M, ops_1]; N_2[M, ops_2]$$

Figure 4 Rule 6 (Pull Up Method) in (QUAN; ZONGYAN; LIU, 2008).

Up Method, depicted in Figure 4, which is very similar to refactoring Rule 2.1 presented in Section 2.1.3. However, Rule 6 is less rigorously specified since its only restriction is that all attributes used in *m()* are declared in M—its satisfaction allows to pull the method *m()* up to the class *M*. The rest of the rule is composed by the substitutions and definition of the set of methods in each class involved in the refinement.

2.2 The Alloy language and the Alloy Analyzer

The Alloy modeling language is a concise formal language, based on first–order logic. Some of its features are unique to Alloy, notably *signatures* and the notion of *scope*. Other constructs—modules, polymorphism, parameterized functions, and so on—are standard features of most programming and modeling languages, and have been designed to be as conventional as possible (JACKSON, 2006).

What is perhaps new to Alloy is the separation of the scope specification from the model itself, and the ability to adjust the scope in a fine–grained manner. This separation prevents the model from being polluted with analysis concerns, and makes it easy to run different analyses with different scopes without adjusting the model itself. The fine–grained control goes beyond static configuration parameters (such as the number of processes in a network) to bounds on

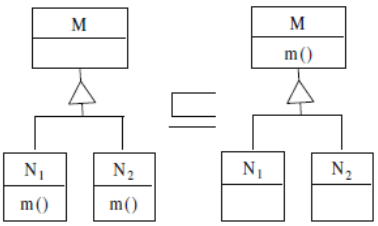


Figure 5 Representation of Rule 6 (Pull Up Method) in UML (QUAN; ZONGYAN; LIU, 2008).

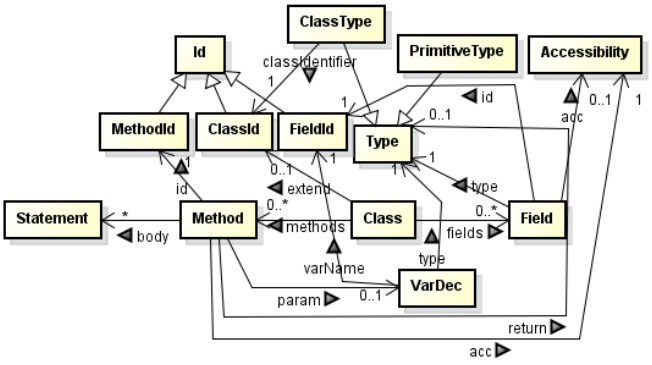
dynamically allocated data (such as the number of messages in a queue, or the number of objects in a heap) (JACKSON, 2006). In the next sections we give an overview of the Alloy language and the Alloy Analyzer.

2.2.1 The Alloy language

The Alloy language supports three different logic styles, which can be mixed and varied at will. In the predicate calculus style (usually too verbose), there are only two kinds of expressions: relation names, which are used as predicates, and tuples formed from quantified variables. In the navigation expression style, expressions denote sets, which are formed by navigating from quantified variables along relations. In the relational calculus style (usually too cryptic), expressions denote relations and there are no quantifiers at all (JACKSON, 2006).

An Alloy specification can be represented by signatures, fields, constraints (facts, predicates or assertions) and functions. Each element declared as a *signature* represents a *type* and can also be associated to other elements by *fields* (or *relations*) along with their types. For instance, Figure 6 shows a type *Class* (and other types it relates with) that owns the following fields: *extend*, *methods* and *fields* whose types, in turn, are, respectively: *ClassId*, *Method*, *Field*. The relation *extend* associates the class declared in the signature with at most one element of type *ClassId*—this is ensured by the keyword *lone*. The relations *methods* and *fields* represent the set of elements of types *Method* and *Field*, respectively. The keyword *one* can be seen in type *ClassType* in the declaration of the relation *classIdentifier*—in this case it means that a type *ClassType* can only be associated with exactly one *ClassId*. This keyword is also used in some other signatures such as *Field*, *VarDec* and *Method*. Observe that *ClassId*, *MethodId* and *FieldId* are all subsignatures or extensions of type *Id*. Subsignatures in Alloy are subsets mutually disjoint of parent signatures. Also observe that some signatures are declared as *abstract*. An *abstract* element means that it has no elements of its own that do not belong to its extensions. Because of this, these elements are not created as concrete instances in the Alloy Models, only their extensions.

The language of relations has constants and operators. Our models used almost all of Alloy operators defined in (JACKSON, 2006). Basically, operators fall into two categories. For the *set operators*, the tuple structure is irrelevant—they are considered as atoms. For the *relational* operators, the tuple structure is essential: these operators make relations powerful ((JACKSON, 2006), see section 3.4). The set operators (i.e. union, intersection, difference, subset, equality) were all used in our models, whereas the relational operators used were: arrow (product), dot (join), box (join), transpose, transitive closure, reflexive–transitive closure, and override (see section 3.4.3 in (JACKSON, 2006)). The most common is dot (join). The dot join (or just join) *p.q* of relations *p* and *q* is the relation you get by taking every combination of a tuple in p and a tuple in q, and including their join, if it exists. These relations can have any arity except they cannot be both unary (since that would result in a relation with zero arity) (JACKSON, 2006). When x is a scalar, and r is a binary relation, x.r is the set of atoms that x maps to. This is the most




```
abstract sig Id {}
sig ClassId, MethodId, FieldId extends Id {}
abstract sig Accessibility {}
abstract sig Type {}
abstract sig PrimitiveType extends Type {}
one sig Long_ extends PrimitiveType {}
sig ClassType extends Type{
 classIdentifier: one ClassId
}
abstract sig Class {
   extend: lone ClassId,
   methods: set Method,
   fields: set Field
}
sig Field {
   id: one FieldId,
   type: one Type,
   acc : one Accessibility
}
sig VarDec {
  varName: one FieldId,
  type: one Type
}
sig Method {
   id: one MethodId,
   param: lone VarDec,
   return: lone Type,
   acc: one Accessibility,
   body: seq Statement
}
```

Figure 6 A UML class metamodel and its representation in Alloy

common use of the join operator in our models as can be seen in Chapter 3. The box operator *[]* is semantically identical to join, but takes its arguments in a different order. For instance, the expression *e1[e2]* is the same as *e2.e1*. The use of this operator is observed in many predicates in Chapter 3.

The arrow product was used in Code 3.5 to define the relation *classDeclarations* in type *Program*. According to (JACKSON, 2006), the arrow product (or just product) p > q of two relations p and q is the relation you get by taking every combination of a tuple from p and a tuple from q and concatenating them. When p and q are sets, p > q is a binary relation. If one of p or q has arity of two or more, then p > q will be a multirelation. In case of the relation *classDeclarations* in type *Program*, the relation *q* corresponds to type *Class* and is restricted with the keyword *one*. Thus, this means that the relation *classDeclarations* in type *Program* simulates a mapping data structure because it only allows a combination of an element of the type *ClassId* with exactly one element of the type *Class*.

Another important relational operator used in our models is the transitive closure operator: ^. The transitive closure ^r of a binary relation r, or just the closure for short, is the smallest relation that contains r and is transitive. You can compute the closure by taking the relation, adding the join of the relation with itself, then adding the join of the relation with that, and so on (section 3.4.3.5 in (JACKSON, 2006)). For instance, the predicate *noCycleInExtends* uses this operator, as can be seen in Code 3.8, in the expression *c.^((p.classDeclarations).extend)*. If the expression

was only *c.((p.classDeclarations).extend)*, it would refer to the class identifier in the relation *extend* of the class referred by the expression *c.((p.classDeclarations))*—the variable *c* is the class identifier in the predicate parameter. Remember *p.classDeclarations* refers to the mapping data structure between class identifiers and classes. Thus, according to the relation that the transitive closure is applied to (that is, *((p.classDeclarations).extend)*, then the join of this relation with itself, and so forth, considering the first expression is *c.((p.classDeclarations).extend)*), represents all the class identifiers in the hierarchy of the class identifier *c*, passed as parameter.

The reflexive transitive closure operator is obtained by adding the identity relation to the just mentioned closure operator. It was also extensively used in our work. One example is in the predicate *fieldMatchesAndIsNotPrivate*, presented in Code 3.18. Observe that in there the operator was applied to the expression *(extend.(p.classDeclarations))*. If the expression was only *c.(extend.(p.classDeclarations))* it would refer to the class whose identifier is in the relation *extend* of the class *c*, passed as parameter. As the reflexive–transitive closure is applied, the identity relation is considered as an option, and because of this the expression can become only *c*. Thus, it is verified if the field f is in the set of fields of this class *c*. Another possibility is if this field is in the set of fields of the classes in the hierarchy of this class *c* (in the case the closure operator is applied since it is the another possibility for the reflexive transitive closure operator).

The constants in Alloy are represented by: **none**, **univ** and **iden**. **None** and **univ** represent the set containing no atom and every atom, respectively, whilst the relation *identity* is binary and contains a tuple relating every atom to itself (section 3.4.1 in (JACKSON, 2006)). In addition, when the operator **univ** is applied to a binary relation, it represents the set of the elements in the domain of this binary relation. For instance, the expression *univ.(p.classDeclarations)* represents all the elemts of type *ClassId* that comprises the domain of this mapping. On the other hand, expression like *(p.classDeclarations).univ* represents the image of the relation, that is, the set of elements of type *Class* representing the image of the binary relation *p.classDeclarations*. The constant **univ** was intensively used in our models as can be observed in Code 3.5.

The override operator was also used in our Alloy models, as can be observed in Code 3.21, line 28. In this case it is used to update the relation *classDeclarations* in the type *Program*, replacing the class of the original program for the respective class of the resulting program, keeping the same identifier for all the classes involved in the transformation. This will be detailed in Section 3.2. In addition, the transpose operator was also used in our models. One example is in Code 3.21, lines 36 and 37. Consider the expression *c.~((ss.classDeclarations).extend)*. If the expression was only *c.((ss.classDeclarations).extend)* (without the transpose), it would refer to the class identifier in the relation *extend* of the class represented by the expression *c.(ss.classDeclarations)*. As the transpose operator is used, the expression refers to the sons of the class *c*.

With regard to Alloy constraints, a *fact* records a constraint that is always assumed to hold in the model. On the other hand, an assertion, marked by the keyword *assert*, introduces a constraint that is intended to follow from the facts (and the predicates expanded) of the model.

The command *check* tells the analyzer to find a counterexample to the assertion: that is, an instance that makes it false (JACKSON, 2006). An example of an assertion and its use is given in Section 4.1. A predicate defines a reusable constraint. Many examples of predicates are shown in Chapter 3. Observe that the use of the navigational style is predominant. For instance, in predicate *wellFormedProgram*, the quantified variable *c* (of type *ClassId*) navigates along with the elements in the set represented by the expression *(p.classDeclarations).univ*. It is ensured that if an element *c* belongs to this set (the type of each element is *Class*), then it is well–formed. From an analysis perspective, detailed further in Section 2.2.2, there is no fundamental difference between assertions and predicates. Running a predicate involves searching for an instance of its constraint; checking an assertion involves searching for an instance of the negation of its constraint. So, checking an assertion C is equivalent to running a predicate not C.

2.2.2 The Alloy Analyzer

The Alloy Analyzer is a model finder able to generate what we call Alloy *instances* from an Alloy model. These instances can be displayed in graphical form, or in textual form, or as an expanding tree. It can also generate model diagrams from model text. The variables that are assigned in an instance comprise: the sets associated with the signatures, the relations associated with the fields, and, for a predicate, its arguments.

For doing this generation, a scope must be specified. A scope sets a bound on the size of each of the top-level signatures, and, optionally, on subsignatures too (described in Section 2.2.1). An instance is within a scope if each signature constrained by the scope has no more elements than its associated bound permits. To perform an analysis, the analyzer considers all candidate instances within the scope. Of course, the number of candidates is usually so large that an explicit enumeration would be infeasible. The analysis therefore uses pruning techniques to rule out whole sets of candidate cases at once (JACKSON, 2006). If it finds no instance, it is guaranteed that none exists within that scope, although there might be instances in a larger scope.

Every analysis involves solving a constraint: either finding an instance (for a run command) or finding a counterexample (for a check). The Alloy Analyzer is therefore a constraint solver for the Alloy logic. In its implementation, however, it is more of a compiler, because, rather than solving the constraint directly, it translates the constraint into a boolean formula and solves it using an off-the-shelf SAT solver (JACKSON, 2006). The Alloy Analyzer is bundled with several SAT solvers and a preference setting lets one choose which is used.

The resulting boolean formula is passed to the SAT solver. If it finds no solution, the Alloy Analyzer just reports that no instance or counterexample has been found. If it does find a solution, the solution is mapped back into an *instance*. Each Alloy instance is composed by the objects generated for each signature defined in the specification, according to the constraints specified. The analyzer lets one customize how instances are displayed; one can select a set and project all relations in the instance onto the columns associated with that set.

The translation from the Alloy logic to a boolean formula, performed by the Alloy

Analyzer, applies a variety of optimizations, the most significant one (and interesting) is *symmetry breaking*, that is: every Alloy model has a natural symmetry—one can take any instance of a command and create another one by permuting the atoms. This means that when an analysis constraint has a solution, it actually has a whole set of solutions corresponding to all the ways in which the atoms of the solution can be permuted (JACKSON, 2006).

The Alloy approach is therefore less capable of establishing the absence of bugs, but when there is a bug, it may be more rapidly found by Alloy SAT-based analysis than by model checking, because of the depth-first nature of SAT solving. The machine description language of most model checkers is very low-level, so describing a protocol such as this tends to be much more challenging. Model checkers are generally capable of exhausting an entire state space. In an Alloy trace analysis, only traces of bounded length are considered, and the bound is generally small.

2.3 Metamodels for Java

The Model Driven Architecture (MDA) (KLEPPE; WARMER; BAST, 2003) is a framework for software development defined by the Object Management Group (OMG). Key to MDA is the importance of models in the software development process. Within MDA the software development process is driven by the activity of modeling your software system. Traditionally, the transformations from model to model, or from model to code, are done mainly by hand. Many tools can generate some code from a model, but that usually goes no further than the generation of some template code, where most of the work still has to be filled in by hand (KLEPPE; WARMER; BAST, 2003).

The traditional four–layer Object Management Group modeling infrastructure, depicted in Figure 7, consists of a hierarchy of model levels, each (except the top) being characterized as an instance of the level above. The bottom level, M0, holds the *userdata* – the actual data objects the software is designed to manipulate. The next level, M1, is said to hold a *model* of the M0 user data. User models reside at this level. Level M2 holds a model of the information at M1. Because it is a model of a model, it is often referred to as a metamodel. Finally, level M3 holds a model of the information at M2, and therefore is often called the meta–metamodel (ATKINSON; KüHNE, 2003). For historical reasons, it is also referred to as the Meta–Object Facility (GROUP, 2016). In this work, the term metamodel is grounded on the prefix *meta* concept, commonly used to denote that something is applied twice: in this case a model of models. More specifically, we use the term Java metamodel to represent a model upon from another model, that is: the EBNF description in the Java Language Specification (ORACLE, 2016). However, we can find similarities among the MDA aproach and ours. For instance, the UML metamodel produces instances as concrete models whereas our metamodel produces instances that are models characterizing Java well–formed programs, if the *wellFormed* predicate (available in this model) is used for these programs. Besides, we enhance the MDA approach regarding the code

generation upon from an Alloy model specification, as further described.

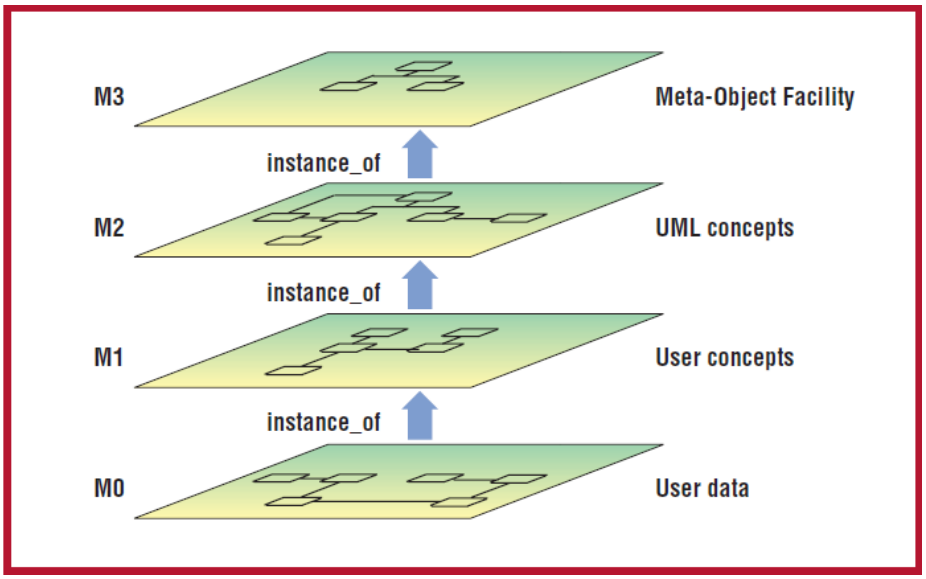


Figure 7 Traditional Object Management Group modeling infrastructure (ATKINSON; KüHNE, 2003).

Generally, in Model–Driven Architecture methods, there is a creation of meaningful models, which evolve systematically through model transformations (that can specify and apply patterns, for instance, that reflect useful and reusable design abstractions), and thus these models are refined and can generate code. In (FRANCE *et al.*, 2003), a pattern–based approach to refactoring is presented, where a process of transforming a model using a design pattern is presented. Model transformation is defined as a process to modify a source model to produce a target model. They call metamodel the transformation specifications that characterize families of transformations. With the development of these metamodels, they consider achieving rigorous pattern–based refactorings. In addition, they consider that metamodels act as points against which they can check model transformations for conformance. They claim that popular descriptive forms of design patterns, although effective at communicating design experiences to software developers, are too informal to use as a basis for developing pattern–aware modeling tools, and conclude that they need precise forms of patterns, called *pattern specification*, and to codify them in tools. Besides, transformation rules must also be precisely specified if these tools are to support them and, for solving this problem, they use a metamodeling approach to specify both patterns and transformation rules.

Our approach defines Alloy Models that act as a source and as a target model in a transformation process because it is able to generate the source and target representations of programs, supported by a transformation specification and its corresponding rules in Alloy. In this way, we are using a formal infrastructure as a basis for developing or applying program transformations. In addition, through the use of the Alloy Analyzer (see Section 2.2.2) together with our Alloy–To–Java Translator (explained in Chapter 4), we also generate Java code which contains programs before and after the application of transformations. Thus, we rely on the

bounded exhaustive analysis offered by the Alloy Analyzer to capture all the variations of a transformation, according to a given scope (and hence its application in a program), specified in an Alloy model. Not capturing all the variations of a pattern in one specification is described as a problem in (FRANCE *et al.*, 2003), where it is necessary moving, create or remove elements to match the structure of what they call solution specification. This formal form is only mentioned in the article but not described or shown in detail.

The work in (JUDSON; CARVER; FRANCE, 2003) represents a continuation of the one in (FRANCE *et al.*, 2003) and details the metamodeling approach, which defines families of transformations in terms of classes of model elements that are created and deleted during transformations. The classes of model elements are distinguished in the metamodel as subclasses of UML metamodel classes. An example of the Bridge pattern is used. In addition, the transformation approach is described in two levels: M1' and M2', where M2' is an extension of the UML metamodel level that supports the specification of transformation rules and M1' is an extension of the UML model level that supports representation of model transformations. A model transformation at the M1' level (model level) takes a source model and transforms it to a target model. The M1' model transformations are characterized by a transformation pattern at the M2' level that includes a source pattern that characterize source models and a target pattern that characterizes target models. In this direction, we can make an analogy between this approach and ours in the sense that the M1' layer can be represented by our Alloy models, since they establish a transformation pattern, a source and target pattern. On the other hand, the M2' layer can be represented by the Alloy Analyzer as it generates instances conforming to each of these patterns, or else, representations of Java classes (in our case) characterizing the starting– and resulting hand-side classes of a transformation. The main difference is that, in our case, the classes are not removed and created, but tagged (or identified) as classes before and after a transformation. Another similarity is that their approach is also used to model refactorings.

Metamodels involving Java are also addressed in the literature (MILLER; MUKERJI, 2003; OMG, 2015). In (ALANEN *et al.*, 2003), algorithms are described to map any BNF grammar into a metamodel and also the other way around. An example for a subset of Java is shown. However, besides its subset being much smaller than ours, little is described with regard to how well–formedness rules are guaranteed in their Java metamodel. In our case, our Java metamodel is also generated from the Java EBNF described in (ORACLE, 2016), but in a manual way (not automatically) and with some simplifications as our goal is only to cover a subset of the Java language, but broader than theirs.

3

JAVA AND TRANSFORMATION–SPECIFIC MODELS

As described in Chapter 1, our strategy is comprised by four steps (see Figure 8 again), where the main goal is validating transformation specifications. In these steps, Alloy models are used as inputs to the Alloy Analyzer. The transformation–specific Alloy model describes the transformation specification; this model includes predicates and some auxiliary functions necessary to encode the conditions, provisos and substitutions of the specific transformation being analysed. In addition, this model uses all the elements (i.e. types, predicates, auxiliary functions, and so forth) defined in the OO metamodel. No additional Alloy signature, other than the ones in the OO model, is necessary in the transformation–specific model. On the other hand, the validator models detect static semantics errors or behavioural problems of the transformation, depending on which validator model is used.

The transformation–specific model owns a main predicate, which is responsible for the transformation. This predicate (along with its parameters) is the parameter for the commands *run* or *check* in the validator models, along with a scope defined for the specific transformation being analysed. The validator models are actually the inputs for the Alloy Analyzer, but these models use the transformation–specific, which in turn uses the OO model (see Figure 8). Then, the Alloy Analyzer generates Alloy instances characterizing the programs before and after the transformation, and according to the restriction or restriction violation, respectively—it will depend on the command specified.

This chapter describes the OO model, that gives support to the transformation–specific model, in Section 3.1. Section 3.1.1 details the subset of the Java language considered in our OO Model since a correspondence between our OO Model and the Java Language Specification (JLS) (ORACLE, 2016) is explicitly described, with the indication of the elements in JLS that were disconsidered or simplified in our OO Model. Section 3.1.2 complements the description of the OO Model with details of each relevant predicate as well as the well–formedness rules. Finally, the transformation–specific model is detailed in Section 3.2 where each subsection describes a model for each transformation analysed in this work.

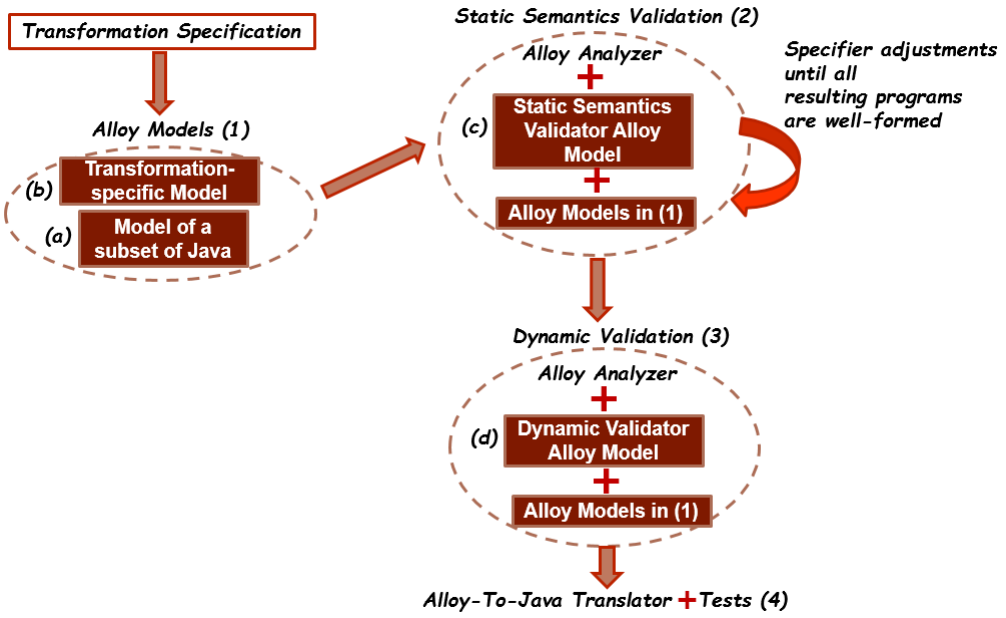


Figure 8 Overview of our strategy.

3.1 The OO Metamodel

Our OO model brings together some OO features which makes possible to represent OO programs in Alloy. It is important to emphasize that, although the OO model was used to evaluate transformations in languages other than Java, we define the elements of this metamodel as close as possible to the ones defined in the Java Language Specification (JLS (ORACLE, 2016)) syntactic grammar as we use Java to express concrete transformations in the behavioural analysis phase. Besides, the proximity with the JLS eases the understanding and enhancement, if necessary. However, we had to do some simplifications not to compromise the scalability of the Alloy model. In Section 3.1.1, we detail the mapping (and their simplifications) between what exists in the JLS (ORACLE, 2016) syntactic grammar and our Alloy signatures.

Our OO model defines types through signatures, shown in Figure 9. We started to explain our OO metamodel in Section 2.2.1 with Figure 6 (left), which represents a subset of our OO metamodel depicted in Figure 9. As seen in Figure 6 (left), the signatures and their respective relations are analogous to classes and associations in the UML class diagram. In addition to the signatures already described in Section 2.2.1, Code 3.1 shows the *MethodInvocation* signature in Alloy. The *pExp* relation links a *MethodInvocation* with exactly one *PrimaryExpression* whilst the *realParam* relation links a *MethodInvocation* with at most one *Expression*, excluding the *PrimaryExpression´s this*, *super* and *newCreator* and the *AssignmentExpression* as well. The reason is simplifying type checking as we almost do not do it. We assume all methods have a return (of type *long*), and when there is a formal parameter, this one is also of type *long*. Because of this, the only expression allowed for *realParam* relation is *MethodInvocation*—for simplification purposes. This constraint can be seen in the associations with *MethodInvocation* in Figure 9.

Mapping the relations of the *MethodInvocation* type to a concrete example, it would be *"pExp.methodInvoked (realParam)"*, where we replace each relation name to a concrete instance that is generated by the Alloy Analyzer. For instance, in case the *pExp* relation is of *newCreator* type (whose class in its *cf* relation is *A*), whilst the *methodInvoked* is a method identifier like *method01* and the *realParam* is a method invocation like *this.method02()*, then the *MethodInvocation* type in question is represented as *"new A().method01(this.method02())"*. This is translated from an Alloy instance to a Java program by our Alloy–To–Java translator. The remaining Alloy types defined in our OO model and described in this section follows the same reasoning.

Code 3.1 Representation of the MethodInvocation signature

```
1 sig MethodInvocation extends Statement {
2    pExp: one PrimaryExpression,
3    methodInvoked: one MethodId,
4    realParam: lone {Expression - this_ - super - newCreator -
        AssignmentExpression }
5 }
```

Some relevant signatures in our OO metamodel were not described in Section 2.2.1 but they are present in Figure 9. They are important to ease the understanding of some predicates described in this chapter. These signatures are: the *AssignmentExpression* signature (Code 3.2), the *FieldAccess* signature (Code 3.3), and the group of *PrimaryExpression* subsignatures (Code 3.4). The *AssignmentExpression* signature contains two relations: the *pExpressionLeft* represented by only a *FieldAccess* type (for simplification purposes) and the *pExpressionRight* relation (with exactly one *Expression*, excluding the expressions *this*, *super* and *newCreator* and the *AssignmentExpression* as well—for the same reason explained earlier for *realParam*). These restrictions can simplify our Alloy Models as much as possible but not compromising our transformation evaluations (see Chapter 4).

Code 3.2 Representation of the AssignmentExpression signature

```
1 sig AssignmentExpression extends Statement {
2    pExpressionLeft: one FieldAccess,
3    pExpressionRight: one {Expression - this_ - super -
        newCreator - AssignmentExpression}
4 }
```

In addition, the *FieldAccess* signature also contains a *pExp* relation, similarly to the one in *MethodInvocation*, and an *id_fieldInvoked* relation to represent the *id* of the field being invoked in the *FieldAccess* expression.

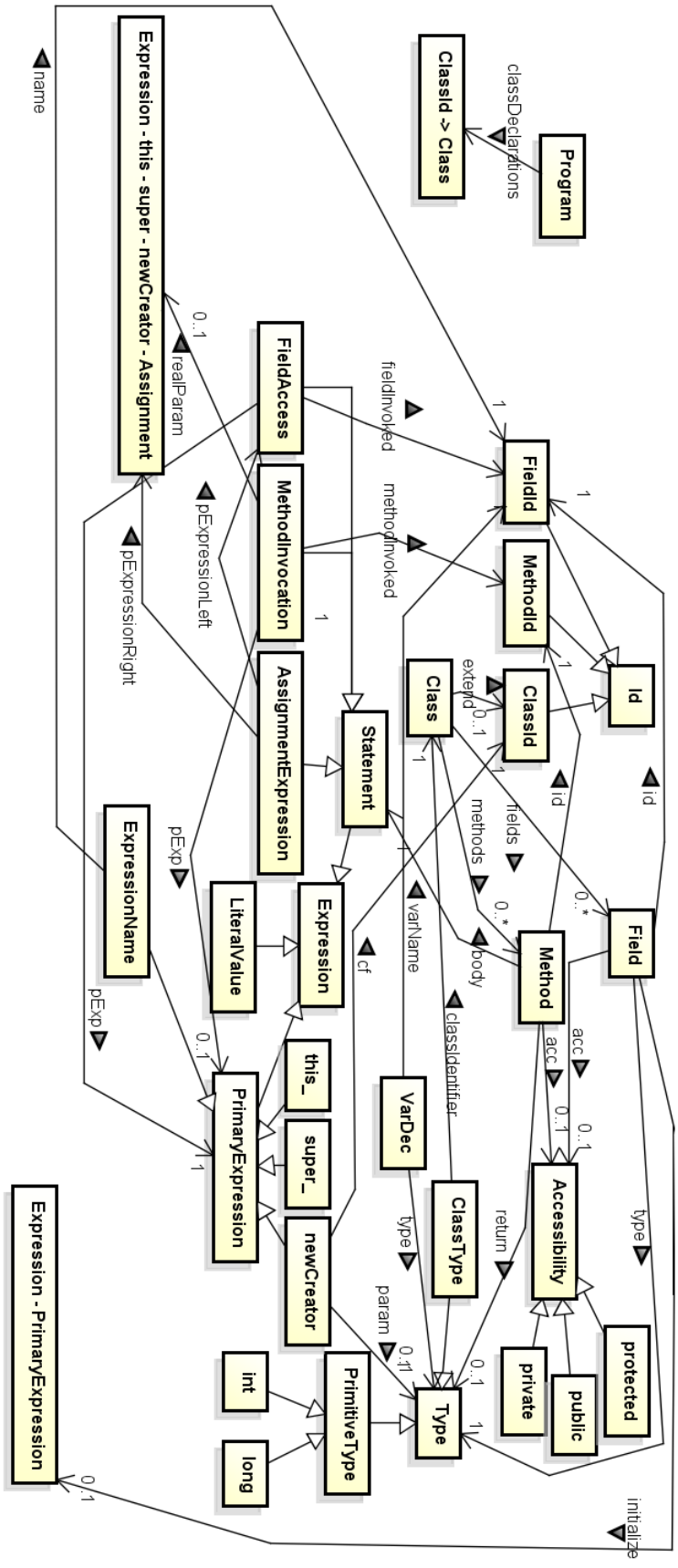


Figure 9 OO Metamodel embedded in Alloy

Code 3.3 Representation of the FieldAccess signature

```
1 sig FieldAccess extends Statement{
2    pExp: one PrimaryExpression,
3    id_fieldInvoked: one FieldId
4 }
```

Finally, Code 3.4 contains the group of *PrimaryExpression* subsignatures, which is: *this*, *super*, *newCreator* (with its *cf* relation, that represents the *ClassId* linked to the *newCreator*) and *ExpressionName* (with its *name* relation, that represents the *FieldId* linked to the *ExpressionName*). Besides, our OO model also contains well–formedness rules that are diluted into its *wellFormed* predicates: one for each signature in our model—see Section 3.1.2. The well–formedness rules are presented when we describe these predicates. They were validated through the Java Compiler when our Alloy–To–Java Translator (Section 4.3) compiles each of the starting programs (where the respective validated transformations were applied) and no compiler errors were found in them.

Code 3.4 Group of PrimaryExpression's signatures

```
1  abstract sig PrimaryExpression extends Expression {
2  }
3
4  sig this_, super extends PrimaryExpression {}
5
6  sig newCreator extends PrimaryExpression {
7     cf :one ClassId
8  }
9
10 sig ExpressionName extends PrimaryExpression{
11    name: one FieldId
12 }
```

As can be seen in this section and in Appendix A, only a subset of JLS was chosen not to increase unnecessarily the complexity of the Alloy model, and consequently to avoid compromising its scalability, since not all elements in the JLS syntactic grammar are necessary to analyse specific transformations and the more signatures and relations available in the model, the greater are the constraints and predicates to guarantee programs well–formedness. We verify that it is possible to analyse transformations with only a subset of the Java language or OO features, and this subset may even vary (minimally) depending on the transformations being analysed.

3.1.1 Correspondence between the Java Language Specification and our OO metamodel

As discussed before, we have followed the pattern of the JLS syntactic grammar as much as possible. The figures presented in this section exhibit subparts of the JLS specification in (ORACLE, 2016). The *MethodDeclaration* definition, depicted in Figure 10, is represented

as the *Method* signature in our OO model. Observe that the *MethodModifier* element was simplified in our model by the relation *acc* in the *Method* signature. This is represented by only an *Accessibility* type element that in turn can be *public*, *private* or *protected* (its subsignatures). As the *Accessibility* signature is defined as *abstract*, only its subsignatures appear as an Alloy element instance in our Alloy model.

MethodDeclaration:
{MethodModifier} MethodHeader MethodBody

MethodModifier:
(one of)
Annotation `public protected private`
`abstract static final synchronized native strictfp`

MethodHeader:
Result MethodDeclarator [Throws]
TypeParameters {Annotation} Result MethodDeclarator [Throws]

Result:
UnannType
`void`

MethodDeclarator:
Identifier `(` *[FormalParameterList]* `)` *[Dims]*

FormalParameterList:
ReceiverParameter
FormalParameters `,` *LastFormalParameter*
LastFormalParameter

Figure 10 Subpart of the JLS syntactic grammar.

As can be observed, the definition for the *MethodModifier* element in JLS is more detailed since it has more production rules than simply to be *public*, *private* or *protected*; however, according to our transformation specification elements, it was not necessary to all production rules of the *MethodModifier* element in JLS to be completely defined in our model. The *MethodHeader* element in JLS was also simplified and we adopt only the first possibility in the grammar: *Result MethodDeclarator [Throws]*, excluding the [*Throws*] element. The *Result* element is represented by the *return* relation in our OO model, which links our *Method* signature with at most one *Type*. When the Alloy instance is generated without this *Type* element, then a *void* method is represented (second case for the *Result* element in the JLS). Otherwise, a non–void method is represented and thus the first case for the *Result* element in the JLS is contemplated. In addition, our *Method* signature contains the relations *id* (corresponding to the *Identifier* element in the *MethodDeclarator* in JLS) and *param* (corresponding to the *FormalParameterList* element in JLS). Aiming at simplification, the [*Dims*] element in the *MethodDeclarator* in JLS was not considered and we limited the number of formal parameters of a method to at most one element—

FormalParameterList:
ReceiverParameter
FormalParameters , *LastFormalParameter*
LastFormalParameter

FormalParameters:
FormalParameter { , *FormalParameter*}
ReceiverParameter { , *FormalParameter*}

FormalParameter:
{*VariableModifier*} *UnannType VariableDeclaratorId*

VariableModifier:
(one of)
Annotation `final`

LastFormalParameter:
{*VariableModifier*} *UnannType* {*Annotation*} . . . *VariableDeclaratorId*
FormalParameter

ReceiverParameter:
{*Annotation*} *UnannType* [*Identifier* .] `this`

Figure 11 FormalParameterList element and its dismemberment.

see the keyword *lone* in our *Method* signature (see Figure 6). Thus, the *FormalParameterList* element was resumed to only the *LastFormalParameter* element, which in turn was resumed to the *FormalParameter* element—as shown in Figure 11. In the *FormalParameter* element, we only consider the elements *UnanmType* and *VariableDeclaratorId*, since the *VariableModifier* element is optional. Observe that the sequence *UnanmType VariableDeclaratorId* contained in *FormalParameter* element is exactly what our *VarDec* signature represents—it is the type of the *param* relation in our *Method* signature.

Finally, the last element composing the *MethodDeclaration* element—the *MethodBody* element, shown in Figure 12—is equivalent to the type *Block* in our OO model. In turn, a *Block* is a potentially empty or is a sequence of *BlockStatements* elements. As a simplification, in our OO model we assume that a *BlockStatements* element can only be represented by a *Statement* element, the third option for the *BlockStatements* element. Hence, in our Alloy model, a type *Block* is represented by a sequence of *Statement* types. In turn, we consider the *Statement* element (in Figure 13) always as a *StatementWithoutTrailingSubstatement* element, which in turn is always considered by us as an *ExpressionStatement* element that is always a *StatementExpression* in JLS grammmar (see Figure 14). A *StatementExpression* is then resumed to a *MethodInvocation* or an *Assignment*, which are the two subsignature possibilities of the *Statement* abstract type signature in our model.

With regard to the *MethodInvocation* element in the JLS, we adopt the third, fourth and fifth options (see Figure 15), with some simplifications. For instance, *ExpressionName*

MethodBody:
Block
;
Block:
{ [BlockStatements] }

BlockStatements:
BlockStatement {BlockStatement}

BlockStatement:
LocalVariableDeclarationStatement
ClassDeclaration
Statement

LocalVariableDeclarationStatement:
LocalVariableDeclaration ;

LocalVariableDeclaration:
{VariableModifier} UnannType VariableDeclaratorList

Figure 12 MethodBody element and its dismemberments.

(first element in Figure 15) in the third option was simplified to be always an *Identifier* and this *Identifier* can only be of *FieldId* type in our OO model—this corresponds to the relation *name* (of type *FieldId*) of our *ExpressionName* type signature. Our *varName* relation in *VarDec* type of our Alloy model is also of *FieldId* type for compatibility reasons.

With regard to the *Primary* element in the JLS grammar, contained in the fourth option, we consider *this* and a *ClassInstanceCreationExpression* as possible values for this element, since a *Primary* element can be a *PrimaryNoNewArray* which in turn can be a *this* or *ClassInstanceCreationExpression* element (see Figure 16).

In the latter case we restrict this element to be *UnqualifiedClassInstanceCreationExpression* (see Figure 17), which is then restricted to *new ClassOrInterfaceTypeToInstantiate* (since the other possible elements are optional) which is finally restricted to *new Identifier*, where the *Identifier* in our OO model is the type *ClassId*. The formation *newClassId* is represented in our model by the type *newCreator*, which has a relation called *cf* that links this type with a type *ClassId*. The fifth option has a *super* element in its beginning, which is also a signature in our OO model. *ExpressionName*, *this*, *newCreator* and *super* are all subsignatures of type *PrimaryExpression* in our model, which is the type of the relation *pExp* in our *MethodInvocation* signature. In addition, we do not consider the [*TypeArguments*] element in any of the options and our [*ArgumentList*] element has always one as the maximum size—it corresponds to the relation *realParam* in our type *MethodInvocation*. We also consider the *Identifier* element in all of the options as the *MethodId* of the method being invoked, which is represented by the relation *id_methodInvoked* in our type *MethodInvocation*.

On the other hand, the *Assignment* element in JLS, the other possibility we consider

Statement:
StatementWithoutTrailingSubstatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement

StatementWithoutTrailingSubstatement:
Block
EmptyStatement
ExpressionStatement
AssertStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStatement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement

Figure 13 Statement element and its dismemberments.

to the *StatementExpression* element in JLS, is represented as a sequence like *LeftHandSide AssignmentOperator Expression*, shown in Figure 15. In our OO model, we have an type *AssignmentExpression* that has two relations, as already mentioned: the *pExpressionLeft*—whose type is *FieldAccess* which is then a possibility for the *LeftHandSide* element in JLS—and the *pExpressionRight*. The type of this last one is *Expression*, excluding *this*, *super*, *newCreator* (for simplification in type checking mechanism as earlier mentioned) and *AssignmentExpression*—to avoid recursion calls in the model. The Alloy Analyzer only allows a maximum depth size of 3 in recursion calls, meaning the Alloy Analyzer can unroll recursive calls up to 3 times only, regardless of the scope of our analysis, and, even so, unless we configure this in the Alloy Analyzer tool. The *AssignmentOperator* element in JLS is considered in our model to be always the *equals* or *assignment* operator. Finally, the *FieldAccess* element in JLS was simplified in our OO model to its first and second option, depicted in Figure 15.

Likewise described for the *MethodInvocation* element, our *FieldAccess* type signature has the *pExp* relation (whose type is *PrimaryExpression*)—which embraces the *Primary* and *super* elements in JLS, respectively, for the first and second options for the *FieldAccess* element—and the *id_fieldInvoked* relation that corresponds to the *Identifier* element in JLS which in this case is of type *FieldId* in our OO model.

A program is defined through two main relations: the *classDeclarations* relation, which represents a mapping between *ClassId* and *Class* elements, and the *main* relation that represents the main method of the specific program. Because of this mapping between *ClassId* and *Class*

ExpressionStatement:
StatementExpression ;

StatementExpression:
Assignment
PreIncrementExpression
PreDecrementExpression
PostIncrementExpression
PostDecrementExpression
MethodInvocation
ClassInstanceCreationExpression

Figure 14 ExpressionStatement metamodel.

elements in the type *Program*, the type *Class* does not contain a relation *id* with a *ClassId* as its type—it is not necessary. Observe that this is not the case for the *Method* and *Field* signatures (see Figure 6). For these cases, we opt not to work with mappings (between *Method* and *MethodId*, and *Field* and *FieldId*) due to scalability problems we experienced using the Alloy Analyzer. An interesting thing we guarantee in our Alloy instances is that the value of the *ClassId* element remains the same for both starting and ending programs (it is unique for each pair of classes in the starting and resulting programs) of a transformation while the *Class* element can vary in the different programs according to the restrictions of the transformation—this is described in details in Section 3.2.

Code 3.5 Representation of a Program

```
1 sig Program {
2     classDeclarations: ClassId -> one Class,
3     main: Method
4 }
```

3.1.2 Predicates in our OO metamodel – specifying well–formedness rules

Each predicate in our OO model captures well-formedness rules, which are responsible for guaranteeing the correct static semantics of the instances, according to the scope (of elements) defined. The *wellFormedProgram* is the main predicate in the OO model and guides the invocation of the other ones in a cascade effect as depicted in Figure 18. For instance, as a program is comprised by a mapping from the identifiers of the classes to each corresponding class type (see line 2, Code 3.5), a program is well–formed exactly when each of its classes is also well–formed (line 3, Code 3.6).

The class well–formedness is guaranteed by the *wellFormedClass* predicate (see Code 3.7). Following the usual OO structure, a class is well–formed in our model if each of its method is well–formed (see line 9, Code 3.7). A method is well–formed when its statements are (see line 6, Code 3.12), and so forth.

ExpressionName:
Identifier
AmbiguousName . *Identifier*

FieldAccess:
Primary . *Identifier*
`super` . *Identifier*
TypeName . `super` . *Identifier*

MethodInvocation:
MethodName (*[ArgumentList]*)
TypeName . *[TypeArguments] Identifier* (*[ArgumentList]*)
ExpressionName . *[TypeArguments] Identifier* (*[ArgumentList]*)
Primary . *[TypeArguments] Identifier* (*[ArgumentList]*)
`super` . *[TypeArguments] Identifier* (*[ArgumentList]*)
TypeName . `super` . *[TypeArguments] Identifier* (*[ArgumentList]*)

ArgumentList:
Expression {, Expression}

AssignmentExpression:
ConditionalExpression
Assignment

Assignment:
LeftHandSide AssignmentOperator Expression

LeftHandSide:
ExpressionName
FieldAccess
ArrayAccess

Figure 15 Dismemberments of the StatementExpression.

Primary:
PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray:
Literal
ClassLiteral
`this`
TypeName . `this`
(*Expression*)
ClassInstanceCreationExpression
FieldAccess
ArrayAccess
MethodInvocation
MethodReference

Figure 16 Primary element and its dismemberments.

ClassInstanceCreationExpression:
UnqualifiedClassInstanceCreationExpression
ExpressionName . UnqualifiedClassInstanceCreationExpression
Primary . UnqualifiedClassInstanceCreationExpression

UnqualifiedClassInstanceCreationExpression:
new *[TypeArguments]*
ClassOrInterfaceTypeToInstantiate (*[ArgumentList]*) *[ClassBody]*

ClassOrInterfaceTypeToInstantiate:
{Annotation} Identifier {. {Annotation} Identifier}
[TypeArgumentsOrDiamond]

TypeArgumentsOrDiamond:
TypeArguments
<>

Figure 17 ClassInstanceCreationExpression element and its dismemberments.

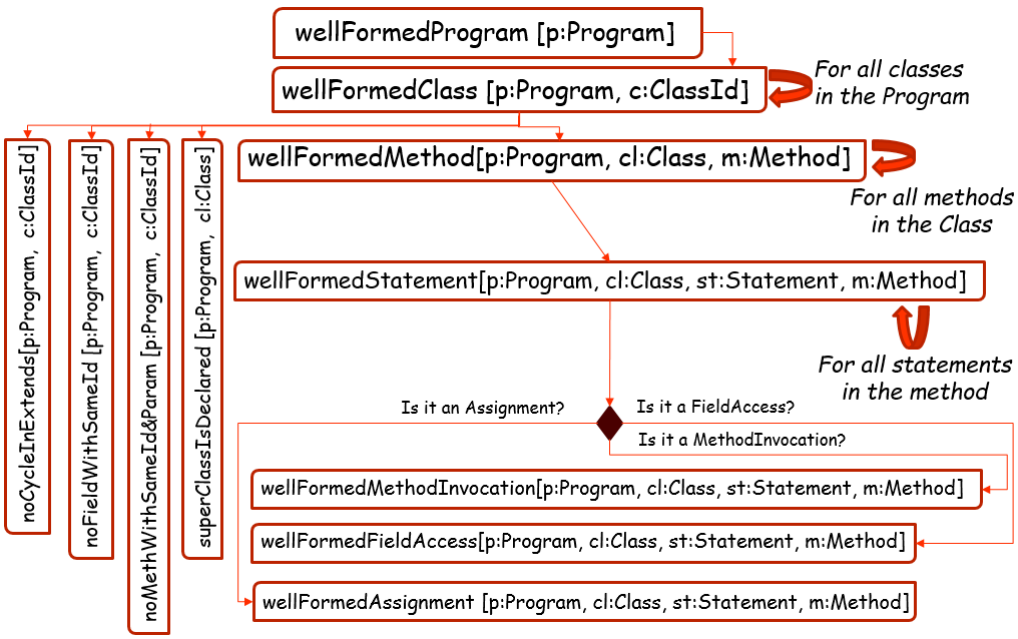


Figure 18 An overview of the main predicates in our OO Model

Code 3.6 Representation of the *wellFormedProgram* predicate

```
1 pred wellFormedProgram [p:Program] {
2     all c:ClassId | c in (p.classDeclarations).univ =>
3     wellFormedClass[p,c]
4 }
```

With regard to the class well–formedness, it is necessary that not only each of its methods be well–formed but also other restrictions guaranteed through the other predicates present inside the predicate *wellFormedClass*, as shown in Code 3.7 and in Figure 18. Firstly, it is necessary that there is no cycle in the *extend* relation of the class (see line 2, Code 3.7); secondly, no distinct fields with the same id (line 3 and corresponding definition in Code 3.9), and, thirdly, no distinct methods with the same id and parameters (line 4 and Code 3.10). Finally, completing the requirements for a class to be well–formed in our OO model, its superclass needs to be in the mapping representing all the classes in the specific program (line 7 and Code 3.11).

Code 3.7 Representation of the *wellFormedClass* predicate

```
1  pred wellFormedClass[p:Program, c:ClassId] {
2      noCycleInExtends[p,c]
3      noFieldWithSameId[p,c]
4      noMethWithSameIdAndParam[p,c]
5
6      let class = c.(p.classDeclarations) {
7          superClassIsDeclared[p,class]
8          all m: Method | m in class.methods =>
9          wellFormedMethod[p,class,m]
10     }
11 }
```

The predicate in Code 3.8 ensures that a class does not have cycles in its *extend* relation when its *ClassId* (passed as parameter for the predicate) is not in its *extend* relation nor in the *extend* relation of the classes in its hierarchy (line 2, Code 3.8). Observe that the expression $c.(p.classDeclarations)$ retrieves the corresponding *Class* element to the *ClassId*, represented by the c variable, in the p's Program mapping. Then, the *extend* relation of this *Class* can be retrieved through the expression $c.(p.classDeclarations).extend$. We use the transitive closure operator (^) in order to retrieve the set of all ids (type *ClassId*) in the extend relation of each class in the hierarchy of a specific *ClassId* c passed as parameter. In this way, c cannot be in this set to avoid a cycle in the *extend* relation.

Code 3.8 Representation of the *noCycleInExtends* predicate

```
1 pred noCycleInExtends[p:Program, c:ClassId] {
2     c !in c.^((p.classDeclarations).extend)
3 }
```

The predicate in Code 3.9 guarantees that a class (whose identifier is passed as one of its parameters) does not own two distinct field with the same identifier (this situation would cause a compilation error). Predicate *noMethWithSameIdAndParam* (see Code 3.10) is similar but concerning methods. The difference is that the distinct methods in the same class can have the same identifier if their parameters are different (number or the corresponding types).

Code 3.9 Representation of the *noFieldWithSameId* predicate

```
1 pred noFieldWithSameId[p:Program,c:ClassId] {
2     no disj f1,f2: p.classDeclarations[c].fields |
3     f1.id = f2.id
4 }
```

Code 3.10 Representation of the *noMethWithSameIdAndParam* predicate

```
1 pred noMethWithSameIdAndParam[p:Program,c:ClassId] {
2     no disj m1,m2: p.classDeclarations[c].methods |
3     m1.id = m2.id &&
4     #(m1.param) = #(m2.param) &&
5     (m1.param.type = m2.param.type)
6 }
```

Predicate *superClassIsDeclared* (see Code 3.11) checks if the identifier in the *extend* relation of a class (passed as one of the predicate parametera) is in the domain of class identifiers contained in the mapping of the specific program *p* (also passed as parameter). In other words, the predicate checks if the super class of the class passed as parameters exists in the program context.

The predicate *wellFormedMethod* (see Code 3.12) checks if a specific method, passed as parameter, is well–formed considering a program context. In our OO model, a method is well–formed if all of its statements comprising its body are well–formed (line 4), which is guaranteed by the predicate *wellFormedStatement* (see line 5, Code 3.12), that in turn is described in Code 3.13.

Code 3.11 Representation of the *superClassIsDeclared* predicate

```
1 pred superClassIsDeclared[p:Program,c:Class] {
2     c.extend in (p.classDeclarations).univ
3 }
```

Code 3.12 Representation of the *wellFormedMethod* predicate

```
1 pred wellFormedMethod[p:Program,class:Class,m:Method]{
2     let body = (m.body).elems
3     {
4     all stm: Statement | stm in body =>
5     wellFormedStatement[p, class, stm, m]
6     }
```

```
7 }
```

As shown in our OO metamodel (see Figure 9), a statement can be an *Assignment*, a *MethodInvocation* or a *FieldAccess*. For each of these possibilities, there is a corresponding *wellFormed* predicate (lines 2 to 4, Code 3.13). Following the same reasoning described in the other *wellFormed* predicates, an *Assignment* is well–formed (Code 3.14) when the elements that represent each of its relations are also well–formed. That is, when its left (line 2) and right–hand side (line 4 to 10) expressions are well–formed. The former is represented by a *FieldAccess* while the latter can be a *MethodInvocation* (line 6) or another *FieldAccess* (line 9). The predicates *wellFormedFieldAccess* and *wellFormedMethodInvocation* are further described. As already discussed at the end of Section 3.1.1, another *Assignment* was discarded to represent an *AssignmentExpression* right–hand side expression for simplification purposes, essentially because of the restriction of the Alloy Analyzer in doing recursive calls.

Code 3.13 Representation of the *wellFormedStatement* predicate

```
1 pred wellFormedStatement[p:Program,class:Class,st:Statement,m:
     Method]{
2     st in AssignmentExpression => wellFormedAssignment[p,class,
        st,m]
3     st in MethodInvocation => wellFormedMethodInvocation[p,
        class,st,m]
4     st in FieldAccess => wellFormedFieldAccess[p,class,st,m]
5 }
```

In order to guarantee the well–formedness of a *FieldAccess*, it is necessary to do some type checking. For doing this, the first thing is retrieving its *pExp* relation (type *PrimaryExpression*, retrieved in line 3, Code 3.16) and the formal parameter of the method whose body contains this *FieldAccess* (formal parameter is also retrieved in line 3). The only checking done by the *wellFormedPrimaryExpression* (line 5, Code 3.16 and described in Code 3.17) is verifying if the *cf* relation, in the case of an expression of type *newCreator* (which is a subsignature of the *PrimaryExpression* type), which in turn represents the identifier of the class (type *ClassId*, see Figure 9), is in the set of class identifiers (*ClassId* elements) in the mapping relation *classIdentifiers* of the *Program* type. The other possibilities for a *pExp* relation in a *FieldAccess* type are the expressions *this* and *super* (the other subsignatures of *PrimaryExpression* type) which are considered well–formed by default. Afterwards, the type checking starts. In line 7, Code 3.16, it is ensured that if the formal parameter is a primitive type (so it is not from the *ClassType* type), and the target of the *FieldAccess* being analysed is represented by an *ExpressionName* (which is a subsignature of *PrimaryExpression*), then the identifier of the field (type *FieldId*) contained in its *name* relation cannot be the same as the one in the *varName* relation of the formal parameter since its type is primitive (line 8). Our model does not allow a variable to be declared in a method body and the only possibility for a *name* relation (from an *ExpressionName* type) is the identifier of the field contained in the method formal parameter, if it

exists, or the identifier of a field declared in the class where the method is located.

Code 3.14 Representation of the *wellFormedAssignment* predicate

```
pred wellFormedAssignment[p:Program,class:Class,stm:
   AssignmentExpression,m:Method]{
    wellFormedLeftHandSide[p,class,stm.pExpressionLeft,m]

    let rightExp = stm.pExpressionRight
    {
    rightExp in MethodInvocation =>
    wellFormedMethodInvocation[p,class,rightExp,m]

    rightExp in FieldAccess =>
    wellFormedFieldAccess[p,class,rightExp,m]
    }
}
```

Code 3.15 Representation of the *wellFormedLeftHandSide* predicate

```
pred wellFormedLeftHandSide[p:Program,class:Class,stm:
   FieldAccess,m:Method] {

    stm in FieldAccess => wellFormedFieldAccess[p,class,stm,m]
}
```

Code 3.16 Representation of the *wellFormedFieldAccess* predicate

```
pred wellFormedFieldAccess[p:Program,class:Class,stm:
   FieldAccess,m:Method]{

    let target = stm.pExp, formal = m.param
    {
    wellFormedPrimaryExpression[p,class,target]

    (formal.type !in ClassType && target in ExpressionName) =>
    formal.varName != target.name

    target in newCreator =>
    fieldMatchesAndIsNotPrivate[p,stm.id_fieldInvoked,(target.
       cf).(p.classDeclarations)]

    target in this_ => (fieldIsInTheClass[p,stm.id_fieldInvoked,
       class] ||
    (!fieldIsInTheClass[p,stm.id_fieldInvoked,class] &&
    fieldIsInTheHierarchyAndIsNotPrivate[p,stm.id_fieldInvoked,
       class]))

```

```
17      target in super =>
18      fieldIsInTheHierarchyAndIsNotPrivate[p,stm.id_fieldInvoked,
           class]
19      }
20  }
```

Line 10, Code 3.16, state that, in all cases where the access to the field (of the *FieldAccess* being analysed) is through an expression of type *newCreator*, then the predicate *fieldMatchesAndIsNotPrivate* needs to hold (line 11). This predicate is detailed in Code 3.18 and checks if a field—whose value in its *id* relation matches the *FieldId* value for the *fieldInvoked* relation of the *FieldAccess* being analysed (line 3)—exists in some class in the parent level of class *c*, including its own (this is represented by the expression *c.*(extend.(p.classDeclarations))*, see line 2). In addition, this field must not be *private* (line 4). Observe that the class *c* is the one associated to the expression (of type *newCreator*) of the *FieldAccess* (see line 11, Code 3.16, expression *(target.cf).(p.classDeclarations)*) and is passed as parameter to the predicate *fieldMatchesAndIsNotPrivate*. For simplification purposes, we assume that all classes are in the same package since there is no *Package* element in our model.

On the other hand, when the access to the field is through an expression of type *this* (lines 13 to 15, Code 3.16), the associated field must lie in the class itself (call to the predicate *fieldIsInTheClass*) or, when this is not the case, the field must lie in any class in the parent class level. This is represented by the predicate *fieldIsInTheHierarchyAndIsNotPrivate* that is similar to the predicate *fieldMatchesAndIsNotPrivate*, just explained, with the difference that the class itself is not included in the parent hierarchical level – this condition was already verified in line 14. In addition, when the access is through an expression of type *super* (line 17), the field must lie in classes at the parent level of class *c* and must not be *private*. This is represented by the predicate *fieldIsInTheHierarchyAndIsNotPrivate* in line 18.

Code 3.17 Representation of the *wellFormedPrimaryExpression* predicate

```
1 pred wellFormedPrimaryExpression[p:Program,c:Class,stm:
     PrimaryExpression]{
2     stm in newCreator => classIsDeclared[p,stm.cf]
3 }
```

Code 3.18 Representation of the *fieldMatchesAndIsNotPrivate* predicate

```
1 pred fieldMatchesAndIsNotPrivate[p:Program,fId:FieldId,c:Class
     ]{
2     some f:Field | f in c.*(extend.(p.classDeclarations)).fields
          &&
3     f.id = fId &&
4     fieldIsNotPrivate[f]
5 }
```

The predicate *wellFormedMethodInvocation* follows the same reasoning (Code 3.19) as the predicate *wellFormedFieldAccess* but for methods and method invocations. However, it is slightly different since a method invocation can have real parameters (only one in case of our OO model) whilst field invocations do not. Thus, additional checkings should be done. This is represented by the predicate *wellFormedRealParameter* expanded in lines 24, 34, 42, 48 and 56, Code 3.19 and shown in Code 3.20. In addition, lines 10 and 11 ensure that, if the formal parameter of the method is of type *long* (1), then it should not be the target of a method invocation (line 12)—this would cause a compilation error since a method can not be invoked from a primitive type. On the other hand, when the type of the formal parameter is not primitive (line 14) and coincides to be the variable in the target of a method invocation (2) (see line 15), then a type checking is done, similarly as done in cases where the target is a expression of types *newCreator*, *this* or *super* (see codes from lines 26 to 34, 36 to 48 and 50 to 56, respectively, in Code 3.19).

Code 3.19 Representation of the *wellFormedMethodInvocation* predicate

```
1  pred wellFormedMethodInvocation[p:Program,class:Class,stm:
       MethodInvocation,m:Method]{
2
3      let
4        target = stm.pExp,
5        formal = m.param,
6        formalCId = formal.type.classIdentifier {
7
8        stm.pExp in PrimaryExpression =>
           wellFormedPrimaryExpression[p, class, stm.pExp]
9
10       (#formal > 0 && formal.type in Long_ &&
11        target in ExpressionName) =>
12       formal.varName != target.name
13
14       (#formal > 0 && formal.type in ClassType &&
15        target in ExpressionName && formal.varName = target.name
16       ) =>
17       some m': Method | m' in formalCId.(p.classDeclarations)
18       .*(extend.(p.classDeclarations)).methods &&
19       m'.id = stm.id_methodInvoked &&
20       #(stm.realParam) = #(m'.param) &&
21       ((m' in formalCId.(p.classDeclarations).methods &&
22        formalCId.(p.classDeclarations) != class) =>
23        m'.acc !in private_) &&
24        wellFormedRealParam[p,stm,class,m']
25
26       target in newCreator => (some m': Method | m' in
27       (target.id_cf).(p.classDeclarations).*(extend.(p.
```

```
          classDeclarations)).methods &&
28      m'.id = stm.id_methodInvoked &&
29      #(stm.realParam) = #(m'.param) &&
30      ((m' in (target.id_cf).(p.classDeclarations).methods
31        &&
32       (target.id_cf).(p.classDeclarations) != class ) =>
33       m'.acc !in private_) &&
34      wellFormedRealParam[p, stm, class, m']
35
36      target in this_ =>
37      ((some m': Method | m' in
38        class.^(extend.(p.classDeclarations)).methods &&
39        m'.id = stm.id_methodInvoked &&
40        #(stm.realParam) = #(m'.param) &&
41        m'.acc !in private_ &&
42        wellFormedRealParam[p, stm, class, m'])
43        ||
44        (some m': Method |
45        m' in class.methods &&
46        m'.id = stm.id_methodInvoked &&
47        #(stm.realParam) = #(m'.param) &&
48        wellFormedRealParam[p, stm, class, m']))
49
50      target in super =>
51      some m': Method | m' in
52      class.^(extend.(p.classDeclarations)).methods &&
53      m'.id = stm.id_methodInvoked &&
54      #(stm.realParam) = #(m'.param) &&
55      m'.acc !in private_ &&
56      wellFormedRealParam[p,stm,class,m]
57  }
```

In (2) and also when the target of a method invocation is of type *newCreator*, then it is necessary to check if there is a method in the class (or in any class in its hierarchy) corresponding to the target and to the corresponding relation of the method invocation (lines 17 to 19 and 26 to 28, respectively). This method should have the same identifier (lines 19 and 28, respectively) and number of parameters (lines 20 and 29, respectively) as the one in method invocation. The specific type of the parameter is not checked since we assume all of them have the same type: *long*. For simplification purposes, we assume that the only possible type for a real parameter is *FieldAccess*. As a method parameter is always of type *long* and likewise the type of a field, then type checking errors are avoided. In addition, the method should not be private (lines 23 and 33, respectively) when the method being called is in the set of the methods of the class in the formal parameter (lines 18 and 21), or of the class represented by the expression *newCreator*) (lines 27 and 30) and this class is not the one where the method invocation is (lines 22 and 32,

respectively).

Code 3.20 Representation of the predicate *wellFormedRealParam*

```
1  pred wellFormedRealParam[p:Program, stm:MethodInvocation, class
       :Class, m:Method]{
2      #(stm.realParam) = #(m.param)
3      #(stm.realParam) = 1 => some f:Field |
4      wellFormedFieldAccess[p, class, stm.realParam] &&
5
6      f.id = stm.realParam.id_fieldInvoked &&
7      (m.param.type in Long_ => f.type in Long_) &&
8      (m.param.type in ClassType =>
9       (f.type in ClassType &&
10       firstIsSubtypeOfTheSecondOneClass[p, f.type.
             classIdentifier,m.param.type.classIdentifier]))
11
12 }
```

The type checking when the target is *this* is almost similar but there are some differences. Firstly, the checking for the method in class (line 45) or in its hierarchy (line 38) was divided. As explained earlier, in both cases the method identifier should be the same as the one in the method invocation expression (lines 39 and 53) and the number of formal and real parameters should be the same (lines 36 and 31). In addition, the real parameter should be well–formed (lines 37 and 32). The difference is that if the method is in the class, it can be private as well. Finally, when the target is *super*, the same checkings when the target is *this* are done, except for in this case the method is not in the class where the method invocation is but in its hierarchy, are done.

It is important to say that our OO metamodel was useful to validate not only Java transformation specifications, but specifications in other languages (i.e. rCOS, ROOL, an object–oriented language with reference semantics presented in (PALMA, 2015)) that use common OO features defined in our metamodel. Even considering a subset of the language and some simplifications, we observe that all starting–hand side programs (where the predicate *wellFormedProgram* is applied to) used in the transformations analysed are 100% compilable, different from the model presented in (SOARES, 2015), which generates only 68,8% compilable programs. However, our OO model consider different elements of the JLS. For instance, they consider packages whilst we do not. The elements we defined in our OO model follows the JLS BNF syntax, whilst they do not. In addition, they argue that less rules are incorporated in their model not to inhibit the generation of interesting programs, even at risk generating non–compilable programs. This decision did not compromise the goal of their work because the non–compilable programs generated are discarded and only the compilable ones are submitted to the engine implementations. On the other hand, the OO model in our work has a different goal, as already described, beyond generating input programs to engine implementations.

3.2 Transformation–Specific Models

A second model, for each transformation being analysed, is also specified in Alloy. Each one uses the metamodel described in the previous section to represent the elements, such as classes and conditions, involved before and after the transformation. Usually, a transformation–specific model is comprised by one or two main predicates, depending on whether the specification refers to a uni– or bidirectional transformation, respectively. For instance, specifications based on algebraic laws or refactoring rules in (CORNELIO, 2004) refer to bidirectional transformations whereas the ones based on rules in rCOS have only one direction, since a refinement is specified (and as a consequence, there is just one main predicate). Each of these predicates represents the changes provoked by the transformation in the corresponding programs (and their contexts) according to the specific direction of the transformation.

Code 3.21 depicts an abstract predicate, for a specific direction of a transformation. The goal is illustrating how this kind of predicate is structured. We follow the same structural division (or pattern) along with the main predicates for the other transformation specifications, as can be seen in the following sections. Such predicates are used by our Validators, which are described in detail in Chapter 4.

Considering the direction from the left– to the right–hand side, the parameters SS and RS correspond to left and right contexts, respectively. As already discussed along this thesis, SS and RS states, respectively, for the starting– and the resulting–hand sides programs. The other parameters (line 3) refer to the elements involved in the transformation. Usually, these ones are explicitly highlighted in the specification. Afterwards, the mapping between the elements *ClassId* and *Class* from the SS and the RS programs are retrieved (variables *ssCds* and *rsCds*, respectively, lines 5 and 6, Code 3.21). Afterwards, classes B and C from the SS (*bss* and *css*, respectively) and RS (*brs* and *crs*, respectively) are also retrieved from the mappings (see lines 7 to 10).

We split the general predicate for a specific direction of a transformation into four main parts: firstly the SS description (line 13); secondly, the predicates that establish the provisos (from the SS to the RS) specified in the transformation (lines 16 and 19); thirdly, the RS description; and finally the equivalence between the SS and the RS is stated (lines 28 to 36). The SS and RS descriptions stands for the relationships among the elements before and after the transformation, respectively. For instance, as illustrated as SS description (line 13), *B* is the *C* super class, or better, *B* is in the *css´*s *extend* relation.

Code 3.21 Predicate in Alloy for illustrating the general template for specifying a transformation, considering a specific direction

```
1 module transformationSpecificModel
2
3 pred predicateForASpecificDirection[b,c:ClassId, element:Field,
      ss,rs: Program] {
4
```

```
5     let ssCds = ss.classDeclarations,
6         rsCds= rs.classDeclarations,
7         bss = b.ssCds,
8         brs = b.rsCds,
9         css = c.ssCds,
10        crs = c.rsCds {
11
12        // ss description
13       css.extend = b
14
15        // proviso (1)
16        proviso1[f, crs]
17
18
19        // proviso (2)
20        proviso2[b, c, f, rs]
21
22
23       //rs description
24       crs.extend = b
25       ...
26       ...
27
28       //equivalence between ss and rs-hand sides
29       ssCds = rsCds ++ {b -> bss} ++ {c -> css}
30
31       css.fields = crs.fields + f
32       bss.fields = brs.fields - f
33       bss.methods = brs.methods
34       css.methods = crs.methods
35
36       bss.extend = brs.extend
37       c.~((ss.classDeclarations).extend) = c.~((rs.
            classDeclarations).extend)
38       b.~((ss.classDeclarations).extend) = b.~((rs.
            classDeclarations).extend)
39 }
```

Notice that the equivalence between the SS and RS classes is captured in line 29, Code 3.21, through the equivalence of the respective mappings, except for the classes involved in the transformation (in this case, classes B and C). This occurs because these classes can contain different sets of fields or methods—depending on the particular transformation. In the case of the transformation illustrated in Code 3.22 (Law 1, from the right– to the left–hand side), the methods are the same (lines 32 and 33, Code 3.22) as they are not changed in the transformation. Nevertheless, the set of fields are not (lines 30 and 31, Code 3.22)—the difference is just the field

being moved. This behaviour was copied in Code 3.21, lines 31 to 34, just to give an example of the equivalence among methods and fields in different sides of a transformation. The equivalence of the classes is also required for their *extend* relation as a simplification (see line 36 Code 3.21, and line 35, Code 3.22). It is guaranteed that the super class of the class B, present in both sides of the transformation, is the same—in other words, the corresponding class in their *extend* relation is the same. In addition, it is established that the C's sons from both sides are the same (see lines 37 and 38, Code 3.21 and lines 36 and 37, Code 3.22).

Observe that the mapping equivalence (line 29, Code 3.21 and line 28, Code 3.22) guarantees the correspondence of the *ids* (the same *id*) of each pair of classes in the two sides. It also guarantees that each class, in a specific side (right or left), owns a different *id* from each other since a mapping structure is being used. In addition, as the context of a program (being left or right) is given as parameter, a right–hand side class is prevented from being related with a left–hand side class and vice-versa. This occurs because, given a class identifier (type *ClassId*), the corresponding class is always retrieved from the corresponding mapping (as shown in lines 7 to 10, Code 3.21) and all the object manipulation is done in this way. This optimises the Alloy model because it avoids the existence of some predicates to avoid, for instance, left classes to extend the right ones and vice-versa, among other undesirable scenarios. Actually, these classes are the same but they need to be differentiated to record the situations before and after the transformation. In the following subsections, we only detail the lines (in the main predicates of the respective transformations) that are specific for each transformation; the others are as in Code 3.21.

3.2.1 Transformation–Specific Model for Law 1

In this section, we describe the Transformation–Specific Model for Law 1, from the right–(SS) to the left–hand side (RS) direction, as our initial example. We consider it the most simple transformation we have analysed. The Alloy predicate named *law1RL*, Code 3.22, groups all the necessary steps to capture the transformation from the right– to the left–hand side described in Law 1. Complementarily to what was already discussed in Section 3.2 about the general predicate for a specific direction of a transformation, it is stated in RS description that *B* is the *C* super class (line 23) (or *B* is in the *crs extend* relation); the field *f* is in the B class in the SS (line 24) and it is public (line 25); and finally the equivalence between the left and right–hand sides is stated (lines 28 to 37). The predicates expanded to satisfy the conditions established by the provisos in Law 1 (from the right– to the left–hand side) are *fieldIsNotDeclaredInTheClass* and *forbidsAccessToFieldF* (lines 16 and 19). Besides, to ease understanding, Figure 19 indicates these predicates along with the corresponding provisos.

Code 3.22 Predicate that represents the transformation specified by Law 1

```
1 module transformationSpecificMetaModel
2
3 pred law1RL[b,c: ClassId,f:Field,right,left: Program] {
```

```
4
5       let leftCds = left.classDeclarations,
6           rightCds= right.classDeclarations,
7           bl = b.leftCds,
8           br = b.rightCds,
9           cl = c.leftCds,
10          cr = c.rightCds {
11
12          // RS description
13          cl.extend = b
14
15          // proviso (1)
16          fieldIsNotDeclaredInTheClass[f, cr]
17
18          // proviso (2)
19          forbidsAccessToFieldF[b, c, f, right: Program]
20
21
22        //SS description
23        cr.extend = b
24        f in br.fields
25        f.acc in public
26
27        // equivalence between left and right-hand sides
28        leftCds = rightCds ++ {b -> bl} ++ {c -> cl}
29
30        cl.fields = cr.fields + f
31        bl.fields = br.fields - f
32        bl.methods = br.methods
33        cl.methods = cr.methods
34
35        bl.extend = br.extend
36        c.~((left.classDeclarations).extend) = c.~((right.
              classDeclarations).extend)
37        b.~((left.classDeclarations).extend) = b.~((right.
              classDeclarations).extend)
38  }
```

With regard to the predicate *fieldIsNotDeclaredInTheClass*, that represents the proviso ← (1) of Law 1 and is shown in Code 3.23, it guarantees that there is no field in the set of fields of c right–hand side class (cr) whose id is the same as the id of the field being moved. On the other hand, the predicate *forbidsAccessToFieldF* is an embedding of the proviso (← (2)) of Law 1 and is shown in Code 3.24. It states that the access to the field being moved is forbidden from any method in any of the right–hand side classes. This is guaranteed by the predicate *accessToFieldFIsForbidden*, available in Code 3.25, which does the checking for each statement

of a specific method body (line 3, Code 3.25) where an access to the field can appear. This happens in the case where the statement is a *FieldAccess* itself (line 4), or the statement is an *AssignmentExpression*, but its left–hand side expression (relation *pExpressionLeft*) contains a *FieldAccess*. We represent this through the expression in the second parameter of the predicate *forbidsAccessToFieldFFromAFieldAccess* in line 8: *st.pExpressionLeft*.

Likewise, an access to the field can also appear in the Assignment's right–hand side expression, that is represented by the expression *st.pExpressionRight* in line 10. In addition, an access to the field can also appear in a right–hand side expression of an Assignment, as stated in line 11. In this law evaluation, we consider an OO metamodel where the *MethodInvocation* signature has no parameter. If it was the case, it would be necessary to also include the possibility for an access to the field through the parameter of a *MethodInvocation*.

The predicate *forbidsAccessToFieldFFromAFieldAccess* is used as auxiliary and guarantees that, in all cases where there is an access (represented by the *FieldAccess* type, variable *fa*) to the specific field moved (lines 3, 7 and 11, Code 3.26), and (1) whose class in its *pExp* relation is of type *newCreator* (line 3), or (2) in case the *pExp* relation points to a *this* expression (line 7), or (3) if the *pExp* relation points to a *super* expression (line 11), then this class (in case (1)), or the class where the invocation occurs (*this*, case (2)), or the *super* class referred (in case (3)) is not in the set of classes that are subtypes of B and also are not subtypes of C (lines 4 and 5, 8 and 9, 12 and 13, respectively). The predicate *firstIsSubtypeOfTheSecondOneClass*, referenced in Code 3.26 and detailed in Code 3.27, is used to check if the first ClassId passed as parameter is a subtype of the second one.

Code 3.23 Predicate that restricts a field from being declared in a specific class.

```
1 pred fieldIsNotDeclaredInTheClass[f:Field,c:Class] {
2    f.id !in c.fields.id
3 }
```

Code 3.24 Predicate that restricts a field from being declared in any subclass of the class passed as parameter.

```
1  pred forbidsAccessToFieldF[b,c:ClassId,f:Field,right:Program]{
2
3    let rightCds = right.classDeclarations {
4
5       all someClassId: ClassId,someClass: Class,m_: Method |
6       (someClassId in rightCds.univ &&
7       someClass = someClassId.rightCds &&
8       m_ in someClass.methods) => accessToFieldFIsForbidden[m_,f
           ,someClass,someClassId,b,c,right]
9    }
10 }
```

Figure 19 Law 1 with the indication of the predicates expanded in Code 3.22 corresponding to provisos, from the right– to the left–hand side direction.

Code 3.25 Predicate that forbids the access to the field being moved from B to C

```
pred accessToFieldFIsForbidden[m: Method,f: FieldId,someClass:
   Class,someClassId,b,c: ClassId,p: Program]{

    all st:Statement | st in univ.(m.body) =>
    ((st in FieldAccess =>
       forbidsAccessToFieldFFromAFieldAccess[p,st,f,someClass,
          someClassId,b,c]) &&

     (st in AssignmentExpression =>
      (forbidsAccessToFieldFFromAFieldAccess[p,st.
         pExpressionLeft,f,someClass,someClassId,b,c] &&

     (st.pExpressionRight in FieldAccess =>
      forbidsAccessToFieldFFromAFieldAccess[p,st.
         pExpressionRight,f,someClass,someClassId,b,c])
      )
     )
    )
}
```

Code 3.26 Auxiliary predicate to restrict the access of the field moved.

```
pred forbidsAccessToFieldFFromAFieldAccess[p: Program,fa:
   FieldAccess,f: FieldId,someClass: Class,someClassId,b,c:
   ClassId]{

   (fa.id_fieldInvoked = f && fa.pExp in newCreator) =>
    !(firstIsSubtypeOfTheSecondOneClass[p, mi'.pExp.cf, b] &&
    !firstIsSubtypeOfTheSecondOneClass[p, mi'.pExp.cf, c])

   (fa.id_fieldInvoked = f && fa.pExp in this_) =>
    !(firstIsSubtypeOfTheSecondOneClass[p, someClassId, b] &&
    !firstIsSubtypeOfTheSecondOneClass[p, someClassId, c])

   (fa.id_fieldInvoked = f && fa.pExp in super) =>
    !(firstIsSubtypeOfTheSecondOneClass[p,someClass.extend,b]
       &&
    !firstIsSubtypeOfTheSecondOneClass[p,someClass.extend,c])
}
```

Code 3.27 Predicate that indicates if the first ClassId parameter is subtype of the second one.

```
pred firstIsSubtypeOfTheSecondOneClass[p: Program,first,second:
    ClassId] {
  let secondSubClasses=second.*~((p.classDeclarations).extend)
  {
```

```
4     first in secondSubClasses
5   }
6 }
```

3.2.2 Transformation–Specific Model for Law 2

The transformation–specific Alloy model for Law 2 is defined in almost the same way as Law 1, since these laws typically follow the same template or have the same elements involved in the transformation—the only difference is that now a method is being moved, instead of a field. Code 3.28 shows the transformation from the right– to the left–hand side, which is equivalent to the *push down method* refactoring. To ease understanding, Figure 20 shows Law 2 with the indication of the main elements involved in the transformation as well as the predicates that guarantee the provisos specific for the transformation direction. With regard to the main elements involved in the transformation, there is the method *m*, that references the method being moved. In some other transformations, when changes or substitutions occur in the method body, indicated by the transformation specification, there are the *mR* and *mL* methods which actually represent the same method, standing, respectively, for the occurrence before and after the transformation. In addition, the set of methods in the B left–hand side class is represented by the expression in Alloy *bl.methods*, the ones in B right–hand side, by *br.methods*, and so forth. The predicates and the main elements can also be seen in Code 3.28.

Code 3.28 Predicate that captures the transformation from the right– to the left–hand side in Law 2.

```
1  pred law2RL[b,c: ClassId,m: Method,right,left: Program] {
2     let leftCds = left.classDeclarations,
3         rightCds= right.classDeclarations,
4         bl = b.leftCds,
5         br = b.rightCds,
6         cl = c.leftCds,
7         cr = c.rightCds {
8
9         // SS description: right
10        m in (br.methods)
11        cr.extend = b
12
13        // proviso (<->) (1)
14        noSuperOrPrivateAttributesInM[right,m,b]
15
16        //proviso <-> (2)
17        mIsNotDeclaredInAnySubclassOfTheClassInParam[b,m,left,
             right]
18
19        // proviso (<->) (3)
20        methodIsNotPrivate[m]
```

Figure 20 Law 2 with the indication of the main elements and the predicates (in Code 3.28) corresponding to provisos, from the right to the left–hand side direction.

```
21
22        // proviso (<-) (1)
23        methodIsNotDeclaredInTheClass[m,cr]
24
25        // provisos (<-) (2)
26        forbidsAccessToMethodM[b,c,m,right]
27
28        // RS description: left
29        cl.extend = b
30        m in (cl.methods)
31
32
33        // equivalence between left and right-hand sides
34        leftCds = rightCds ++ {b -> bl} ++ {c -> cl}
35
36        cl.fields = cr.fields
37        bl.fields = br.fields
38        bl.methods = br.methods - m
39        cl.methods - m = cr.methods
40
41        bl.extend = br.extend
42        c.~((left.classDeclarations).extend) = c.~((right.
             classDeclarations).extend)
43        b.~((left.classDeclarations).extend) = b.~((right.
             classDeclarations).extend)
```

The most significant difference between the specifications of Law 1 and Law 2 is the predicate *noSuperOrPrivateAttributesInM*—line 14, Code 3.28 (an embedding of the proviso ($(\leftrightarrow)$ (1)) in Law 2). The predicate *noSuperOrPrivateAttributesInM* (Code 3.29) guarantees that no super or private attributes appear in the body of the method being moved. An attribute in this case can refer to both fields or methods. The only way for this fact to happen in our model is through a *FieldAccess* (line 3), an *AssignmentExpression* (line 6) or a *MethodInvocation* (line 9). For the first case, the *noSuperOrPrivateAttrInLeftHandSide* predicate (line 4) should hold. This predicate is shown in Code 3.30 and states that, for all *FieldAccess* inside the body of the method being moved (it is passed as the second parameter of the predicate), there is no *pExp* relation of that *FieldAccess* in the set of *super* instances (expression *lhs.pExp !in super* (line 2)). In addition, the field represented by the relation *fieldInvoked* of the *FieldAccess* should not have the *private* accessibility, guaranteed by the predicate *fieldIsNotPrivate* (line 3).

Code 3.29 Predicate that captures no access to super or private attributes.

```
1 pred noSuperOrPrivateAttributesInM[p: Program,m: Method,class:
     ClassId]{
2     all st:Statement | st in univ.(m.body) =>
3         (st in FieldAccess =>
```

```
4         noSuperOrPrivateAttrInLeftHandSide[p,st,class]) &&
5
6        (st in AssignmentExpression =>
7         noSuperOrPrivateAttrInAssignment[p,st,m,class]) &&
8
9        (st in MethodInvocation =>
10        noSuperOrPrivateAttrInMethodInvocation[p,st,m,class])
11 }
```

Code 3.30 Predicate *noSuperOrPrivateAttrInLeftHandSide*.

```
1 pred noSuperOrPrivateAttrInLeftHandSide[p: Program,lhs:
     LeftHandSide,class: ClassId]{
2    lhs in FieldAccess => lhs.pExp !in super &&
3    fieldIsNotPrivate[p,lhs.id_fieldInvoked,class]
4 }
```

For the second case, when the statement in the body of the method being moved is an *AssignmentExpression* (line 6, Code 3.29), the predicate *noSuperOrPrivateAttrInAssignment* (Code 3.31) should hold. In this predicate, since the left–hand side of an *AssignmentExpression* (represented by the relation *pExpressionLeft* in this Alloy signature) is always a *FieldAccess* in our OO model, the predicate *noSuperOrPrivateAttrInLeftHandSide*, just explained, is applied to the *pExpressionLeft* of an *AssignmentExpression*, which appears in line 2, Code 3.31. See also Figure 9 to remember these relationships. Yet regarding the *AssignmentExpression* (Code 3.31), it is also necessary to avoid the super or private access on its right–hand side expression (represented by the relation *pExpressionRight*, accessed in lines 4 and 7, Code 3.31), which can be a *MethodInvocation* or another *FieldAccess*. In the latter case, the predicate *noSuperOrPrivateAttrInLeftHandSide* (already explained) must hold (line 8). In case the relation *pExpressionRight* is represented by a *MethodInvocation*, the predicate *noSuperOrPrivateAttrInMethodInvocation* (Code 3.32) should hold (line 5).

Code 3.31 Predicate *noSuperOrPrivateAttrInAssignment*.

```
1 pred noSuperOrPrivateAttrInAssignment[p: Program,ae:
     AssignmentExpression,m: Method,class: ClassId]{
2    noSuperOrPrivateAttrInLeftHandSide[p, ae.pExpressionLeft,
        class]
3
4    ae.pExpressionRight in MethodInvocation =>
5    noSuperOrPrivateAttrInMethodInvocation[p,ae.
        pExpressionRight,m,class]
6
7    ae.pExpressionRight in FieldAccess =>
8    noSuperOrPrivateAttrInLeftHandSide[p,ae.pExpressionRight,m,
        class]
9 }
```

Likewise the restriction in FieldAccess (lines 2 and 3, Code 3.30), it is necessary to guarantee that the *pExp* relation of the *MethodInvocation* is not in the set of *super* instances (expression *mi.pExp !in super*, line 3, Code 3.32) and that the method represented by the relation *methodInvoked* of the *MethodInvocation* does not have the *private* accessibility (call to the predicate *methodIsNotPrivate* in line 3, Code 3.32). In addition, if there is a real parameter in *MethodInvocation* (expression *#(mi.realParam) > 0* in line 4), and our OO model limits that a real parameter of a *MethodInvocation* can be only a *FieldAccess*, then the application of the predicate *noSuperOrPrivateAttrInLeftHandSide* must hold. Finally, for the case a *MethodInvocation* statement is inside the body of the method being moved (line 9, Code 3.29), the predicate *noSuperOrPrivateAttrInMethodInvocation* (Code 3.32), just described, is expanded.

Code 3.32 Predicate *noSuperOrPrivateAttrInMethodInvocation*.

```
1 pred noSuperOrPrivateAttrInMethodInvocation[p: Program,mi:
     MethodInvocation,m: Method,class: ClassId]{
2
3     mi.pExp !in super && methodIsNotPrivate[p,mi.
        id_methodInvoked,class]
4     #(mi.realParam) > 0 => noSuperOrPrivateAttrInLeftHandSide[p,
        mi.realParam,class]
5 }
```

Lines 17 and 20 from Code 3.28 are an embedding of the provisos ($(\leftrightarrow)$ (2) and (3)), respectively, in Law 2. In the first case, predicate *mIsNotDeclaredInAnySubclassOfTheClassInParam* (Code 3.33) states that, for all methods with the same signature of the method being moved, *mR*, inside all classes on the right–hand side that are also B subtypes, then these methods are not in the set of methods of these classes (line 9). In the second case, predicate *methodIsNotPrivate* (Code 3.34) establishes that the method being moved cannot be *private*.

Code 3.33 Predicate that restricts a method from being declared in any subclass of the class passed as parameter.

```
1 pred mIsNotDeclaredInAnySubclassOfTheClassInParam[b: ClassId,mR
     : Method,left,right: Program] {
2
3     let leftCds = left.classDeclarations,
4         rightCds= right.classDeclarations {
5
6     all someClass:{ClassId-b}, m:Method |
7     (sameSignature[m,mR] && someClass in rightCds.univ &&
8     firstIsSubtypeOfTheSecondOneClass[right,someClass,b]) =>
9     m !in someClass.rightCds.methods
10    }
```

Code 3.34 Predicate that restricts a method from being private.

```
1 pred methodIsNotPrivate[meth: Method] {
2
3     mR.acc !in private_
4 }
```

The remainder predicates in Code 3.28 are similar to the ones already explained in the predicate *law1RL*, Code 3.22. For instance, the predicate *methodIsNotDeclaredInTheClass* (expanded in line 23, Code 3.28 and depicted in Code 3.35) is equivalent to the predicate *fieldIsNotDeclaredInTheClass* (see line 16, Code 3.22).

In addition, predicate *forbidsAccessToMethodM* (expanded in line 26, Code 3.28 and depicted in Code 3.36) is similar to the line 19, Code 3.22 and shown in Code 3.24. The only difference in both cases is that the restriction is to a method instead of a field, as can be seen in Codes 3.35 and 3.36.

Code 3.35 Predicate that restricts a method from being declared in the set of methods of a specific class, passed as.

```
1 pred methodIsNotDeclaredInTheClass[mR: Method,cr: Class] {
2
3     no m':Method | m' in (cr.methods) && m'.id = mR.id &&
4     #(m'.param) = #(mR.param)
5 }
```

Code 3.36 Predicate that forbids the access from the classes of a specific context (left or right) to a method passed as parameter.

```
1  pred forbidsAccessToMethodM[b,c: ClassId,mR: Method,right:
       Program] {
2
3      let rightCds= right.classDeclarations {
4
5      all someClassId: ClassId,someClass: Class,m_: Method |
6      (someClassId in rightCds.univ &&
7      someClass = someClassId.rightCds &&
8      m_ in someClass.methods) =>
9      accessToMethodMIsForbidden[m_,mR,someClass,someClassId,b,c,
           right]
10     }
11 }
```

3.2.3 Transformation–Specific Model for Push Down Refactoring

We also created a model to represent the push down refactoring. We assume the same specification in Law 2 but we include a common practice used by the developers. That is, suppose

a scenario where, inside the body of the method to be pushed down, there is an access, through the keyword *this*, to another method. After the method being pushed down, it is desirable that this access is changed to the *super* keyword, to guarantee that the same method is invoked. In this way, a behavioural problem is apparently avoided and the transformation application is not rejected, hence giving some flexibility to the refactoring application. This is not possible to be done in Law 2 because of the proviso ($\leftrightarrow$) (1), that states, as already mentioned, that the method can be pulled up or pushed down provided there is *no* access to *super* (or *private*) attributes in its body.

Considering a transformation from the right– to the left–hand side (the case being discussed, since it is a push down refactoring), this proviso does not interfere and can exist in the main predicate since *this* is the access keyword inside the method body. Thus, the only necessary change in the main predicate, Code 3.28, that represents Law 2, is the presence of the predicate *correspondenceBetweenMethods* (see Code 3.37). As the body of the method being moved changes, it is necessary to have two different methods in Alloy—one to represent the method before and the other to represent the method after the transformarion, as mentioned in the first paragraph of the Section 3.2.2.

Code 3.37 Predicate that does the correspondence between the methods before *mR* and after *mL* the transformation.

```
1 pred correspondenceBetweenMethods[mR,mL:Method]{
2     mR.id = mL.id
3     (mL.param) = (mR.param)
4     (mL.return) = (mR.return)
5     mR.acc = mL.acc
6     correspondentMethodBodies[mR,mL]
7 }
```

The predicate *correspondentMethodBodies* is depicted in Code 3.38. Firstly, it establishes that the number of statements in each method body must be the same (line 2). Besides, for each corresponding statement (the ones having the same index, lines 7 and 8), predicates *correspondingAssignment* (line 11), *correspondingMethodInvocation* (line 14) and *correspondingFieldAccess* (line 17) are used to do the correspondence for all possible statements in our OO model, respectively, *AssignmentExpression*, *MethodInvocation* or *FieldAccess*.

Code 3.38 Predicate that does the correspondence between method bodies involved in the transformation.

```
1 pred correspondentMethodBodies[mRight,mLeft:Method]{
2     #(mRight.body) = #(mLeft.body)
3     let
4     indexes = (mRight.body).inds
5     {
6       all i: indexes |
7        let stRight = (mRight.body)[i],
8            stLeft = (mLeft.body)[i]
```

```
9           {
10          stRight in AssignmentExpression => (stLeft in
               AssignmentExpression &&
11          correspondingAssignment[stRight, stLeft])
12
13          stRight in MethodInvocation => (stLeft in
               MethodInvocation &&
14          correspondingMethodInvocation[stRight, stLeft])
15
16          stRight in FieldAccess => (stLeft in FieldAccess &&
17          correspondingFieldAccess[stRight, stLeft])
18          }
19      }
20  }
```

The predicate *correspondingAssignment*, depicted in Code 3.39, does the correspondence between the left–hand side expressions of the Assignment´s statements as well as their right ones. In the former one, the only possibility in our OO model is to be a *FieldAccess* expression so the predicate *correspondingFieldAccess*, depicted in Code 3.41, is used. For the latter, there are two possibilities in our OO model: doing the correspondence of *MethodInvocation´*s statements (line 5, Code 3.39) through the predicate *correspondingMethodInvocation* (see Code 3.40) or saying that if one is of the type *LiteralValue*, so is the other (line 7, Code 3.39).

Code 3.39 Predicate that does the correspondence between 2 *AssignmentExpression* statements.

```
1 pred correspondingAssignment[ass,ass2:AssignmentExpression]{
2     correspondingFieldAccess[ass.pExpressionLeft,ass2.
         pExpressionLeft]
3
4     ass.pExpressionRight in MethodInvocation =>
5     correspondingMethodInvocation[ass.pExpressionRight,ass2.
         pExpressionRight]
6
7     ass.pExpressionRight in LiteralValue => ass2.
         pExpressionRight in LiteralValue
8 }
```

In the predicate *correspondingMethodInvocation*, Code 3.40, the relation *pExp* of the *MethodInvocation* (corresponding to a statement of the method being moved, parameter *mi*, line 2) is checked. It must not be a *super* expression (line 6) not to cause behavioural problems (after the method is moved) and to be compliant with the proviso ($\leftrightarrow$) (1), Law 2. In addition, if it is also not a *this* expression, then the correspondent statements can be the same (line 8). On the other hand, if it is a *this* expression, then the other correspondent statement must also be a *MethodInvocation* but whose *pExp* relation is a *super* expression (line 2), according to the practice adopted by the developers in this kind of refactoring. The other relations such as

id_methodInvoked and *realParam* must be the same (lines 3 and 4).

Code 3.40 Predicate that does the correspondence between 2 *MethodInvocation* statements.

```
1 pred correspondingMethodInvocation[mi,mi2:MethodInvocation]{
2     mi.pExp in this_ => ((mi != mi2) && mi2.pExp in super &&
3     (mi.id_methodInvoked = mi2.id_methodInvoked) &&
4     mi.realParam = mi2.realParam)
5
6     mi.pExp !in super
7
8     mi.pExp !in this_ => (mi = mi2)
9 }
```

Likewise the predicate *correspondingMethodInvocation*, Code 3.40, predicate *correspondingFieldAccess* follows exactly the same reasoning as can be seen in Code 3.41. The only difference is the appearance of the relation *id_fieldInvoked* (from type *FieldAccess*) instead of the relations *id_methodInvoked* and *realParam* (from type *MethodInvocation*).

Code 3.41 Predicate that does the correspondence between 2 *FieldAccess* statements.

```
1 pred correspondingFieldAccess[ae,ae2: FieldAccess]{
2     ae.pExp in this_ => ((ae != ae2) && ae2.pExp in super &&
3     ae.id_fieldInvoked = ae2.id_fieldInvoked)
4
5     ae.pExp !in super
6
7     ae.pExp !in this_ => (ae = ae2)
8 }
```

3.2.4 Transformation–Specific Model for Pull Up/Push Down Method Rule in (CORNELIO, 2004)

The Alloy predicates of this rule in our model (Pull Up/Push Down Method Rule in (CORNELIO, 2004)) follow the same reasoning compared to the ones that represent Law 2, explained in Section 3.2.2. The only difference is that now we have two *A* subclasses instead of only one (see Law 2). However, as long as the structural template changed, even if slightly (from the one in Law 2), we have to adapt all these lines accordingly.

Code 3.42 Predicate that captures the transformation from the left– to the right–hand side direction in Pull Up/Push Down Method Rule in (CORNELIO, 2004) (see Section 2.1.3).

```
1 pred rule44LR[a,b,c: ClassId,m: Method,left,right: Program]{
2     a != b
3     b != c
4
5      let leftCds = left.classDeclarations,
```

```
6         rightCds= right.classDeclarations,
7         al = a.leftCds,
8         ar = a.rightCds,
9         bl = b.leftCds,
10        br = b.rightCds,
11        cl = c.leftCds,
12        cr = c.rightCds {
13
14        // RS description
15        m in (ar.methods)
16        br.extend = a
17        cr.extend = a
18
19        // proviso (<->) (1)
20        noSuperOrPrivateAttributesInM[right,mR,b]
21
22        // proviso (->) (1)
23        methodIsNotDeclaredInTheClass[m,ar]
24
25        // proviso (->) (2)
26        mIsNotDeclaredInAnySuperClassOfTheClassInParam[a,m,
             right]
27
28        // SS description
29        bl.extend = a
30        cl.extend = a
31        m in (bl.methods)
32        m in (cl.methods)
33
34
35      // equivalence between left and right-hand sides
36      al.extend = ar.extend
37      leftCds = rightCds ++ {a->al} ++ {b->bl} ++ {c->cl}
38
39      al.fields = ar.fields
40      bl.fields = br.fields
41      cl.fields = cr.fields
42
43      al.methods = ar.methods - m
44      bl.methods - m = br.methods
45      cl.methods - m = cr.methods
46
47      a.~((left.classDeclarations).extend) = a.~((right.
           classDeclarations).extend)
48      b.~((left.classDeclarations).extend) = b.~((right.
           classDeclarations).extend)
```

```
49          c.~((left.classDeclarations).extend) = c.~((right.
               classDeclarations).extend)
50      }
51  }
```

In addition, the predicate *noSuperOrPrivateAttributesInM*, already explained in Section 3.2.2, and present in line 20, Code 3.42, is also applied as an embedding of the proviso ($\leftrightarrow$) (1), also in Law 2). Besides, the predicate *methodIsNotDeclaredInTheClass*, also explained earlier, needs to be applied for class *ar* (see line 23), as stated in proviso ($\rightarrow$) (1).

The predicate *mIsNotDeclaredInAnySuperClassOfTheClassInParam*, depicted in Code 3.43, is the only one not explained so far in any of the specifications presented. It is an embedding of the proviso ($\rightarrow$) (2) and states that the method *m* is not declared in any superclass of the class passed as parameter (in this case, any *A* superclass). Although not mentioned in Pull Up/Push Down Method Rule in (CORNELIO, 2004) and to do an adaptation for Java language (since the language used in (CORNELIO, 2004) generates a compilation error if the method declared in hierarchical level has same identifier but different parameters), we assume that not only method *m* must not exist in *A* superclasses but also any other method with the same method *m* signature (same identifier and formal parameters—line 6, Code 3.43).

Code 3.43 Predicate that restricts a method from being declared in any super class of the class passed as parameter.

```
1  pred mIsNotDeclaredInAnySuperClassOfTheClassInParam[m: ClassId,
      methMoved: Method,p: Program] {
2
3      let pCds = p.classDeclarations {
4
5      all someClass:{ClassId-m}, meth: Method |
6      (sameSignature[meth,methMoved] &&
7      someClass in pCds.univ &&
8      firstIsSubtypeOfTheSecondOneClass[p,m,someClass]) =>
9      methMoved !in someClass.pCds.methods
10     }
11 }
```

With regard to the transformation from the right– to the left–hand side direction, the main Alloy predicate to represent this transformation is very similar to the one that represents Law 2 (considering the same direction), as can be seen in Code 3.28. The difference is that the requirement for the method not to be private does not exist in Pull Up/Push Down Method Rule in (CORNELIO, 2004) specification. In addition, the predicate *methodIsNotDeclaredInTheClass* needs to be applied twice for classes *br* and *cr*, lines 23 and 24, Code 3.44, as stated in proviso ($\leftarrow$) (1).

Code 3.44 Predicate that captures Pull Up/Push Down Method Rule in (CORNELIO, 2004) (see Sec-

tion 2.1.3).

```
1  pred rule44RL[a,b,c: ClassId,m: Method,left,right: Program] {
2      a != b
3      b != c
4
5      let leftCds = left.classDeclarations,
6          rightCds= right.classDeclarations,
7          al = a.leftCds,
8          ar = a.rightCds,
9          bl = b.leftCds,
10         br = b.rightCds,
11         cl = c.leftCds,
12         cr = c.rightCds {
13
14         // right description
15         m in (ar.methods)
16         br.extend = a
17         cr.extend = a
18
19         // proviso (<->) (1)
20         noSuperOrPrivateAttributesInM[right,mR,b]
21
22         // proviso (<-) (1)
23         methodIsNotDeclaredInTheClass[m,br]
24         methodIsNotDeclaredInTheClass[m,cr]
25
26         // provisos (<-) (2)(3)
27         forbidsAccessToMethodM[b,c,m,right]
28
29         //left description
30         bl.extend = a
31         cl.extend = a
32         m in (bl.methods)
33         m in (cl.methods)
34
35        // equivalence between the left- and the right-hand
             sides
36        al.extend = ar.extend
37        leftCds = rightCds ++ {a->al} ++ {b->bl} ++ {c->cl}
38
39        al.fields = ar.fields
40        bl.fields = br.fields
41        cl.fields = cr.fields
42
43        al.methods = ar.methods - m
44        bl.methods - m = br.methods
```

```
45        cl.methods - m = cr.methods
46
47        a.~((left.classDeclarations).extend) = a.~((right.
             classDeclarations).extend)
48        b.~((left.classDeclarations).extend) = b.~((right.
             classDeclarations).extend)
49        c.~((left.classDeclarations).extend) = c.~((right.
             classDeclarations).extend)
50        }
51 }
```

In addition, the predicate *forbidsAccessToMethodM* (line 27) follows the same pattern as the same predicate expanded in line 26, Code 3.28. Compare Codes 3.36 with 3.45 and the expanded predicates in each one—the only difference is the existence of the predicate in Code 3.47, that we adapt to what is required by the proviso (←) (2) and (3). The parameters *b* and *c* (type *ClassId*) represent in this case the *A* subclasses shown in Pull Up/Push Down Method Rule in (CORNELIO, 2004) specification (Section 2.1.3), different from the parameters *b* and *c* in Code 3.26, which represent the classes *B* and *C* in Law 2. This predicate, as we already described in Section 3.2.1, works similarly to the predicate *forbidAccessToMethodM*, used for Law 2. The difference is that we have to replace the *FieldAccess* for *MethodInvocation* type along with its corresponding relations.

Code 3.45 Predicate *forbidsAccessToMethodM*.

```
1 pred forbidsAccessToMethodM[b,c: ClassId,m: Method,right:
     Program] {
2
3     let rightCds= right.classDeclarations {
4
5     all someClassId: ClassId, someClass: Class, m_: Method |
6     (someClassId in rightCds.univ &&
7     someClass = someClassId.rightCds &&
8     m_ in someClass.methods) =>
9     accessToMethodMIsForbidden[m_,m,someClass,someClassId,b,c,
         right]
10    }
11 }
```

Code 3.46 Predicate *accessToMethodMIsForbidden*.

```
1 pred accessToMethodMIsForbidden[m_,m: Method,someClass: Class,
     someClassId,b,c: ClassId,p: Program]{
2
3     all st: Statement |
4     st in univ.(m_.body) =>
```

```
5     ((st in AssignmentExpression && st.pExpressionRight in
        MethodInvocation) =>
6      forbidAccessToMethodM[p,st.pExpressionRight,m,someClass,
         someClassId,b,c])
7      &&
8      (st in MethodInvocation => forbidAccessToMethodM[p,st,m,
         someClass,someClassId,b,c])
9  }
```

Besides, due to the difference between Law 2 and Rule 2.1 structural templates and in the way the provisos ($\leftarrow$) (2) and ($\leftarrow$) (2) and (3), respectively, are described in each specification, observe that there is also a slightly difference between the respective *forbidAccessToMethodM* predicates that represent them. Actually the provisos mean the same thing, but we tried to write the predicates exactly the way they are described. Thus, in Code 3.47, it is ensured that the type representing the keyword access (someClass) from the MethodInvocation *mi* is a *B* or *C* subtype (lines 6 and 7, 11 and 12, 16 and 17), exactly as described in provisos ($\leftarrow$) (2) and (3). In addition, there is a restriction to the keyword access *super* in calls to a method *m*. This is shown in line 4, Code 3.47.

Code 3.47 Predicate *forbidAccessToMethodM*.

```
1  pred forbidAccessToMethodM[p: Program,mi': MethodInvocation,m:
     Method,someClass: Class,someClassId,b,c: ClassId]{
2
3      //proviso (<-) (2)
4      mi'.id_methodInvoked = m.id && mi'.pExp in super =>
5      (someClassId != b && someClassId != c &&
6      (firstIsSubtypeOfTheSecondOneClass[p,someClass.extend,b] ||
7      firstIsSubtypeOfTheSecondOneClass[p,someClass.extend,c]))
8
9      //proviso (<-)(3)
10     mi'.id_methodInvoked = m.id && mi'.pExp in newCreator =>
11     (firstIsSubtypeOfTheSecondOneClass[p,mi'.pExp.id_cf,b] ||
12     firstIsSubtypeOfTheSecondOneClass[p,mi'.pExp.id_cf,c])
13
14     //proviso (<-)(3)
15     mi'.pExp in this_ && mi'.id_methodInvoked = m.id =>
16     (firstIsSubtypeOfTheSecondOneClass[p,someClassId,b] ||
17     firstIsSubtypeOfTheSecondOneClass[p,someClassId,c])
18 }
```

3.2.5 Transformation–Specific Model for Rule 3

In this section, we describe the transformation–specific model for Rule 3, mentioned in Section 2.1.4. Differently from the laws, rules in rCOS only have one direction, since they are a

refinement and we assume this direction is from the left– to the right–hand side, if we consider the directions adopted by algebraic laws. In Rule 3 specification, we have predicates that did not appear in the earlier specifications and this specification owns a higher degree of difficulty, compared to the others explained so far. This occurs because of the many replacements necessary to be done.

The parameters *m* and *n* refers to the ids of the *M* and *N* classes in the specification template, respectively, while *mBefore* and *mAfter* refers to the method *m* before and after the application of the transformation, respectively. Although they are the same method, they have different chronological time, and need to be differentiated in our Alloy model due to the change of its internal context as the rule determines substitutions in its body. The parameter *f* corresponds to the field *b* (of type *N*) in the template.

Code 3.48 Predicate that captures the refinement described in Rule 3.

```
1  pred rule3LR[m,n: ClassId, mBefore,mAfter: Method, f: Field,
       left,right: Program] {
2      let leftCds = left.classDeclarations,
3          rightCds= right.classDeclarations,
4          ml = m.leftCds,
5          mr = m.rightCds,
6          nl = n.leftCds,
7          nr = n.rightCds{
8
9          // SS description
10         mBefore in (ml.methods)
11         mBefore !in (mr.methods)
12         mBefore !in (nl.methods)
13         mBefore !in (nr.methods)
14
15         // premise3LR
16         f in ml.fields && f.type.classIdentifier = n
17         methodOnlyRefersToAttrOrMethodsThroughB[mBefore,f,ml,
               left]
18
19         mIsOnlyUsedLocallyInM[m,ml,left,mBefore]
20
21         // RS description
22         mAfter in (nr.methods)
23         mAfter !in (nl.methods)
24         mAfter !in (mr.methods)
25
26         nl.extend = m
27         nr.extend = m
28
29         // ops
```

```
30          // correspondenceBetween ml and mr
31          correspondenceBetweenMlAndMr[left,right,ml,mr,m,mBefore,
               mAfter,f]
32
33          // replacements inside the body
34          equalsMethodsRule3[left,n,mBefore,mAfter,f.id]
35
36          //equivalence between left and right-hand sides
37          leftCds = rightCds ++ {m -> ml} ++ {n -> nl}
38
39          nl.fields = nr.fields
40          ml.fields = mr.fields
41          nl.methods = nr.methods - mAfter
42
43          //classes equivalence
44          nl.extend = nr.extend
45          n.~((left.classDeclarations).extend) = n.~((right.
               classDeclarations).extend)
46          ml.extend = mr.extend
47          m.~((left.classDeclarations).extend) = m.~((right.
               classDeclarations).extend)
48        }
49  }
```

Concerning the equivalence of the methods *mr* and *ml*, and *nr* and *nl*, we cannot do in Rule 3 specification the same as we have done in Law 2 specification, lines 38 and 39 in Code 3.28, that is, simply establish that the set of methods in the left–hand side is the same as the ones in the right–hand side except for the method being moved, which is in the left– but not in the right–hand side. The reason is because in Rule 3 we have the *ops* substitution that says: for every access to $m()$ in M, which is the method being moved to class N, we have to change the access for $b.m()$. Thus, we cannot do an equivalence in classes that represent M such as *ml.methods - mBefore = mr.methods + mAfter*. In other words, it is necessary to be careful in the correspondence of classes *before* and *after* the transformation, since every method, where the replacement defined in *ops* is necessary, needs to change its internal body. In this case, we have different but correspondent methods (with same *id* and *parameters* but different bodies). The other methods, where the replacement is not necessary, are equals, with only a different context (left or right), which is passed as a predicate parameter. Thus, not to violate what the substitution determines, lines 10 to 13 and 22 to 24, Code 3.48, establishes where methods *mBefore* and *mAfter* exist before and after the transformation. Besides, we have the predicate *correspondenceBetweenMlAndMr* (depicted in Code 3.49 and present in line 31) to guarantee that the substitutions will be done correctly, according to the Rule 3 specification in Section 2.1.4. However, as a substitution is not defined for the class N, we can adopt the pattern followed in the earlier specifications, that is shown in line 41. The only difference for this class after the

transformation is the addition of the method *m*() in its set of methods.

Figure 21 shows a kind of call graph to ease the visualization and understanding of the predicates relationships (which ones are present in the others). The ones underlined represent functions used in the predicates. For instance, we have some auxiliary functions such as *methodsThatFitTheConditions* and *methodsThatFitTheConditionsInRight* in predicate *correspondenceBetweenMlAndMr*.

Code 3.49 Predicate that establishes the correspondence among the methods in class M before and after the transformation.

```
1  pred correspondenceBetweenMlAndMr[left,right: Program, ml,mr:
       Class, m: ClassId, mBefore,mAfter: Method, f: Field]{
2
3      all meth:{Method - mBefore} |
4      (meth in methodsThatFitTheConditions[left,mBefore,ml]) =>
5      thereIsACorrespondingMethodInMr[right,left,m,meth,mBefore,f
           ]
6
7      all meth:{Method - mBefore} |
8      (meth !in methodsThatFitTheConditions[left,mBefore,ml]) =>
           meth in mr.methods
9
10     all meth:{Method - mAfter} |
11     (meth in methodsThatFitTheConditionsInRight[right,m,mAfter,
           f]) =>
12     thereIsACorrespondingMethodInMl[left,right,m,meth,mAfter,f]
13
14     all meth:{Method - mAfter} |
15     (meth !in methodsThatFitTheConditionsInRight[right,m,mAfter,
           f]) =>
16     meth in ml.methods
17 }
```

The first function, depicted in Code 3.50, returns the set of methods in a specific class (see line 2, Code 3.50)—in this case, it is the *M* left–hand side class or simply *ml* (see third parameter in line 5, Code 3.49)—that owns a call to the method being moved (method *mBefore*). The method being moved is excluded from the set returned by the function, as can be seen in line 4, Code 3.50. This occurs due to the definition of *ops* substitution in Figure 2—observe that *ops* excludes method *m()*.

Code 3.50 Function that returns the set of methods in mClass (passed as parameter) that owns a call to the method being moved.

```
1  fun methodsThatFitTheConditions[p: Program, mBefore: Method,
       mClass: Class]: set Method {
2      {meth:Method | meth in mClass.methods &&
3       methodFitsTheConditions[p,meth,mBefore,mClass] &&
```

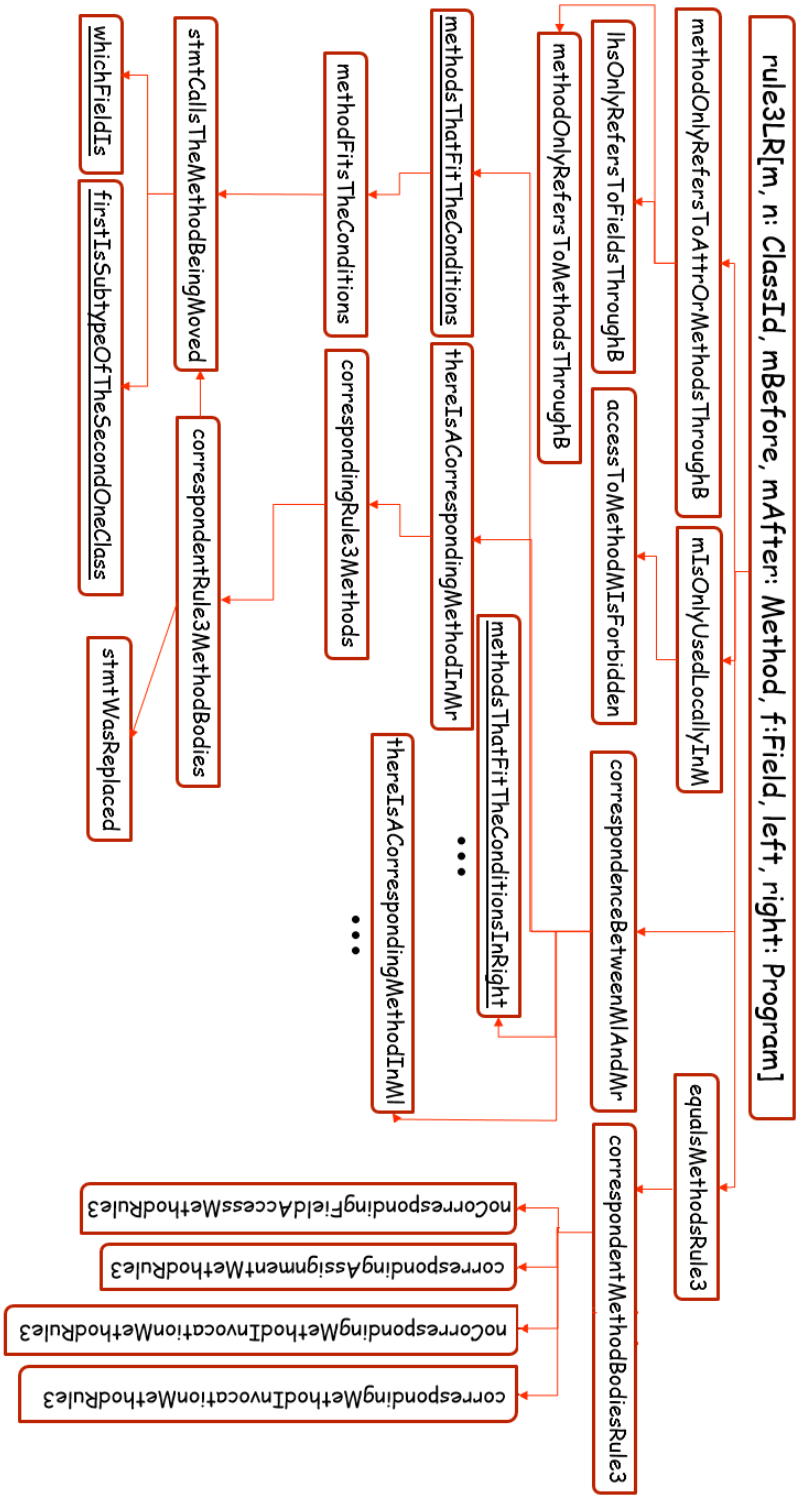


Figure 21 Call graph of the main predicates and functions used in the predicate rule3LR.

```
4      meth != mBefore}
5  }
```

The predicate *methodFitsTheConditions* in Code 3.51 is responsible for verifying if there is any statement (line 3) from the method being analysing which owns a call to the method being moved. For doing this, it uses an auxiliary predicate *stmtCallsTheMethodBeingMoved* (see lines 4 and 7), which is depicted in Code 3.52.

Code 3.51 Predicate that ensures that a method passed as parameter owns a call to the method being moved

```
1  pred methodFitsTheConditions[p: Program, m,mBefore: Method,
       mClass: Class]{
2
3      some st:Statement | st in (m.body).elems &&
4      (st in MethodInvocation => stmtCallsTheMethodBeingMoved[p,
          st,mBefore,mClass]) &&
5      (st in AssignmentExpression =>
6       (st.pExpressionRight in MethodInvocation &&
7        stmtCallsTheMethodBeingMoved[p,st.pExpressionRight,mBefore
            ,mClass]
8       )
9      )
10 }
```

There, it is checked at first if the identifier of the method being invoked is the same as the one of the method being moved (lines 13, Code 3.52). Secondly, in the case of the MethodInvocation being analysed owns a real parameter (see line 2), it is checked if the type of this real parameter is the same as the formal parameter type of the method being moved. There are two possibilities: both are primitive (line 8) or one is subtype of the other (lines 9 and 10). The real parameter (along with its type) is retrieved through the function *whichFieldIs* (line 5), depicted in Code 3.53, which returns the first field in the class hierarchy (the class is passed as parameter) that owns a specific identifier.

Code 3.52 Predicate that ensures that a MethodInvocation, passed as parameter, owns a call to the method being moved

```
1  pred stmtCallsTheMethodBeingMoved[p: Program,stmt:
       MethodInvocation,m: Method,c: Class]{
2      ((#stmt.realParam > 0) =>
3      {
4       let
5       field = whichFieldIs[p,stmt.realParam.id_fieldInvoked,c]
6       {
7           #stmt.realParam = #m.param
8           ((#stmt.realParam = 1 && field.type in Long_ && m.param.
                type in Long_ ) ||
9            (#stmt.realParam = 1 && field.type in ClassType &&
10           firstIsSubtypeOfTheSecondOneClass[p,field.type.
                classIdentifier,m.param.type.classIdentifier])
11          )
12      }
13     }) && (stmt.id_methodInvoked = m.id)
14 }
```

Code 3.53 Function that returns the first field in the class hierarchy (the class is passed as parameter) that owns a specific identifier.

```
1  fun whichFieldIs[p: Program,fId: FieldId,c: Class]: Field{
2      {f:Field | f in c.*(extend.(p.classDeclarations)).fields &&
           f.id = fId}
3  }
```

Hence, the predicate *correspondenceBetweenMlAndMr* ensures that, for every method in class *ml* that owns a call to the method being moved (line 4, Code 3.49), there is a corresponding method, in the class *mr*, that will follow the substitution rule *ops* defined in Figure 2 (see line 5, Code 3.49). This is guaranteed by the predicate *thereIsACorrespondingMethodInMr*, depicted in Code 3.54. It ensures the existence of a method in class *mr* (lines 2 and 3) correspondent to the one that calls the method being moved, according to the requirement of the *ops* substitution. This method is different from the one moved to class *N* (line 4) and it is not in class *ml* (line 5). In addition, the correspondence of the methods is done by the predicate *correspondingRule3Methods* (line 6) whose code is shown in Code 3.55.

Code 3.54 Predicate that ensures the existence of a method correspondent to the one that calls the method being moved, according to the requirement of the *ops* substitution.

```
1  pred thereIsACorrespondingMethodInMr[p,p': Program, m: ClassId,
       meth,moved: Method, f: Field]{
2      one m':Method |
3      m' in (m.(p.classDeclarations)).methods &&
4      m' != meth &&
```

```
5     m' !in (m.(p'.classDeclarations)).methods &&
6     correspondingRule3Methods[p,meth,m',moved,f.id,m.(p.
          classDeclarations)]
7  }
```

The predicate *correspondingRule3Methods* establishes that the identifiers of the correspondent methods are the same (line 2) as well as their parameters (line 3), return types (line 4) and accessibility modifiers (line 5). In addition, the correspondence of the method bodies is done by the predicate *correspondentRule3MethodBodies* (line 6) whose code is shown in Code 3.56.

Code 3.55 Predicate that do the correspondence defined in *ops* substitution.

```
1  pred correspondingRule3Methods[p: Program, mR,mL,moved: Method,
       varN: FieldId, c: Class]{
2      mR.id = mL.id
3     (mL.param) = (mR.param)
4     (mL.return) = (mR.return)
5     mR.acc = mL.acc
6     correspondentRule3MethodBodies[p,mR,mL,moved,varN,c]
7  }
```

In predicate *correspondentRule3MethodBodies* (Code 3.56), it is guaranteed that the number of statements in each body is the same (line 2). Besides, as a method body is defined as a sequence of Statement (see Section 3.1), each corresponding statement (in the left– and the right–hand side) is compared one by one, through its corresponding index, which is the same for both (lines 6 and 7). In this way, the correspondence defined in the *ops* substitution can be done with the predicates *stmtCallsTheMethodBeingMoved* and *stmtWasReplaced* as auxiliaries. The first one was already explained (see Code 3.52) whilst the other effectively do the substitution.

In this replacement definition (see the *ops* substitution), we assume that in every place where there exists a call to the method $m()$, regardless the target, we have to replace $m()$ for $b.m()$. Thus, if we have a call such as $this.m()$, we have to replace for $this.b.m()$, if the call is $newMSubtype().m()$, we have to replace for $newMSubtype().b.m()$, and so forth. If the statement of a body is a *MethodInvocation* (lines 9 and 15), so is the other statement (lines 11 and 17). In the first case (line 9), if besides being a *MethodInvocation*, it owns a call to the method being moved to class *N* (*m()* corresponds to the parameter *moved()* method in line 10), the predicate *stmtWasReplaced* would be applied (line 12) to replace the *pExp* (type *PrimaryExpression*) from the *MethodInvocation* in line 9 by another *pExp* (for *MethodInvocation* in line 11)—which would be a *FieldAccess* whose *pExp* relation is the same as the one in the *MethodInvocation* in line 9. In addition, its *id_fieldInvoked* relation is the identifier of the field *b* in class *M*.

Code 3.56 Predicate that do the correspondence, defined in *ops* substitution, between the bodies of the methods.

```
1  pred correspondentRule3MethodBodies[p: Program,mRight,mLeft,
       moved: Method,varN: FieldId,c: Class]{
```

```
2     #(mRight.body) = #(mLeft.body)
3     let indexes = (mRight.body).inds
4     {
5         all i: indexes |
6             let stRight = (mRight.body)[i],
7                 stLeft = (mLeft.body)[i]
8                 {
9                 (stRight in MethodInvocation &&
10                 stmtCallsTheMethodBeingMoved[p,stRight,moved,c])
                      =>
11                (stLeft in MethodInvocation &&
12                 stmtWasReplaced[stLeft,varN,moved] &&
13                 stRight.realParam = stLeft.realParam)
14
15                (stRight in MethodInvocation &&
16                 !stmtCallsTheMethodBeingMoved[p,stRight,moved,c])
                       =>
17                (stLeft in MethodInvocation && stLeft = stRight)
18
19                (stRight in AssignmentExpression &&
20                 stRight.pExpressionRight in MethodInvocation &&
21                 !stmtCallsTheMethodBeingMoved[p,stRight.
                      pExpressionRight,moved,c]) =>
22                (stLeft in AssignmentExpression && stLeft =
                     stRight)
23
24                (stRight in AssignmentExpression &&
25                 stRight.pExpressionRight in MethodInvocation &&
26                 stmtCallsTheMethodBeingMoved[p,stRight.
                      pExpressionRight,moved,c]) =>
27                (stLeft in AssignmentExpression &&
28                 stLeft.pExpressionRight in MethodInvocation &&
29                 stmtWasReplaced[stLeft.pExpressionRight,varN,
                      moved] &&
30                 stRight.realParam = stLeft.realParam)
31                }
32    }
33 }
```

In this way, the *pExp* relation of a *MethodInvocation* needs to be also a *FieldAccess*. Thus a *FieldAccess* type needs to be a *PrimaryExpression* subsignature, in addition to *this*, *super* and *newCreator* types. The problem is, if we do this, we generate a recursive situation in *FieldAccess* type, since this type also owns a *pExp* relation of type *PrimaryExpression*. As mentioned in Section 3.1.1, it is difficult to deal with recursive predicates in Alloy. Because of this, the predicate *stmtWasReplaced* in Code 3.57 is not correct (or complete) since it does

not consider the *PrimaryExpression* (in *pExp* relation) of the *MethodInvocation*'s in class *M*. The *pExp* relation from these statements are discarded and the new *MethodInvocation* formed (see Code 3.56, lines 12 and 29—this last one is the right expression of an Assignment) has a *pExp* relation of *ExpressionName* type (line 2, Code 3.57). Its *name* relation is the field *b* in Rule 3 (it corresponds to the field *f* in line 2). Finally, the identifier of the method in this new *MethodInvocation* is the same as the one in the earlier *MethodInvocation* which in turn is the identifier of the method moved to class *N* (line 3).

Hence, our specification has a limitation and the substitution defined in *ops* substitution in Rule 3 (represented in our model by the predicate *stmtWasReplaced*) is not done completely. The expression in the *pExp* relation (from the statement having a call to the method being moved) should be *pExp.b* (in the corresponding new statement in class *M*) instead of only *b*. In other words, the first *pExp* relation is discarded. As already explained, for being *pExp.b*, as this expression corresponds to a *FieldAccess*, it would be necessary that a *FieldAccess* was a possibility for a *PrimaryExpression* (which in turn is a type of the relation *pExp* in *MethodInvocation*). But this is not possible since this would generate a recursion scenario in the *FieldAccess* type since it also owns a *PrimaryExpression* in one of its relation. Actually, it is an Alloy limitation—its difficulty in dealing with recursive calls, as we already discussed in Section 3.1.1. Although the recursion would be caused in *FieldAccess* type, as the model is unique for all types, the execution of the substitution *ops* in Rule 3 is compromised. One can think the solution is to remove the *FieldAccess* type but it does not make sense doing this for Rule 3 since all the substitutions defined involves a field (*b*) and consequently the access to it.

Code 3.57 Predicate that do the replacement defined in *ops* substitution, in the case of a MethodInvocation.

```
1 pred stmtWasReplaced[st: MethodInvocation,f: FieldId,moved:
     Method]{
2     st.pExp in ExpressionName && st.pExp.name=f &&
3     st.id_methodInvoked = moved.id
4 }
```

As a consequence, the *correspondentRule3MethodBodies* predicate (see Code 3.56) is not completely performed, because of the part where the *ops* substitution (in Rule 3) would be performed—mainly through the expansion of the *stmtWasReplaced* predicate (lines 12 and 29). The remaining lines (lines 15 to 22) cover the cases where there is no invocation for the method being moved (to class*N*) inside the methods of class *M*. In other words, the cases not contemplated by the *ops* substitution in Rule 3.

Returning to the main predicate, named *predicateRule3LR*, in Code 3.48, line 16 establishes that the location of the field f is in class *ml* and its type is *N*—remember the field *f* is equivalent to the field *b* in Figure 2. By following our interpretation of the Rule 3, we assume that the sentence "If *c* only refers to an attribute b.x of N and a method b.n() of b for theoretical neatness" implies that there is no access from the body *c* (see Figure 2) to attributes or methods through an expression different from $b.x$ or $b.n()$, where $n()$ is a method from class *N*. Hence,

the predicate *methodOnlyRefersToAttrOrMethodsThroughB*, present in line 17, Code 3.48, and depicted in Code 3.58, represents this sentence.

Code 3.58 Predicate methodOnlyRefersToAttrOrMethodsThroughB.

```
1  pred methodOnlyRefersToAttrOrMethodsThroughB[method: Method, b:
       Field, correspondingMClass: Class, p: Program] {
2      all st:Statement |
3      st in univ.(method.body) => (st in AssignmentExpression =>
4      lhsOnlyRefersToFieldsThroughB[p,correspondingMClass,st.
         pExpressionLeft,b]) &&
5
6      (st in AssignmentExpression &&
7       st.pExpressionRight in MethodInvocation =>
8       methodOnlyRefersToMethodsThroughB[p,correspondingMClass,st.
          pExpressionRight,b]) &&
9
10     (st in MethodInvocation =>
         methodOnlyRefersToMethodsThroughB[p,correspondingMClass,
         st,b])
11 }
```

As a statement in our OO model can be an *AssignmentExpression* or a *MethodInvocation*, then the predicate in Code 3.58 ensures that, in dealing with an *AssignmentExpression* (line 3), (1) its *pExpressionLeft* (type *LeftHandSideExpression*, abbreviation *lhs*) only accesses a field in *N* through the *b* attribute (see call to the *lhsOnlyRefersToFieldsThroughB* predicate in line 4) and (2) its *pExpressionRight*, which can only be a *MethodInvocation* (line 7), only accesses a method in *N* through the *b* attribute (see call to the *methodOnlyRefersToMethodsThroughB* predicate in line 8). Finally, if the statement in the body is a *MethodInvocation* itself, then the same predicate just mentioned can be used (line 10).

The predicates *lhsOnlyRefersToFieldsThroughB* and *methodOnlyRefersToMethodsThroughB* are very simple as depicted in Code 3.59 and Code 3.58, respectively. In the former one, it is ensured that the *pExpressionLeft* of the *AssignmentExpression* should be a *FieldAccess* whose *pExp* relation is a *ExpressionName* (line 3) which in turn has a relation *name* correspondent to the *id* of the attribute *b* (line 4, Code 3.59)—remember that the type of the relation *pExpressionLeft* of an *AssignmentExpression* in our model can only be a *FieldAccess*. In the latter case, it is ensured the same for the *pExp* relation but now considering that this relation is of a *MethodInvocation* type (line 3, Code 3.60). In addition, as the real parameter of a *MethodInvocation* can be a *FieldAccess*, then in the case a real parameter exists (line 5, Code 3.60), we apply the predicate *lhsOnlyRefersToFieldsThroughB* again but now using this real parameter as parameter of the predicate (line 6).

Code 3.59 Predicate *lhsOnlyRefersToFieldsThroughB*.

```
1 pred lhsOnlyRefersToFieldsThroughB[p: Program,
     correspondingMClass: Class, lhs: LeftHandSide, b: Field] {
2
3     lhs in FieldAccess && lhs.pExp in ExpressionName &&
4     lhs.pExp.name = b.id
5 }
```

Code 3.60 Predicate *methodOnlyRefersToMethodsThroughB*.

```
1 pred methodOnlyRefersToMethodsThroughB[p: Program,
     correspondingMClass: Class, mi': MethodInvocation, b: Field]
      {
2
3     mi'.pExp in ExpressionName && mi'.pExp.name = b.id
4
5     #(mi'.realParam) = 1 =>
6     lhsOnlyRefersToFieldsThroughB[p,correspondingMClass,mi'.
        realParam,b]
7 }
```

Because of the affirmative "where m is only used locally in M" in Rule 3 specification, we have the predicate *mIsOnlyUsedLocallyInM* (see line 19, Code 3.48) whose definition is in Code 3.61. It is very similar to the predicates *forbidsAccessToFieldF* (see line 19, Code 3.22) and *forbidsAccessToMethodM* (see line 26, Code 3.28). The main difference is that in the case of the predicate *mIsOnlyUsedLocallyInM*, the restriction in the access to the method is for all the classes in the left context, except for the class *M*, whilst in the other cases the restriction is for all the classes in the correspondent context.

Code 3.61 Predicate that restricts method m() is only used in class M.

```
1 pred mIsOnlyUsedLocallyInM[m: ClassId,ml: Class,left: Program,
     mBefore: Method] {
2
3     let leftCds = left.classDeclarations
4     {
5     all someClassId:{ClassId-m},someClass:{Class-ml},m_:Method |
6     (someClassId in leftCds.univ &&
7      someClass = someClassId.leftCds &&
8      m_ in someClass.methods && m_ != mBefore) =>
9     accessToMethodMIsForbidden[m_, mBefore]
10    }
11 }
```

By following the description of the predicate *rule3LR*, in Code 3.48, lines 26 and 27 only establishes that class *M* is in the extend relation of class *N*. The predicate *equalsMethodsRule3*, line 34, performs the substitutions in the body of method $m()$ being moved, as established in the

second substitution rule in Rule 3 specification (Figure 2). It is shown in Code 3.62 and is very similar to the predicate *correspondingRule3Methods* and likewise establishes that the identifiers of the corresponding methods are the same (line 2) as well as their parameters (line 3), return types (line 4) and accessibility modifiers (line 5). In addition, the correspondence of the method bodies is done by the predicate *correspondentMethodBodiesRule3* (line 6). In this case, only the body of the method being moved is taken into account (the one from the left– and right–hand side), differently from the substitution done by the predicate *correspondentRule3MethodBodies* (Code 3.56), already discussed, which is applied for every method body where exists a call to the method being moved.

Code 3.62 Predicate that establishes the equality of the method being moved, before and after the transformation, in Rule 3 specification.

```
1 pred equalsMethodsRule3[p: Program, n: ClassId, mR,mL: Method,
     varN: FieldId]{
2      mR.id = mL.id
3     (mL.param) = (mR.param)
4     (mL.return) = (mR.return)
5     mR.acc = mL.acc
6     correspondentMethodBodiesRule3[p,n,mR,mL,varN]
7 }
```

With regard to the predicate *correspondentMethodBodiesRule3* (see Code 3.63), we assume that the expressions that will be replaced are of type *FieldAccess* and *MethodInvocation* (respectively, in the case of *b.x* and *b.n()*, see Figure 2). Thus, the replacements can be done without major problems. The statements in each method body are compared one by one likewise the *correspondentRule3MethodBodies* predicate (Code 3.56), already explained. Hence, if we have an *AssignmentExpression*, as a statement of the *m()* method body in one side, the other statement in the *m()* method body of the other side (corresponding to the same index), is also an *AssignmentExpression* (lines 11 and 15). The same occurs for the *MethodInvocation* statements (lines 19 and 23). The *noCorrespondingFieldAccessMethodRule3* and *noCorrespondingMethodInvocationMethodRule3* predicates, used as auxiliaries in the predicate *correspondentRule3MethodBodies*, ensures that the *pExp* relation of these both statements do not suit in the pattern previewed by the second substitution rule in Figure 2, as depicted in Codes 3.64 and 3.65. Hence, the negative of these predicates mean the expression suits the rule. Thus, if this is the case (lines 12 and 24), the predicates *correspondingAssignmentMethodRule3* (Code 3.66) and *correspondingMethodInvocationRule3* (Code 3.68) are used to do the correspondence, according to the substitution rule, between the correspondent *AssignmentExpression* and *MethodInvocation*, respectively.

Code 3.63 Predicate that establishes the equality of the bodies of the methods mLeft and mRight, which represent the method being moved, before and after the transformation, respectively.

```
1  pred correspondentMethodBodiesRule3[p: Program, n: ClassId,
       mRight,mLeft: Method, varN: FieldId]{
2      #(mRight.body) = #(mLeft.body)
3      let
4      indexes = (mRight.body).inds
5      {
6      all i: indexes |
7      let stRight = (mRight.body)[i],
8          stLeft = (mLeft.body)[i]
9      {
10         stRight in AssignmentExpression =>
11         stLeft in AssignmentExpression &&
12         (!noCorrespondingFieldAccessMethodRule3[stRight.
              pExpressionLeft,varN] =>
              correspondingAssignmentMethodRule3[p,n,stRight,stLeft,
              varN])
13
14         stRight in AssignmentExpression =>
15         stLeft in AssignmentExpression &&
16         (noCorrespondingFieldAccessMethodRule3[stRight.
              pExpressionLeft,varN] => (stLeft=stRight))
17
18         stRight in MethodInvocation =>
19         stLeft in MethodInvocation &&
20         (noCorrespondingMethodInvocationMethodRule3[p,n,stRight,
              stLeft,varN] => (stLeft=stRight))
21
22         stRight in MethodInvocation =>
23         stLeft in MethodInvocation &&
24         (!noCorrespondingMethodInvocationMethodRule3[p,n,stRight,
              stLeft,varN] =>
              correspondingMethodInvocationMethodRule3[p,n,stRight,
              stLeft,varN])
25     }
26     }
27 }
```

Code 3.64 Predicate that ensures that the *FieldAccess*, passed as parameter, does not suit in the pattern previewed by the second substitution rule in Figure 2.

```
1  pred noCorrespondingFieldAccessMethodRule3 [ae: FieldAccess,
       varN: FieldId]{
2      ae.pExp !in ExpressionName ||
3      (ae.pExp in ExpressionName && ae.pExp.name != varN)
4  }
```

Code 3.65 Predicate that ensures that the *MethodInvocation*, passed as parameter, does not suit in the pattern previewed by the second substitution rule in Figure 2.

```
1 pred noCorrespondingMethodInvocationMethodRule3 [p: Program, n:
      ClassId, mi,mi2: MethodInvocation, varN: FieldId]{
2     mi.pExp !in ExpressionName ||
3     (mi.pExp in ExpressionName && mi.pExp.name != varN)
4 }
```

As the left–hand side expression of an *AssignmentExpression* is always a *FieldAccess* expression, another predicate (named *correspondingFieldAccessMethodRule3*) is present in the predicate *correspondingAssignmentMethodRule3* (see line 3, Code 3.66) to guarantee the substitutions required by the second substitution rule in Figure 2 are done in the *FieldAccess* expression. On the other hand, if the right–hand side expression of the *AssignmentExpression* refers to a *MethodInvocation* expression (line 5), then another predicate is necessary to guarantee the substitutions: the *correspondingMethodInvocationRule3* (Code 3.68). Otherwise, in case this expression refers to a *LiteralValue* we simplify saying that the other correspondent expression needs also to be a *LiteralValue* (line 7).

Code 3.66 Predicate that does the correspondence between the *AssignmentExpressions* that are statements (with the same index) in the method being moved, according to the pattern previewed by the second substitution rule in Figure 2.

```
1 pred correspondingAssignmentMethodRule3 [p: Program, n: ClassId
     , ass,ass2: AssignmentExpression, varN: FieldId]{
2
3     correspondingFieldAccessMethodRule3[p,n,ass.pExpressionLeft,
        ass2.pExpressionLeft,varN]
4
5     ass.pExpressionRight in MethodInvocation =>
        correspondingMethodInvocationRule3[p,n,ass.
        pExpressionRight,ass2.pExpressionRight,varN]
6
7     ass.pExpressionRight in LiteralValue => ass2.
        pExpressionRight in LiteralValue
8 }
```

The *correspondingFieldAccessMethodRule3* predicate, Code 3.67, attributes the *this* expression to the value for the relation *pExp* of the *FieldAccess* expression being replaced in the method *m()* after the transformation (line 3). In addition, the *id_fieldInvoked* relation remains the same as before the transformation (line 3). This is done only in the case that the relation *pExp* of the *FieldAccess* expression before the transformation is of type *ExpressionName*, and whose *name* relation is equal to the field *b*, which is represented by the *varN* variable (line 2)—this scenario fits the one required for the substitution rule *c* defined in Figure 2. Otherwise, or in the case the *noCorrespondingFieldAccessMethodRule3* predicate (line 5) fits, both *AssignmentExpression*

expressions (in each method *m* before and after the transformation) can be the same (line 5). The difference here, if we compare to what is defined by the rule, is the application of *this* expression, when the rule does not specify anyone. We apply the *this* expression only to fit in our OO model that requires exactly one *PrimaryExpression* for the *pExp* relation in the *FieldAccess* type. And, in this case, having or not the *this* expression does not make difference.

Code 3.67 Predicate that does the correspondence between the *FieldAccess* that are statements (with the same index) in the method being moved, according to the pattern previewed by the second substitution rule in Figure 2.

```
1 pred correspondingFieldAccessMethodRule3[p: Program, n: ClassId
     , ae,ae2: FieldAccess, varN: FieldId]{
2     (ae.pExp in ExpressionName && ae.pExp.name = varN) =>
3      ((ae != ae2) && ae2.pExp in this_ && ae2.id_fieldInvoked =
          ae.id_fieldInvoked)
4
5     noCorrespondingFieldAccessMethodRule3[ae,varN] => (ae = ae2)

6 }
```

The predicate *correspondingMethodInvocationMethodRule3*, Code 3.68, do exactly the same as the predicate just described but for a *MethodInvocation*. Besides, it checks if the real parameter of the *MethodInvocation* (which in our OO model can only be a *FieldAccess* expression) fits in the scenario described by the substitution rule *c* defined in Figure 2. If this is the case, predicate *correspondingFieldAccessMethodRule3*, just described, is applied to the real parameter (lines 11 to 13).

Code 3.68 Predicate that does the correspondence between the *MethodInvocations* that are statements (with the same index) in the method being moved, according to the pattern previewed by the second substitution rule in Figure 2.

```
1 pred correspondingMethodInvocationMethodRule3 [p: Program, n:
     ClassId, mi,mi2: MethodInvocation, varN: FieldId]{
2
3     all someNMethod: (n.(p.classDeclarations)).methods |
4     (mi.pExp in ExpressionName && mi.pExp.name = varN &&
5      mi.id_methodInvoked = someNMethod.id) =>
6     mi2.pExp in this_ && mi2.id_methodInvoked = someNMethod.id
7
8     mi.realParam in FieldAccess => mi2.realParam in FieldAccess
          &&
9     (noCorrespondingFieldAccessMethodRule3[mi.realParam,varN] =>
          (mi.realParam=mi2.realParam))
10
11    mi.realParam in FieldAccess => mi2.realParam in FieldAccess
          &&
```

```
12     (noCorrespondingFieldAccessMethodRule3[mi.realParam,varN] =>

13     correspondingFieldAccessMethodRule3[p,n,mi.realParam,mi2.
          realParam,varN])
14 }
```

Line 37 in Code 3.48 establishes the equivalence between the mapping of the classes in left and right–hand side (before and after the refinement), following the same reasoning in the earlier specifications already explained— as can be seen, the only difference is the *M* and *N* classes that, although their correspondent classes in the different sides have the same *id*, they are different internally, because of the difference in their methods as their bodies change. Their set of fields are equals (lines 39 and 40) since they do not change before and after the refinement. Finally, lines 44 to 47 follows the same reasoning already explained in Section 3.2, for Code 3.21.

3.2.6 Transformation–Specific Model for Rule 6

As mentioned in Section 2.1.4, Rule 6 is another *rCOS* refinement law described in (QUAN; ZONGYAN; LIU, 2008). This one is simpler than the Rule 3, just described, since substitutions are not required. As a consequence, the main predicate that represents it (see Code 3.69) is simpler as well as the predicates it uses.

Code 3.69 Predicate that captures the refinement described in Rule 6.

```
1  pred rule6LR[m,n1,n2: ClassId, meth: Method, left,right:
      Program]{
2      m != n1
3      n1 != n2
4
5      let leftCds = left.classDeclarations,
6          rightCds= right.classDeclarations,
7          ml = m.leftCds,
8          mr = m.rightCds,
9          n1l = n1.leftCds,
10         n1r = n1.rightCds,
11         n2l = n2.leftCds,
12         n2r = n2.rightCds
13         {
14
15         // SS description
16         n1l.extend = m
17         n2l.extend = m
18         meth in (n1l.methods)
19         meth in (n2l.methods)
20
21         // premise6LR
22         methodOnlyRefersToAttributesInM[meth,n1l,ml,left]
```

```
23
24          // RS description
25          meth in (mr.methods)
26          meth !in (n1r.methods)
27          meth !in (n2r.methods)
28          n1r.extend = m
29          n2r.extend = m
30
31          //equivalence between left and right-hand sides
32          ml.extend = mr.extend
33          m.~((left.classDeclarations).extend) = m.~((right.
               classDeclarations).extend)
34          n1.~((left.classDeclarations).extend) = n1.~((right.
               classDeclarations).extend)
35          n2.~((left.classDeclarations).extend) = n2.~((right.
               classDeclarations).extend)
36          leftCds = rightCds ++ {m->ml} ++ {n1->n1l} ++ {n2->n2l}
37
38          ml.fields = mr.fields
39          n1l.fields = n1r.fields
40          n2l.fields = n2r.fields
41          ml.methods = mr.methods-meth
42          n1l.methods - meth = n1r.methods
43          n2l.methods - meth = n2r.methods
44
45          }
46  }
```

Following the pattern defined in Code 3.21, the parameters of the main predicate *rule6LR* are the elements involved in the refinement (along with the two possible contexts: left or right). Lines 2 and 3 ensure the *M*, *N1* and *N2* classes are different, thus having different identifiers. Lines 5 to 12 retrieve each class from its correspondent context, likewise earlier specifications do.

Assuming a refinement is a transformation from the left– to the right–hand side, left description (SS) establishes class *M* as the super class of both *N1* (line 16) and *N2* (line 17) classes, and that the method *meth* (which refers to the method that is moved to class *M* after the refinement) is in both *N1* (line 18) and *N2* (line 19) classes, as defined in the specification. The *methodOnlyRefersToAttributesInM* predicate (present in line 22 and defined in Code 3.69) is used to represent what is defined in the specification: *"all attributes used in m() are in M"*. By this sentence, we assume that every field used in m() is a field from class *M*. Observe that this restriction is very similar to the one defined in Rule 3, but there is a slight difference. In Rule 3, the specification restricts that all the accesses in the method being moved are through an attribute b whose type is the same as the type of the super class.

The possibilities to a field appear in a method body is through an expression of type *FieldAccess* that can, in turn, appear in an *AssignmentExpression* (line 4, Code 3.70) or in a *MethodInvocation* statement (line 10, Code 3.70). For the former case, there are two possibilities: through its left–hand side expression (see line 4)—which in our OO model can only be a *FieldAccess*—or through its right–hand side expression (see line 6), which in our OO model can only be a *MethodInvocation*, whose real parameter (if exists) is a *FieldAccess*. This last possibility is also the case for the latter (see lines 10 and 11). As can be seen, in all situations, the predicate *lhsOnlyRefersToFieldsInM* (see Code 3.71) is the responsible for guaranteeing that the field identifier(s) used in the *FieldAccess* expression are in class *M*.

Code 3.70 Predicate that ensures every field used in m() is a field from class *M*, according to the specification in Figure 4.

```
1  pred methodOnlyRefersToAttributesInM[method: Method,
       correspondingNClass,correspondingMClass: Class, p: Program]
       {
2
3       all st:Statement | st in univ.(method.body) =>
4       (st in AssignmentExpression => lhsOnlyRefersToFieldsInM[p,
          correspondingNClass,correspondingMClass,st.
          pExpressionLeft]) &&
5
6       (st in AssignmentExpression && st.pExpressionRight in
          MethodInvocation &&
7        #(st.pExpressionRight.realParam) > 0 =>
8        lhsOnlyRefersToFieldsInM[p,correspondingNClass,
           correspondingMClass,st.pExpressionRight.realParam]) &&
9
10      (st in MethodInvocation && #(st.realParam) > 0 =>
11       lhsOnlyRefersToFieldsInM[p,correspondingNClass,
           correspondingMClass,st.realParam])
12
13 }
```

The *lhsOnlyRefersToFieldsInM* predicate applies the *whichFieldIs* auxiliary function, explained in Section 3.2.5, on both *pExp.name* (see second parameter, line 4, Code 3.71) and *id_fieldInvoked* (see second parameter, line 5, Code 3.71) relations of a *FieldAccess* expression. In the first case (line 4), the function is applied to the *name* relation of the *pExp* relation, in case this one refers to an *ExpressionName* type, which is a subsignature of the *PrimaryExpression* type—see Section 3.1. The goal is to detect the first field in the correspondingNClass class hierarchy that owns the specific identifier referred in *lhs.pExp.name* (if exists) or in *lhs.id_fieldInvoked* expression. This field needs to be in class *M* (lines 9 and 14).

Code 3.71 Predicate that ensures the field used in *FieldAccess* expression is from class *M*.

```
1  pred lhsOnlyRefersToFieldsInM[p: Program,correspondingNClass,
       correspondingMClass: Class,lhs: LeftHandSide] {
2
3      let
4      fieldInExprName = whichFieldIs[p,lhs.pExp.name,
           correspondingNClass],
5      fieldInFieldInvoked = whichFieldIs[p,lhs.id_fieldInvoked,
           correspondingNClass]{
6
7      (lhs in FieldAccess && lhs.pExp in ExpressionName =>
8       (#fieldInExprName = 1 && fieldInExprName.type !in Long_ &&
9       lhs.pExp.name in (correspondingMClass.fields).id ))
10
11     (lhs in FieldAccess && lhs.pExp !in ExpressionName =>
12      (lhs.pExp !in super && #fieldInFieldInvoked = 1 &&
13      fieldInFieldInvoked.type !in Long_ &&
14      lhs.id_fieldInvoked in (correspondingMClass.fields).id ))
15     }
16 }
```

Returning to the main predicate, *rule6LR*, Code 3.69, lines 25 to 29 describes the scenario *after* the refinement is applied. For instance, the method being moved, which is represented by the variable *meth*, is in class *M* (line 25) and not anymore in the set of methods of classes *N1* (line 26) and *N2* (line 27). In addition, class *M* continues in the *extend* relation of classes *N1* (line 28) and *N2* (line 29). Lines 41 to 43 reinforce this.

4

VALIDATING TRANSFORMATION SPECIFICATIONS

In this chapter, the Static Semantics and the Dynamic validation steps are detailed in Sections 4.1 and 4.2, respectively. Section 4.3 describes how our Alloy–To–Java translator works; it is responsible for translating Alloy instances into Java programs as well as generating test classes, which enable the Java transformations to be validated with regard to dynamic behaviour. Finally, Section 4.4 describes how our transformation specifications were validated and the results obtained so far using our strategy.

4.1 Static Semantics Validation Step

Figure 22 details the part of our strategy with regard to the static semantics validation of a transformation specification. It also details the Static Semantics Validation step in Figure 1. When the Static Semantics Validator is executed, it is assumed as a premise that the SS program of the transformation is well–formed. For guaranteeing this, the model of a subset of Java is used through its main predicate: the *wellFormedProgram* predicate, which receives a program as parameter (in case of the premise, the SS program). This predicate guarantees that the program passed as parameter is a well–formed Java program. Apart from syntactic aspects, the model of a subset of Java embodies a static semantics via predicates and constraints that ensure type correctness in the object instances that represent the SS program of the transformation. In other words, in this case we have that the SS program is compilable.

On the other hand, the *wellFormedProgram* predicate is not applied to the RS program, so its well–formedness would have to be guaranteed by the transformation–specific model predicates since a well–formed program is considered before the transformation. Our Static Semantics validator model checks, through an assertion, if the following implication holds, up to a given scope, for a specific transformation: *wellFormedProgram[SS] && transformationPredicate implies wellFormed[RS]*. In this way, if there is any instance returned, it is because the resulting program is not well–formed. Hence, we can assert that there is some failure in the transformation–specification since SS did not present any static semantics problems before. An example of this Alloy validator model is shown in Code 4.1. This checking is executed by the Alloy Analyzer. This step just described is represented in the first decision point, Figure 22.

If counter examples for the assertion stated by the Static Semantics Validator are found

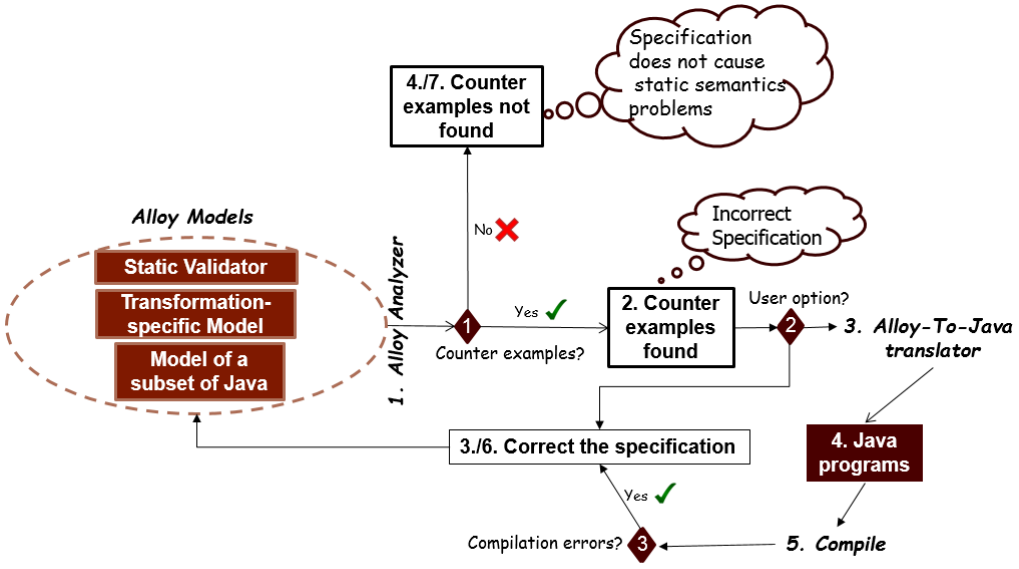


Figure 22 Overview of the first part of our complete strategy.

(Step (2) of Figure 22), then they will be presented indistinctly by the Alloy Analyzer, independent of the static semantics problem they present. Thus, the user has two options (second decision point) to identify the specific static semantics problem presented by the instances: (a) examine the counter examples pointed by the Alloy Analyzer (Step (3./6.)) or (b) use the Alloy–To–Java Translator to translate these counter–examples (Step (3)) into Java programs (Step (4)) and, through a compilation process of each of these programs (this process is done by our strategy, step (5)), examine the compilation errors (decision point (3))—so the static semantics problem caused by the transformation becomes more evident. Afterwards, one needs to correct the transformation specification (Steps (3./6.) and re–submit the models again to the Alloy Analyzer (1). This process is done until no counter examples are found (4./7.); in other words, until all the RS programs generated by the transformation do not present any static semantics problems.

The Static Semantics Validator Model presented in Code 4.1 is relative to Rule 3: see Section 3.2. However, the Static Semantics Validator Model follows the same pattern to all the specifications being analyzed. The only variation is in the *transformationPredicate* (i.e., predicate *rule3LR* in case of Rule 3), that will be replaced by the predicate of the corresponding transformation being analysed, along with the corresponding direction—from the left– to the right–hand side or vice-versa. In case of a refinement, we have only one possible direction: from the left– to the right–hand side, where the left program is the SS one and the right, the RS. In addition, the *wellFormedProgram* predicate is always applied to the program representing the starting side of the transformation (in the case of Code 4.1, the left–hand side because Rule 3 is being analysed from the left to the right–hand side). If the implication described in lines 6 and 7, Code 4.1, always holds, it means that no counter examples are found and the transformation does not introduce static semantics problems in the resulting program.

On the other hand, if a counterexample is returned by the Alloy Analyzer from the assertion used by the Static Semantics Validator Model, this means that there is some error in the transformation specification or in the predicates that specify it (in the Alloy model) because the instances that represent the starting–hand side of the transformation were generated according to a Java metamodel that ensures conformance with the static semantics in its classes, as explained in Chapter 3. Also observe that the analysis scope can be specified (line 9, Code 4.1) to bound the Alloy Analyzer evaluation. As explained in Section 2.2.2, second paragraph, a bound is specified (in this case is 12 in line 9) to enable the analysis be performed. This bound means that there are at most 12 (for each top–level signature) instances in each Alloy instance generated for the model. The exceptions for this bound of 12 is the *Program* type—since only 2 (the one before and the other after the transformation) are necessary— and the *Method* and *Field* types (we observe 6 method and 3 field instances, respectively, are a good number for the analysis). We could change our bound limitation and restrict even more top–level signatures of our model, but this one was enough to generate interesting Alloy instances, which evidence errors described in this thesis. Actually, it is just an example and we vary the bound limitation in our analysis, especially if we consider other specifications analysis.

Code 4.1 Representation of the Static Semantics Validator

```
1 module staticSemanticsValidator
2 open transformationSpecificModel
3
4 assert rule3LeftToRightTransf {
5    all m,n: ClassId, mBefore,mAfter: Method, f: Field, left,
        right: Program |
6    (wellFormedProgram[left] && rule3LR[m,n,mBefore,mAfter,f,
        left,right]) implies wellFormedProgram[right]
7 }
8
9 check rule3LeftToRightTransf for 12 but exactly 2 Program,
   exactly 6 Method, exactly 3 Field
```

4.2 Dynamic Validation Step

The next step of our strategy is to check for dynamic problems. It is depicted in Figure 23, which in turn details the dynamic validation step in Figure 1. We then submit another Alloy Model to the Alloy Analyzer when no more counter examples are found by the Static Semantics Validator (see Figure 23). Because of this, in this step, both SS and RS programs are well–formed (Step 8, Figure 23). They are generated through the Dynamic Validator Alloy Model, which uses the same transformation–specific model and also the model of a subset of Java, but instead generates all the possible instances that represent the transformation, according to the exhaustive analysis provided by the Alloy Analyzer and within a given scope. Afterwards, these instances

are further submitted to the Alloy–To–Java Translator tool (Step (3./9.), Figure 23) that generates Java programs (Step (4./10.)) corresponding to these instances. In this step, we benefit from the ASTs of the programs (generated by the Alloy–To–Java translator) to build the test classes. All the classes (including the test ones) are added in the compilation unit to be submitted to the Java compiler (Step (5./11.)). The non–occurrence of programs with structural or syntactic problems (only well–formed SS and RS programs) is confirmed in the *Validate* step (see step 12 in Figure 23) that compiles (in Java) each side of each transformation, including the test classes, after the translation of the Alloy–To–Java Translator. Finally, the validation with regard to dynamic problems is also done in step (12) through the execution of the corresponding test class—the invocation results (in both SS and RS programs) are compared to check behaviour preservation after the transformation; if they are not the same, then there is a behavioural non-conformance in the transformation, also meaning a transformation specification error.

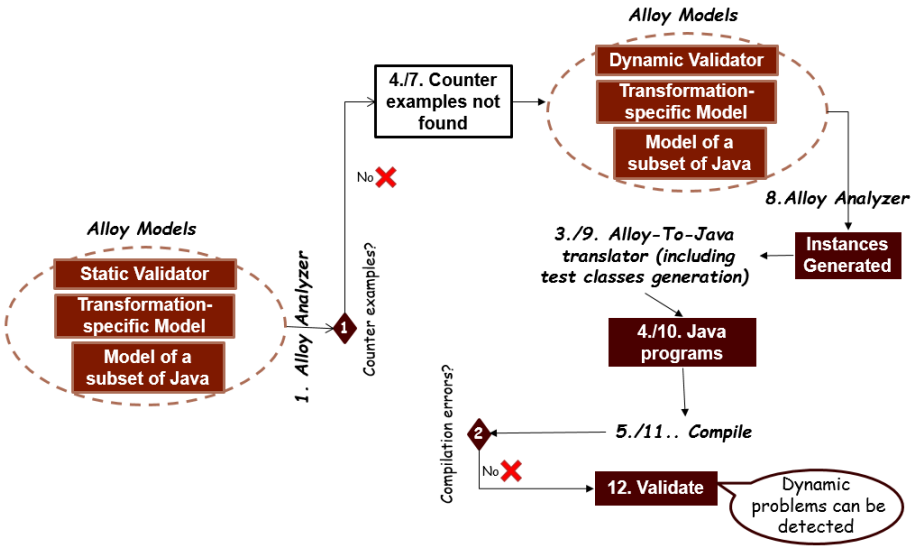


Figure 23 Second part of our Validation Strategy.

The Dynamic Validator is illustrated in Code 4.2. It is submitted to the Alloy Analyzer to execute the second part of our strategy (see Figure 23).

Code 4.2 Representation of the Dynamic Validator

```
1 module dynamicSemanticValidator
2 open transformationSpecificModel
3
4 pred dynValidationRule3[] {
5     all m,n:ClassId, mBefore,mAfter: Method, f: Field, left,
          right: Program |
6     rule3LR[m,n,mBefore,mAfter,f,left,right]
7 }
8
```

```
9 run dynValidationRule3 for 12 but exactly 2 Program, exactly 6
    Method, exactly 3 Field
```

4.3 The Alloy–To–Java Translator

JDolly is a program generator; it is particularly used to automatically generate test inputs to be submitted to the refactoring engine implementations (SOARES, 2015). Actually, JDolly generates Alloy instances according to an Alloy metamodel for a subset of Java. These instances are translated into Java programs and then used as test cases to refactoring implementations. The evaluation of the correctness of these transformations is done (in (SOARES, 2015)) using another tool, called SafeRefactor (SOARES *et al.*, 2010). In order to complete our strategy, we have developed, besides the Alloy OO metamodel (an improvement on the one in (SOARES, 2015)) and several transformation–specific models, an Alloy–To–Java translator tool that translates the Alloy instances generated by both Static Semantics and Dynamic Validators.

As discussed in Section 4.1, the Alloy–To–Java Translator can also be applied in the static validation step (beyond the dynamic one) to ease the examination of the errors in the resulting programs. The generated pair of programs in the static validation step can be translated and then compiled by the Alloy–To–Java Translator, depending on the user decision (see second decision point in Figure 22), so one can examine the compilation errors found in the resulting programs instead of analyzing an Alloy instance in the Alloy Analyzer evaluator tool.

One difference between our OO metamodel and the one used by JDolly in (SOARES, 2015) is that our metamodel captures, when desired, the static semantics of a Program, considering the OO subset defined in this model, which is not the case in the metamodel used in JDolly, which in turn allows programs to be generated with compilation errors. As a second improvement, our Alloy instances, in both static and dynamic validation steps, already represent the pair of programs following a transformation specific model. In the first case, it is known that, if there are static issues, the resulting program present static semantics problems. On the other hand, in case of dynamic validation step, the pair of programs are both well–formed. Afterwards, the Alloy–To–Java translator generates Java programs for the instances that represent the left– and the right–hand sides of the transformation, rather than for an instance that only represents a single Java program, as in JDolly.

Our Alloy–To–Java translator identifies related Java classes from an Alloy transformation instance. Object instances such as the ones of type *Class* or of type *Method* are only generated as pairs when, depending on the transformation defined, substitutions and the internal context of these elements change. These pairs, when generated, are distinct as their internal structure changes, due to these substitutions—if any one is necessary. They have the same id (for each pair) and play corresponding roles on the SS and RS.

The Alloy–To–Java translator maps an Alloy abstract syntax tree into two Java abstract syntax trees (one for each of the SS and RS of the transformation). In addition, test classes are

generated for each method in common among the corresponding classes. If there is no method in common (with the same signature), we check for the case that a method exists in classes with an inheritance relationship (for instance, in Law 2, method *m* does not exist in the same classes of corresponding sides but it exists in a class that is a supertype of the one being analysed—i.e. class C). In this case, the method is invoked by the test class from the class C of both sides. As can be seen, the generation of the test classes is very simple, basically these classes only invoke the already generated method by the Alloy Models, because we consider that the distinguishing feature of the tests is exactly what is generated by the Alloy Analyzer from the Alloy Models. The results of these invocation executions are compared; if they are not the same, then there is a behavioural inconsistency in the transformation. In a future work, we can also use a more elaborate test campaign to improve test generation.

The Alloy–To–Java translator stores the Java trees into Java files (as well as the corresponding test classes) in the corresponding folders for each side of the transformation. Afterwards, another tool, named Java Validator, is used to compile and execute them, comparing the corresponding results and pointing the cases where there are behavioural inconsistency. The results found so far show that our strategy has the potential to find many bugs also found using more sophisticated test Oracles.

4.4 Evaluation

In this section, we describe the validation results focusing on each one of the transformation specifications analysed. We evaluated seven transformation specifications, selected from each kind of specification: algebraic law, refactoring rule and rCOS transformation (see Chapter 2). These transformation specifications were validated in different contexts in the literature. Some of these specifications were (1) postulated regardless proofs of their correctness. Others, although (2) proved or (3) derived from provably correct ones, presented errors identified by our strategy. For instance, concerning the proved ones context, some refactoring rules in (CORNELIO, 2004) presented both static semantics and behaviour problems. On the other hand, laws in (DUARTE, 2008), derived from the ones in (BORBA *et al.*, 2004) and adapted for Java, presented behaviour problems (due to incorrect or missing provisos) and some redundant conditions. The specifications that are only postulated have a higher probability to present problems and we found some of them using our strategy. We uncover problems in these three contexts, namely:

1. Transformation specifications that were postulated, regardless correctness proofs or any kind of validation.
2. Transformation specifications that were proved.
3. Transformation specifications derived from provably correct ones.

Firstly, we present the experiment definition (see Section 4.4.1) and planning (see Section 4.4.2). Next, Section 4.4.3 discusses the results found in each transformation specification. Finally, Section 4.4.4 describes some threats to validity and difficulties faced in the evaluation.

4.4.1 Experiment definition

In all of the contexts mentioned earlier (beginning of the Section 4.4), there are different types of errors that can be found. In particular, we focus on the following research points: (a) whether all the provisos or conditions in the transformation specifications are enough not to cause static semantics problems in the resulting programs after the transformation is applied. The second point (b) is similar, but for behavioural problems instead. In the specifications considered, we identified that some provisos or conditions are missing and the application of the transformations may result in errors. Finally, we address the following concern (c): are there any redundant conditions or provisos in the transformation specification? We found in our analysis that some provisos or conditions present in some transformation specifications, such as the one in Law 2, are not necessary to guarantee the correctness of the transformation because some predicates are of the form $p \wedge q$ where $p \implies q$ and so we can replace $p \wedge q$ with p.

4.4.2 Planning

In this section, we describe the subjects used in the experiment and its instrumentation.

4.4.2.1 Selection of Subjects

The specifications analysed with regard to the contexts just described are summarized in Table 1, where each line represents one of the transformations analysed whereas the columns represent the contexts just described.

Table 1 Classification of main specifications analysed according to each context

Transformations Analysed	1 (Postulated)	2 (Proved)	3 (Derived)
Law 1—Move Attribute in (DUARTE, 2008)			X
Law 2—Move Method in (DUARTE, 2008)			X
Law 4—Push Down Method in (SCHAFER, 2010)	X		
Rule 2.1—Pull Up/Push Down Method in (CORNELIO, 2004)		X	
Rule 3—Move Method in (QUAN; ZONGYAN; LIU, 2008)	X		
Rule 6—Pull Up Method in (QUAN; ZONGYAN; LIU, 2008)	X		
Law 5—Move Attribute, adapted from (DUARTE, 2008)	X		

As already discussed, these specifications were selected for different reasons. Some of them were proved or transcribed from other ones already proved. In these cases, finding

errors using another technique (i.e. our strategy) seems more exciting/interesting. Another point considered for the selection is evaluating transformation specifications in different kind of specifications: algebraic law, refactoring rule and rCOS transformation (see Chapter 2).

4.4.2.2 Instrumentation

We ran the experiment on a notebook 1.80 GHz core i7-4500U with 8 GB RAM running Windows 8 Single Language with JDK 1.8. In addition, the solver used by the Alloy Analyzer was the *Sat4J*. As already mentioned, we used different transformation specifications present in different works (DUARTE, 2008; SCHAFER, 2010; CORNELIO, 2004; QUAN; ZONGYAN; LIU, 2008; PALMA, 2015). Our Static Semantics Validator was used to evaluate whether these specifications preserves the static semantics of the resulting programs (generated after the application of the transformation specification). On the other hand, our Dynamic Validator was used to detect if behaviour is preserved, instead. Our Alloy models did not contemplate the package element, and the programs generated did not contain packages as a consequence. The scope, with regard to the bound used for limiting each top–level signature in the Alloy models being evaluated, varies according to each transformation specification. This scope can be seen in the Static Semantics and Dynamic Validator models, available online[1]. The scope limitation is also discussed in Section 4.4.4.

4.4.3 Results

Considering all of the contexts discussed earlier (in the beginning of this section), different kind of errors were found, namely:

(a) Provisos or conditions in the transformation specifications that are not sufficient to ensure the absence of static semantics errors after the transformation is applied.
(b) Provisos or conditions in the transformation specifications that are not sufficient to ensure the absence of behavioural problems in the resulting programs after the transformation is applied.
(c) Redundant conditions or provisos in the transformation specification.

The results concerning the different type of errors found in the specifications analysed are summarized in Table 2, where each line represents one of the transformations analysed whereas the columns represent the errors just described. In the following subsections, we discuss about the different kind of errors found in each transformation specification.

4.4.3.1 Analysis for Law 2

The first transformation specification analysed, Law 2 (see Section 2.1.2), presented problems in almost all contexts—except the one related to static semantics problems—item (a) of

[1] It can be downloaded from http://www.cin.ufpe.br/~tds/phd/JTransformations.

Table 2 Comparison of main specifications analysed according to the different errors found

Transformations Analysed	(a) Static Semantics	(b) Behavioural	(c) Redundant conditions
Law 1—Move Attribute in (DUARTE, 2008)	No	No	No
Law 2—Move Method in (DUARTE, 2008)	No	Yes	Yes
Law 4—Push Down Method in (SCHAFER, 2010)	No	Yes	No
Rule 2.1—Pull Up/Push Down Method in (CORNELIO, 2004)	Yes	Yes	No
Rule 3—Move Method in (QUAN; ZONGYAN; LIU, 2008)	Yes	Yes	No
Rule 6—Pull Up Method in (QUAN; ZONGYAN; LIU, 2008)	Yes	Yes	No
Law 5—Move Attribute, adapted from (DUARTE, 2008)	Yes	No	No

the enumeration in Section 4.4. Firstly, we describe the one related to behaviour consistency in the resulting programs after the transformation is applied: item (b). Suppose we have a method defined with the same id and formal parameters of the method being moved, in a class *B super* type. When the method *m* is moved from class *C* to class *B* (from the left to the right–hand side transformation direction), it results in a method redefinition on the right–hand side in class *B* (not existing before). Some behavioural changes can occur if we consider a method invocation where the *B* class is the target. This same problem can occur in the inverse direction of the transformation (right to left). This error was identified by our Dynamic Validator. For instance, the method invocation *new B().m()* returns 3 in the left–hand side program and 2 in the right one (see Figure 24).

So, a proviso with this restriction is necessary for both sides of the transformation ($(\leftrightarrow)$)—see proviso ($(\leftrightarrow)$) (2) in Law 3 (below). We rewrite Law 2 as Law 3, fixing all the errors found in provisos of the former. The other contexts in which errors are found are discussed throughout this section.

Law 2 was manually translated into Java (DUARTE, 2008) from Law 8 (BORBA *et al.*, 2004), which is a provably correct specification in ROOL. Hence, even in the case of a transcription, errors can occur and this is the case for Law 2. Thus, context addressed in item 3 in Section 4.4 was also identified by our analysis. Observe that the behavioural error just described, in proviso ($(\leftrightarrow)$ (2)) of Law 2, was apparently a spelling error since in Law 8 (BORBA *et al.*, 2004) this same proviso already existed, meaning that probably the word *super* was incorrectly spelled as *subclasses*.

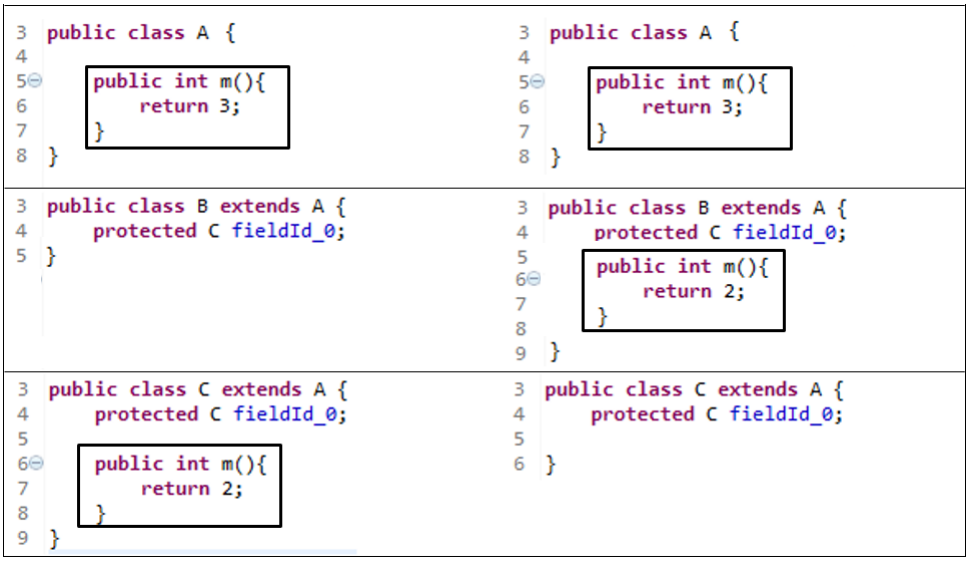


Figure 24 An example of program generated with behavioural problems when Law 2 is applied.

Our strategy can also help to find redundant conditions in the transformation specifications. Analysing the Alloy Model for Law 2 from the right– to the left–hand side, we observe that there are some redundant provisos in the sense that some of them logically imply in others. For instance, we observe that when the predicate responsible for guaranteeing the proviso ($(\leftarrow)$ (2)) in Law 2 is considered, there is no significant changes in the Alloy Analyzer instances when compared with a scenario where the predicate guaranteeing the proviso ($(\leftrightarrow)$ (2)) in Law 2 is considered or not. We realized this by running our Static Semantics validator without all the predicates representing the provisos in the transformation specification—we have intentionally commented some of them. Hence, we did a manual analysis (since some predicates were commented) along with the use of our Static Semantics validator to achieve the analysis of the item (c), described in Section 4.4. We can also verify that, considering the existence of all predicates to guarantee the transformation from the right– to the left–hand side, when the predicate representing the proviso ($(\leftarrow)$ (2)) in Law 2 is considered and the other one ($(\leftrightarrow)$ (2)) in Law 2 is not, the Static Validator remains not returning any counter–examples. This gave us some clue that the predicate representing the proviso ($(\leftarrow)$ (2)) in Law 2 implies the one representing the proviso ($(\leftrightarrow)$ (2)) in Law 2. We further discuss this hypothesis below.

Taking into account the proviso ($(\leftrightarrow)$ (2)) in Law 2, suppose there is a method redefinition of *m* in a class D which is a subclass of B. When the method *m* is moved from B to C, the method in D is not anymore a redefinition and this can cause behavioural changes when the method *m* is called in other classes and D (or a subclass of D) is used as the type of the target of this call. However, this scenario could only occur if the access to the method *m* was allowed from B subtypes that are not C ones (which is the case of class D in question), which is denied by the proviso ($(\leftarrow)$ (2)) in Law 2. So this last proviso logically implies the proviso ($(\leftrightarrow)$ (2)) in Law 2 since it cannot cause behavioural problems if there is no access to the method. Thus, we

can remove the proviso ($(\leftrightarrow)$ (2)) from Law 2 because it is not necessary from the right– to the left–hand side direction of the transformation.

For the same reason, the proviso ($(\leftrightarrow)$ (3)) from Law 2 is also not necessary from the right– to the left–hand side direction of the transformation. Observe that the accessibility of the method being moved does not matter since its access is avoided by the proviso ($(\leftarrow)$ (2)) in Law 2 from *B* but not *C* subtypes. Even supposing we have an access (from a *C* subtype) of the method being moved, this access is only possible if the method is not private. Hence, we have two cases of redundant provisos in Law 2: provisos (2) and (3) ($(\leftrightarrow)$) in Law 2.

We can still think about the need for the existence of these two provisos with regard to from the left– to the right–hand side direction of the transformation. In the former case, if a method *m* with the same id and formal parameters of the method being moved is declared in a *B* subtype, then the behaviour of this method (in some call to it) can change since the method *m* will be moved from *C* to *B* and in this scenario we have a method redefinition that did not exist before. Hence in this case the proviso is required only from the left– to the right–hand side direction, to avoid behavioural problems. In the latter case (proviso ($(\leftrightarrow)$ (3) in Law 2), suppose there is an invocation to the method being moved in class *C*. If this method is moved to class *B*, then the invocation will be invalid if the method is private. Thus, again, the proviso is required from only the left– to the right–hand side direction and the method cannot be private. Hence the provisos $(\leftrightarrow)$ (2) and (3) should be migrated to be provisos in the $(\rightarrow)$ direction only. Unfortunately it was not possible for us to completely validate this law because in proviso ($(\rightarrow)$ (2)) a cast expression appears and this kind of expression is out of the scope of our OO model. Although we did not analyse the entire set of laws in (DUARTE, 2008), we imagine that, just as we have found errors in laws that were derived from provably correct ones in (BORBA *et al.*, 2004), there maybe exist similar errors in the other ones catalogued in (DUARTE, 2008).

Law 3. ⟨move original method to superclass with corrections⟩

class *B* **extends** *A* {		**class** *B* **extends** *A* {
ads		*ads*
cnds		*cnds*
mds		*rt m*(*pds*) {*mbody*}
}		*mds*
class *C* **extends** *B* {	$=_{cds,Main}$	}
ads′		**class** *C* **extends** *B* {
cnds′		*ads′*
rt m(*pds*) {*mbody*}		*cnds′*
mds′		*mds′*
}		}

provided

($\leftrightarrow$) *(1) super and private attributes do not appear in mbody; (2) m(pds) is not declared in any superclass of B in cds.*

($\rightarrow$) *((1) m(pds) is not declared in mds; (2) mbody does not contain uncast occurrences of* **this** *nor expressions in the form* $((C)$**this**$).a$ *for any protected attribute a in ads′; (3) m(pds) is not declared in any subclass of B in cds; (4) m(pds) is not private.*

($\leftarrow$) *(1) m(pds) is not declared in mds′; (2) D.m(e), for any* $D \leq B$ *and* $D \not\leq C$*, does not appear in cds, Main, mds or mds′*

4.4.3.2 Analysis for Push Down Method Refactoring

Due to the errors found in Law 2 and discussed in Section 4.4.3.1, we change the transformation–specific model for Push Down Refactoring accordingly. The specification representing this one is now Law 4. The difference of this one to the Law 3 is that we incorporated a common practice adopted by the developers in this kind of refactoring, described in Section 3.2.3, which is represented by the item (3) of the proviso ($\leftarrow$) in Law 4. The inclusion of this proviso caused behaviour problems in RS programs (after the transformation is applied)—item 3 of the enumeration in the beginning of Section 4.4. Observe also that the item (1) in proviso ($\leftrightarrow$) was modified (compare Law 3 with Law 4), for the reasons already explained in Section 3.2.3. We adapted the Alloy model(s) that represent the specification in Law 3 to the one(s) that represent the specification in Law 4, which in turn represent the Push Down Method Refactoring.

But still note the main difference between the specification in Law 3 and Law 4, with regard to the Alloy Models that represent them, is basically the presence of the predicate *correspondenceBetweenMethods*, already described in Section 3.2.3. It remained the same although many changes occurred in the Alloy model representing the Law 2. With the new specification (Law 4), and considering the right– to the left–hand side direction, programs such as the one in Figure 25 are generated, where behavioural problems can be found.

Observe that, before the transformation, the result of the method invocation in (*new ClassId_6().methodid_1()*) is 2 and, after the transformation, 0. This occurs because, before, as *methodid_1* is invoked on an instance of *ClassId_6*, which in turn invokes *methodid_0()*, using the qualifier *this*. In this case, the keyword *this* refers to the implementation of the method *methodid_0* in *ClassId_6*, which yields 2. On the other hand, after the transformation, *methodid_1()* invokes *methodid_0* of the super class of the *ClassId_6* class, which yields 0. Observe that the test itself (*new ClassId_6.methodid_1()*) does not need to be sophisticated to catch behavioural problems but instead the programs where these tests would be applied.

Law 4. ⟨move original method to superclass—illustrating a common refactoring practice⟩

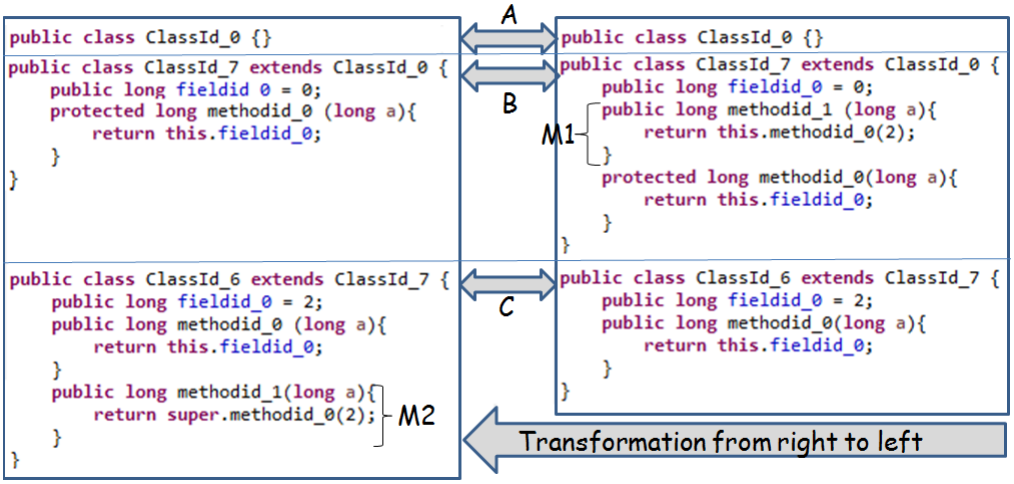


Figure 25 Classes generated according to the Law 4 specification.

class B **extends** A {		**class** B **extends** A {
ads		ads
$cnds$		$cnds$
mds		$rt\ \ m(pds)\ \ \{mbody\}$
}		mds
class C **extends** B {	$=_{cds,Main}$	}
ads'		**class** C **extends** B {
$cnds'$		ads'
$rt\ \ m(pds)\ \ \{mbody\}$		$cnds'$
mds'		mds'
}		}

provided

($\leftrightarrow$) *(1) private* attributes do not appear in *mbody*; (2) $m(pds)$ is not declared in any superclass of B in *cds*.

($\rightarrow$) *1) $m(pds)$ is not declared in mds; (2) mbody does not contain uncast occurrences of* **this** *nor expressions in the form* $((C)$**this**$).a$ *for any protected attribute a in* ads'*; (3)* $m(pds)$ *is not declared in any subclass of B in cds; (4)* $m(pds)$ *is not private.*

($\leftarrow$) (1) $m(pds)$ is not declared in mds'; (2) $D.m(e)$, for any $D \leq B$ and $D \not\leq C$, does not appear in $cds, Main, mds$ or mds'*;(3)* $this.method(e)$ *for any call for a method in mbody is replaced with* $super.method(e)$.

4.4.3.3 Analysis for Pull Up/Push Down Method Rule in (CORNELIO, 2004)

The Pull Up/Push Down Method (Rule 4.4 in (CORNELIO, 2004)) has already been analysed in (PALMA, 2015) (reproduced below as Rule 4.1). We have analysed both specifications. Firstly, we created an Alloy Model to represent the specification in (CORNELIO, 2004): the main

predicates of the transformation–specific model are the ones depicted in Codes 4.3 and 4.4, which describe, respectively, the transformation from the left– to the right–hand side and vice–versa, according to what is defined in the specification.

Code 4.3 Predicate that captures the transformation from the left– to the right–hand side defined in the Rule 2.1 (see Section 2.1.3).

```
1  pred rule44LR[a,b,c: ClassId, m: Method, left,right: Program] {
2      a != b
3      b != c
4
5      let leftCds = left.classDeclarations,
6          rightCds= right.classDeclarations,
7          al = a.leftCds,
8          ar = a.rightCds,
9          bl = b.leftCds,
10         br = b.rightCds,
11         cl = c.leftCds,
12         cr = c.rightCds {
13
14         // right description
15         m in (ar.methods)
16         br.extend = a
17         cr.extend = a
18
19         // proviso (<->) (1)
20         noSuperOrPrivateAttributesInM[right,mR,b]
21
22         // proviso (->) (1)
23         methodIsNotDeclaredInTheClass[m,al]
24
25         // proviso (->) (2)
26         mIsNotDeclaredInAnySuperClassOfTheClassInParam[a,m,left,
               right]
27
28         //left description
29         bl.extend = a
30         cl.extend = a
31         m in (bl.methods)
32         m in (cl.methods)
33
34
35        //equivalence between the left- and the right-hand sides
36        al.extend = ar.extend
37        leftCds = rightCds ++ {a->al} ++ {b->bl} ++ {c->cl}
38
39        al.fields = ar.fields
```

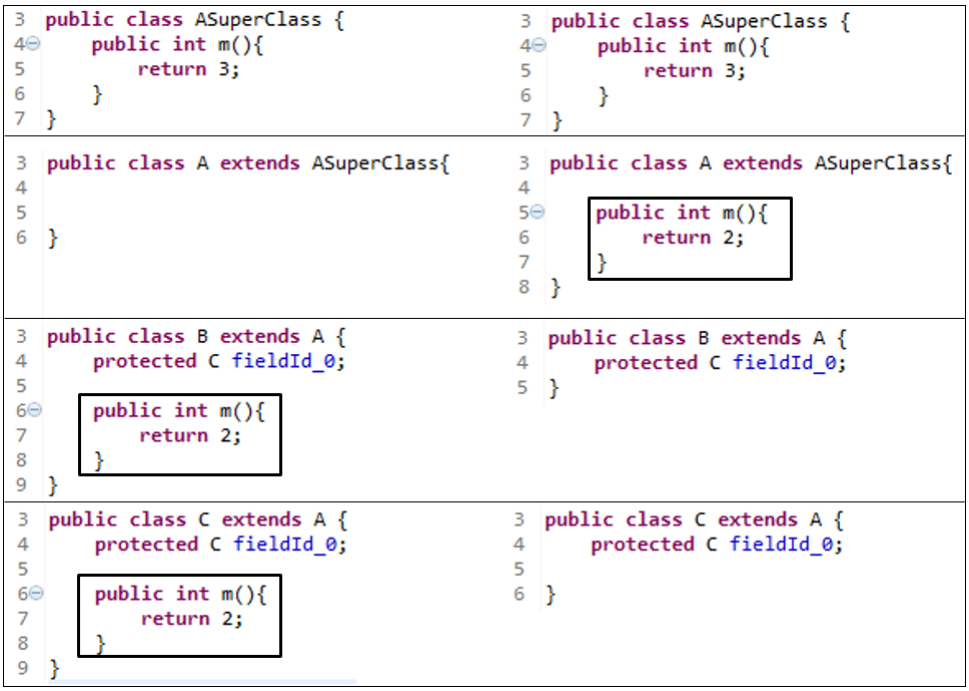


Figure 26 Example of a program generated with behavioural problem using Rule 2.1 specification.

```
40        bl.fields = br.fields
41        cl.fields = cr.fields
42
43        al.methods = ar.methods - m
44        bl.methods - m = br.methods
45        cl.methods - m = cr.methods
46
47        a.~((left.classDeclarations).extend) = a.~((right.
             classDeclarations).extend)
48        b.~((left.classDeclarations).extend) = b.~((right.
             classDeclarations).extend)
49        c.~((left.classDeclarations).extend) = c.~((right.
             classDeclarations).extend)
```

Similarly to the verification by the work in (PALMA, 2015), we conclude the proviso $(\rightarrow)$ (2) in Rule 2.1 and represented in line 26, Code 4.3, is also necessary from the right– to the left–hand side direction. Suppose method *m()* is being pushed down from the class *A* to its subclasses *B* and *C*. Figure 26 shows an example of program before (right) and after the transformation following the specification in Rule 2.1. An simple test like *new A().m()* evidences behavioural problems with different results in the method executions. Thus, Rule 2.1 present behavioural problems in every call to method *m* where the class *A* is the target.

Hence, this proviso $((\rightarrow)$ (2)) was moved to be the proviso $(\leftrightarrow)$ (2) in Rule 4.1. Likewise, it is necessary to add its corresponding predicate *mIsNotDeclaredInAnySuperClassOfTheClass-InParam* in both predicates *rule44LRModified* and *rule44RLModified*, line 23, Codes 4.5 and 4.6.

This is the only necessary modification with regard to the right– to the left–hand side direction.

Code 4.4 Predicate that captures the transformation from the right– to the left–hand side defined in Rule 2.1 (see Section 2.1.3).

```
1  pred rule44RL[a,b,c: ClassId, m: Method, left,right: Program] {
2     a != b
3     b != c
4
5     let leftCds = left.classDeclarations,
6         rightCds= right.classDeclarations,
7         al = a.leftCds,
8         ar = a.rightCds,
9         bl = b.leftCds,
10        br = b.rightCds,
11        cl = c.leftCds,
12        cr = c.rightCds {
13
14        // right description
15        m in (ar.methods)
16        br.extend = a
17        cr.extend = a
18
19        // proviso (<->) (1)
20        noSuperOrPrivateAttributesInM[right,mR,b]
21
22        // proviso (<-) (1)
23        methodIsNotDeclaredInTheClass[m,br]
24        methodIsNotDeclaredInTheClass[m,cr]
25
26        // proviso (<-) (2)(3)
27        forbidsAccessToMethodM[b,c,m,right]
28
29        //left description
30        bl.extend = a
31        cl.extend = a
32        m in (bl.methods)
33        m in (cl.methods)
34
35
36        //equivalence between the left- and the right-hand sides
37        al.extend = ar.extend
38        leftCds = rightCds ++ {a->al} ++ {b->bl} ++ {c->cl}
39
40        al.fields = ar.fields
41        bl.fields = br.fields
42        cl.fields = cr.fields
```

Rule 4.1 ⟨*Pull Up/Push Down Method*⟩—Rule 2.1 in (PALMA, 2015)

```
1  class A extends D
2    ads_a;
3    mts_a
4  end
5  class B extends A
6    ads_b;
7    meth m ≙ (sig • c')
8    mts_b
9  end
10 class C extends A
11   ads_c;
12   meth m ≙ (sig • c')
13   mts_c
14 end
```

$=_{cds,c}$

```
1  class A extends D
2    ads_a;
3    meth m ≙ (sig • c')
4    mts_a
5  end
6  class B extends A
7    ads_b;
8    mts_b
9  end
10 class C extends A
11   ads_c;
12   mts_c
13 end
```

provided

($\leftrightarrow$) .1 **super** and private fields do not appear in c'; and
.2 m is not declared in any superclass of A in cds.

($\rightarrow$) .1 m is not declared in mts_a, and can only be declared in a class N, for any $N \leq A$, if it has signature sig;
.2 there are no occurrences of $self.f$ in c' for any field f declared both in ads_b and ads_c; and
.3 if there is some occurrence of $self.m_1$ in c' for some method m_1 declared both in mts_b and mts_c, m_1 must be declared in A or any superclass.

($\leftarrow$) .1 m is not declared in mts_b or mts_c;
.2 **super**.m does not appear in mts_b or mts_c nor in any class N such that $N \leq A$ and $N \not\leq B$ or $N \not\leq C$; and
.3 $N.m$, for any $N \leq A$ and $N \not\leq B$ or $N \not\leq C$, does not appear in cds, c, mts_a , mts_b or mts_c.

```
43
44        al.methods = ar.methods - m
45        bl.methods - m = br.methods
46        cl.methods - m = cr.methods
47
48        a.~((left.classDeclarations).extend) = a.~((right.
             classDeclarations).extend)
49        b.~((left.classDeclarations).extend) = b.~((right.
             classDeclarations).extend)
50        c.~((left.classDeclarations).extend) = c.~((right.
             classDeclarations).extend)
```

In addition, concerning the left– to the right–hand side direction, we verify static seman-

tics errors when the method is pulled up if the body of this method refers to a field defined in classes *B* and *C* but not defined in class *A*. This is also verified in (PALMA, 2015) and, for this reason, there is an added proviso $(\rightarrow)$ (2) in Rule 4.1. We then create a predicate to represent it, named *noSelfFieldsInM* (and added it in a modified version of the predicate *rule44LR*, named predicate *rule44LRModified* in Code 4.6), that should be applied twice for classes *B* and *C*, according to what is established by the proviso.

Likewise, a static semantics error can appear when a method, only defined in classes *B* and *C*, is referred in the body of the method *m*. Thus, the proviso $(\rightarrow)$ (3) was also added in Rule 4.1. This inclusion does not cause behavioural problems. For instance, consider that the condition stated by the proviso holds, meaning that there is a call to a method m_1, with the qualifier *this* (or *self*), inside method *m* body. Besides, suppose m_1 is declared in both classes *B* and *C*, and in class *A* or any of its superclass. In a supposed call to method *m*, the target for this call can only be class *B* or *C*, since a well–formed program is assumed as a premise. Then, even when the method is pulled up, the qualifier *this* (or *self*) in the method call will refer to the method m_1 in class *B* or *C*, preserving the same behaviour before the transformation is applied. Lines 33 and 34 from Code 4.5 represents the proviso $(\rightarrow)$ (3) in Rule 4.1.

Code 4.5 Predicate that captures the transformation from the left to the right–hand side in Rule 4.1.

```
1  pred rule44LRModified[a,b,c: ClassId, m: Method, left,right:
       Program] {
2      a != b
3      b != c
4
5      let leftCds = left.classDeclarations,
6          rightCds= right.classDeclarations,
7          al = a.leftCds,
8          ar = a.rightCds,
9          bl = b.leftCds,
10         br = b.rightCds,
11         cl = c.leftCds,
12         cr = c.rightCds {
13
14         //right description
15         m in (ar.methods)
16         br.extend = a
17         cr.extend = a
18
19         // proviso (<->) (1)
20         noSuperOrPrivateAttributesInM[right,mR,b]
21
22         // proviso (<->) (2)
23         mIsNotDeclaredInAnySuperClassOfTheClassInParam[a,m,left,
               right]
```

```
24
25        // proviso (->) (1)
26        methodIsNotDeclaredInTheClass[m,al]
27
28        // provisos (->) (2)
29        noSelfFieldsInM[left,m,b]
30        noSelfFieldsInM[left,m,c]
31
32        // proviso (->) (3)
33        all m_:Method | m_ != m && thereIsAMethodInvocationInM[m_
             , m] =>
34        m_ in (a.leftCds.*(extend.leftCds)).methods
35
36        // left description
37        bl.extend = a
38        cl.extend = a
39        m in (bl.methods)
40        m in (cl.methods)
41
42
43        // equivalence between the left- and the right-hand
             sides
44        al.extend = ar.extend
45        leftCds = rightCds ++ {a->al} ++ {b->bl} ++ {c->cl}
46
47        al.fields = ar.fields
48        bl.fields = br.fields
49        cl.fields = cr.fields
50
51        al.methods = ar.methods - m
52        bl.methods - m = br.methods
53        cl.methods - m = cr.methods
54
55        a.~((left.classDeclarations).extend) = a.~((right.
             classDeclarations).extend)
56        b.~((left.classDeclarations).extend) = b.~((right.
             classDeclarations).extend)
57        c.~((left.classDeclarations).extend) = c.~((right.
             classDeclarations).extend)
```

Code 4.6 Predicate that captures the transformation from the right- to the left-hand side in Rule 4.1.

```
1 pred rule44RLModified[a,b,c: ClassId, m:Method, left,right:
     Program] {
2     a != b
3     b != c
```

```
4
5     let leftCds = left.classDeclarations,
6         rightCds= right.classDeclarations,
7         al = a.leftCds,
8         ar = a.rightCds,
9         bl = b.leftCds,
10        br = b.rightCds,
11        cl = c.leftCds,
12        cr = c.rightCds {
13
14        // right description
15        m in (ar.methods)
16        br.extend = a
17        cr.extend = a
18
19        // proviso (<->) (1)
20        noSuperOrPrivateAttributesInM[right,mR,b]
21
22        // provisos (<->) (2)
23        mIsNotDeclaredInAnySuperClassOfTheClassInParam[a,m,left,
              right]
24
25        // proviso (<-) (1)
26        methodIsNotDeclaredInTheClass[m,br]
27        methodIsNotDeclaredInTheClass[m,cr]
28
29        // provisos (<-) (2)(3)
30        forbidsAccessToMethodM[b,c,m,right]
31
32        //left description
33        bl.extend = a
34        cl.extend = a
35        m in (bl.methods)
36        m in (cl.methods)
37
38
39        // equivalence between left- and right-hand sides
40        al.extend = ar.extend
41        leftCds = rightCds ++ {a->al} ++ {b->bl} ++ {c->cl}
42
43        al.fields = ar.fields
44        bl.fields = br.fields
45        cl.fields = cr.fields
46
47        al.methods = ar.methods - m
48        bl.methods - m = br.methods
```

```
49        cl.methods - m = cr.methods
50
51        a.~((left.classDeclarations).extend) = a.~((right.
             classDeclarations).extend)
52        b.~((left.classDeclarations).extend) = b.~((right.
             classDeclarations).extend)
53        c.~((left.classDeclarations).extend) = c.~((right.
             classDeclarations).extend)
```

Observe that the other lines of the predicate *rule44LRModified* remain the same as the ones in *rule44RLModified*. The only predicate included in these predicates that were not presented before was the *noSelfFieldsInM* predicate (see Code 4.7). It verifies if any statement of a method, passed as parameter, owns a *FieldAccess* whose *pExp* relation is not in the set of *this* instances and whose field identifier in its *id_fieldInvoked* relation be a field of the class passed as parameter. This is done through auxiliary predicates as can be seen in Code 4.7. Firstly, the *AssignmentExpression* statement is verified (line 3, Code 4.7) through the predicate *noSelfFieldsInAssignment* (see Code 4.8). Secondly, the *MethodInvocation* statement (line 5, Code 4.7) is verified through the predicate *noSelfFieldsInMethodInvocation* (see Code 4.11).

Code 4.7 Predicate *noSelfFieldsInM*.

```
1 pred noSelfFieldsInM[p:Program, m:Method, class:ClassId]{
2     all st:Statement | st in univ.(m.body) =>
3     (st in AssignmentExpression => noSelfFieldsInAssignment[p,
         st,m,class])
4     &&
5     (st in MethodInvocation => noSelfFieldsInMethodInvocation[p,
         st,class])
6 }
```

In the predicate *noSelfFieldsInAssignment* (see Code 4.8), it is checked if there are any accesses to fields in an *AssignmentExpression* statement (in a class passed as parameter). To do this, firstly its left–hand side expression is checked (through the predicate *noSelfFieldsInLeftHandSide*, line 2). Secondly, its right–hand side expression is also checked. In our OO model used for analysing this rule, a right–hand side expression of an *AssignmentExpression* statement can be a *LiteralValue* or a *MethodInvocation*. Thus, only the second case is checked: if there are any accesses to fields in the class passed as parameter, through the predicate *noSelfFieldsInMethodInvocation* (lines 3 and 4).

Code 4.8 Predicate *noSelfFieldsInAssignment*.

```
1 pred noSelfFieldsInAssignment[p:Program, ae:
     AssignmentExpression, m:Method, class:ClassId]{
2     noSelfFieldsInLeftHandSide[p,ae.pExpressionLeft,class]
3     ae.pExpressionRight in MethodInvocation =>
4     noSelfFieldsInMethodInvocation[p,ae.pExpressionRight,class]
```

```
5 }
```

In predicate *noSelfFieldsInLeftHandSide* (see Code 4.9), an auxiliary predicate, named *existsFieldInClass*, is used to check for a field identifier in the class passed as parameter.

Code 4.9 Predicate *noSelfFieldsInLeftHandSide*.

```
1 pred noSelfFieldsInLeftHandSide[p:Program, lhs:LeftHandSide,
     class:ClassId]{
2     lhs in FieldAccess && existsFieldInClass[p,lhs.
        id_fieldInvoked,class] =>
3     lhs.pExp !in this_
4 }
```

Code 4.10 Predicate *existsFieldInClass*.

```
1 pred existsFieldInClass[p:Program, fieldId:FieldId, class:
     ClassId]{
2     some f: Field | f in class.(p.classDeclarations).fields &&
3     f.id = fieldId
4 }
```

Code 4.11 Predicate *noSelfFieldsInMethodInvocation*.

```
1 pred noSelfFieldsInMethodInvocation[p:Program, mi:
     MethodInvocation, class:ClassId]{
2     #(mi.realParam) != 0 && mi.realParam in FieldAccess =>
3     noSelfFieldsInLeftHandSide[p,mi.realParam,class]
4 }
```

4.4.3.4 Analysis for Rule 3 in (QUAN; ZONGYAN; LIU, 2008)

Analysing the Alloy Model for Rule 3 (Section 3.2.5) and, more specifically, by analysing the resulting instances of this Alloy model, we discover some static semantics errors due to the absence of some conditions in Rule 3: item 2 in the beginning of Section 4.4. Firstly, the most basic problem that causes a static semantics error is not to verify if there is any method with the same *id* and parameters of the method *m* being moved, *before* moving it. Thus, a condition with this restriction in Rule 3 is missing and consequently the corresponding predicate to guarantee this restriction in Code 3.48—see Chapter 3.

Secondly, suppose we have a call to method *m* in the body of another method, which is also in class *M*—see Figure 27, which depicts an example of the application of Rule 3 in three classes, *M*, *N* and *B*, where *N* and *B* are *M* subtypes. Despite the example being in Java, it also applies to rCOS since all OO features presented in Figure 27 also exist in rCOS. The referred access does not violate the specification, that says ”m is only used locally in M.”. The access in question is illustrated in line 12 of class *M* on the left–hand side, Figure 27. According to

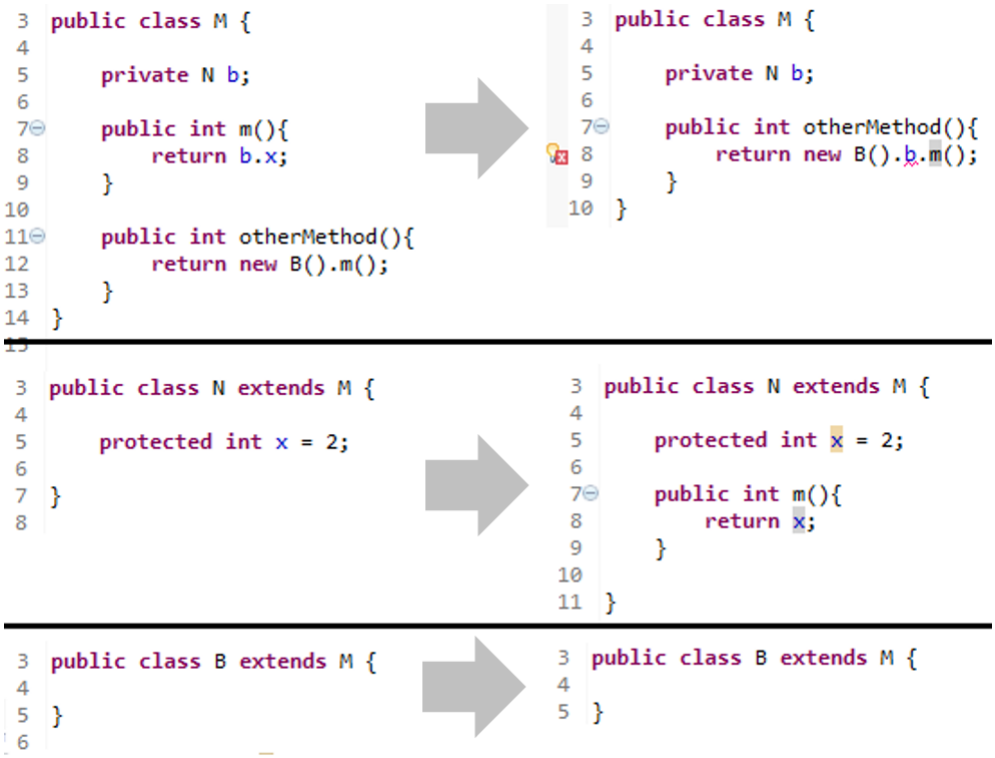


Figure 27 Example of programs where Rule 3 was applied.

the substitution *ops* in the Rule 3 specification, we have to replace every access to *m* by *b.m*() when method *m* is moved to class *N*—see line 8 of class *M* on the right–hand side. However, this expression will not compile if the attribute b is private; it can only be a public or protected attribute. Hence, this condition needs to be included in the specification. For instance, in the initial phrase we have to change for "Let *N* be a public attribute of class *M*". In addition, it is necessary to add the expression *&& f.acc !in private_* in line 16, Code 3.48, to complete the premise required for the refinement and avoid this kind of error.

However, the replacement defined in *ops* turns this rule specification the most complicated one—considering the translation to Alloy—so far. As already mentioned in Section 3.2.5, the *pExp* relation of a *MethodInvocation* needs to also be a *FieldAccess*. Thus a *FieldAccess* type needs to be a *PrimaryExpression* subsignature, beyond *this*, *super* and *newCreator* types. This generates a recursive situation in *FieldAccess* type, since this type also owns a *pExp* relation of type *PrimaryExpression*.

As mentioned in Section 3.1.1, it is difficult to deal with recursive predicates in Alloy. Because of this limitation, although we have found some errors in this rule, we had to limit our analysis and the potential discovery of more errors. In addition, it is necessary to be careful in the correspondence of classes *M before* and *after* the transformation, since every method where the replacement defined in *ops* is necessary needs to change its internal body. So in this case we have different but corresponding methods (with same *id* and *parameters*, but different bodies), and the other ones (where the replacement is not necessary) are equal, with only a different

context, passed as a predicate parameter (that is, the *Program* parameter which indicates the context before or after the transformation). However, the execution of this correspondence (and consequently of the predicates that do it) is compromised since we cannot do the *ops* replacement completely, as just explained. In spite of this limitation, we were able to find many errors related with this rule, but it would be possible to find even more if we were able to specify it entirely.

Finally, a dynamic error can occur if b has not been initialized, i.e. the attribute *B* is *null*. There is no restriction or premise in the rule specification that ensures that *B* should have been initialized in class *M* and thus the replacements defined in *ops* can throw a null pointer exception.

4.4.3.5 Analysis for Rule 6 in (QUAN; ZONGYAN; LIU, 2008)

Rule 6 specification in (QUAN; ZONGYAN; LIU, 2008) says *"all attributes used in m() are in M"* as a premise. This sentence leaves holes that can provoke behavioural problems—item 3 in the beginning of Section 4.4. Suppose for instance a field that is referenced by the method *m()* in classes *N1* and *N2*, but this field is also in class *M*. Thus, the sentence is not violated. However, when the method *m()* is moved to class *M*, a behavioural problem can occur since the field, also in class *M*, can assume a different value. We solve this ambiguity by modifying the predicate *lhsOnlyRefersToFieldsInM* in Code 3.71 by the one we call *lhsOnlyRefersToFieldsInMCorrected*, in Code 4.12. Lines 10 and 16 were added to guarantee that the field does not exist in neither class *N1* nor *N2* (possible returns to the function *whichFieldIs*) but only in class *M* (the next class after classes *N1* and *N2* in the return of the function *whichFieldIs*).

Code 4.12 Predicate that ensures the field used in *FieldAccess* expression is from class *M*.

```
1  pred lhsOnlyRefersToFieldsInMCorrected[p: Program,
       correspondingNClass,correspondingMClass: Class, lhs:
       LeftHandSide] {
2
3      let
4      fieldInExprName = whichFieldIs[p,lhs.pExp.name,
          correspondingNClass],
5      fieldInFieldInvoked = whichFieldIs[p,lhs.id_fieldInvoked,
          correspondingNClass]{
6
7      (lhs in FieldAccess && lhs.pExp in ExpressionName =>
8      (#fieldInExprName = 1 && fieldInExprName.type !in Long_ &&
9       lhs.pExp.name in (correspondingMClass.fields).id &&
10      lhs.pExp.name !in (correspondingNClass.fields).id ))
11
12     (lhs in FieldAccess && lhs.pExp !in ExpressionName =>
13     (lhs.pExp !in super && #fieldInFieldInvoked = 1 &&
14      fieldInFieldInvoked.type !in Long_ &&
15      lhs.id_fieldInvoked in (correspondingMClass.fields).id &&
16      lhs.id_fieldInvoked !in (correspondingNClass.fields).id ))
```

```
17      }
18  }
```

A similar problem occurs when there are calls for methods in *m* that are in classes *M* and *N* with the same identifiers. This fact does not violate what is said in the rule (i.e. *"all attributes used in m() are in M"*), but also causes a behavioural problem for the same reasons already presented for fields. Hence, it is necessary some additional restrictions in this premise. Firstly, we can enhance it to *"all attributes and methods used in m() are only in M, not in N"*. As a consequence, the *methodOnlyRefersToAttributesInM* predicate, Code 3.70 changes to the *methodOnlyRefersToAttrAndMethsInM* predicate, shown in Code 4.13. The difference is the presence of the just mentioned predicate *lhsOnlyRefersToFieldsInMCorrected* and also of the *methodOnlyRefersToMethodsInM* predicate (shown in Code 4.14).

Code 4.13 Predicate that ensures every field or method used in m() is a field or method from class M.

```
1  pred methodOnlyRefersToAttrAndMethsInM[method:Method,
       correspondingNClass,correspondingMClass:Class, p:Program] {
2
3      all st:Statement | st in univ.(method.body) =>
4      (st in AssignmentExpression =>
5       lhsOnlyRefersToFieldsInMCorrected[p,correspondingNClass,
          correspondingMClass,st.pExpressionLeft]) &&
6
7      (st in AssignmentExpression && st.pExpressionRight in
         MethodInvocation =>
8       (methodOnlyRefersToMethodsInM[p,correspondingNClass,
          correspondingMClass,st.pExpressionRight])) &&
9
10     (st in MethodInvocation =>
11      methodOnlyRefersToMethodsInM[p,correspondingNClass,
          correspondingMClass,st])
12
13 }
```

Observe that the predicate *methodOnlyRefersToMethodsInM* includes the checking done in the real parameters of the *MethodInvocation* (see line 17). This was also done in the predicate *methodOnlyRefersToAttributesInM*, Code 3.70 (lines 7 and 10). In addition, the predicate *methodOnlyRefersToMethodsInM* does the checking in the *pExp* relation (type *PrimaryExpression*) of the *MethodInvocation* (see lines 7 to 10, Code 4.14), in the case this expression refers to an expression of type *ExpressionName*. This checking is similar to what is done in the predicate *lhsOnlyRefersToFieldsInMCorrected* (lines 7 to 10, Code 4.12).

In spite of this correction, the premise remains incomplete. There are still other errors in Rule 6. For instance, even if the method *m()* accesses only fields or methods from class *M*, but if this access is through a *super* expression, when the method is moved to class *M*, then a

compilation error can occur, if the super class of *M* does not have a similar field or method, or a behavioural problem, otherwise. This scenario is avoided in Law 3 through the existence of the proviso ($(\leftrightarrow)$ (1)). Hence, we include in the predicates *lhsOnlyRefersToFieldsInMCorrected* and *methodOnlyRefersToMethodsInM* the restriction not to access fields or methods through a *super* expression (see lines 13 and 13, respectively, from Codes 4.12 and 4.14).

Besides, as just mentioned, if there is a method *m()* also defined in a super class of *M*, a behavioural problem can occur. Thus, similarly as done in Law 2, it is necessary to include this restriction in Rule 6, likewise the proviso ($(\leftrightarrow)$ (2)) in Law 3. The predicate *mIsNotDeclaredInAnySuperClassOfTheClassInParam*, defined in Code 3.43 ensures this restriction and it can be included in Code 3.69, in line 22, for example.

Code 4.14 Predicate that checks if the MethodInvocation refers to a field or method from class *M*.

```
1  pred methodOnlyRefersToMethodsInM[p:Program,
       correspondingNClass, correspondingMClass:Class, mi':
       MethodInvocation] {
2
3      let
4      field = whichFieldIs[p,mi'.pExp.name,correspondingNClass],
5      method = whichMethodIs[p,mi'.id_methodInvoked,
           correspondingNClass]{
6
7      mi'.pExp in ExpressionName =>
8      (#field = 1 && field.type !in Long_ &&
9      mi'.pExp.name in (correspondingMClass.fields).id &&
10     mi'.pExp.name !in (correspondingNClass.fields).id )
11
12     mi'.pExp !in ExpressionName =>
13     (mi'.pExp !in super && #method = 1 &&
14     method.id in (correspondingMClass.methods).id &&
15     method.id !in (correspondingNClass.methods).id )
16
17     #(mi'.realParam) = 1 =>
18     lhsOnlyRefersToFieldsInM[p,correspondingNClass,
           correspondingMClass,mi'.realParam]
19
20     }
21 }
```

Another restriction, not included as a premise in this Rule, but already discussed for Rule 3, is the requirement for the non existence of a method in class *M* with same identifier and formal parameters as method *m()*, not to cause a compilation error. This restriction was also contemplated in Law 3 with the proviso ($(\rightarrow)$ (1)).

4.4.3.6 Other Analyses

We have also explored several variations aiming at generating syntactic errors and in all cases counter examples were generated, which gives some evidence that our Static Semantics Validator embedded in Alloy indeed captures the syntax and static semantics of a subset of Java.

For instance, suppose we change the specification for Law 1 to Law 5. If we observe the only change is in the proviso ($\leftarrow$) of Law 1. More specifically, *D*, mentioned in this proviso, assumes the value of a class of type *B* when it should not. This is because class D belongs to the set of the *B's* subtype—it is *B* itself. Thus, the body in the predicate in Code 3.27, which is referenced in Code 3.26, has to change to the one presented in Code 4.15. In this new body, class B (or C) is not included in its own subtype. Only their subclasses are because there we are using transitive closure ($\hat{}$) instead of a reflexive transitive closure ($*$).

Mapping the situation to our model, a *FieldAccess*, using its *pExp* relation of type *newCreator* (a subtype of *PrimaryExpression*), whose *cf* relation, in turn, is of class *B* would be generated trying to access the specific field (relation *fieldInvoked* of *FieldAccess* signature) being moved by the transformation.

Law 5. ⟨move attribute to superclass⟩

class *B* **extends** *A* {		**class** *B* **extends** *A* {
ads		**public** *T f*; *ads*
cnds		*cnds*
mds		*mds*
}		}
class *C* **extends** *B* {	$=_{cds,Main}$	**class** *C* **extends** *B* {
public *T f*; *ads′*		*ads′*
cnds′		*cnds′*
mds′		*mds′*
}		}

provided

(→) *The attribute name f is not declared by the subclasses of B in cds;*

(←) *(1) The attribute name f is not declared in ads'; (2) D.f, for any D ↾ B and D ⋦ C, does not appear in cds, Main, cnds, cnds', mds, or mds'*

Code 4.15 Mutation of the predicate in Code 4.15

```
1 pred firstIsSubtypeOfTheSecondOneClassMutate[p:Program, first,
     second:ClassId] {
2     let secondSubClasses = second.^~((p.classDeclarations).
        extend) {
3         first in secondSubClasses
4     }
5 }
```

This modification causes the Alloy Analyzer to find counter examples when the model in Code 4.1—Static Semantics Validator—is executed. Thus, programs such as the ones presented in Figure 28 are generated. This program shows the class *ClassId_4* (*left*) trying to access the field (*fieldid_2*) (It is no longer available because it was moved from right to the left–hand side).

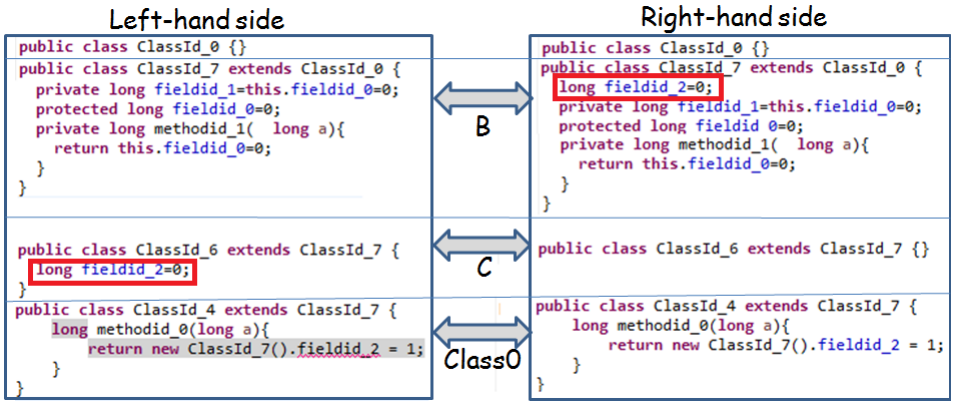


Figure 28 Classes generated showing anomalies in the program transformation.

4.4.4 Threats and Validity

Next we present the threats to validity of our evaluation.

4.4.4.1 Construct Validity

We reported to almost all the transformation specifiers the errors found (except the specifier of the Law 4—Push Down Method in (SCHAFER, 2010)). All of them (reported) accepted the errors.

4.4.4.2 Internal Validity

The scope limitation certainly hide possibly detectable errors. At first, we have the scope defined by the type elements defined in our OO model. Secondly, there is the scope defining a bound on the size of each of the top-level signatures in the Alloy model, that will be submitted to the Alloy Analyzer. In the former case, Java programs are only generated using the types in our OO model. For instance, there is no package (this type was not considered for simplification purposes, as already discussed in Chapter 3).

Besides, some specifications analysed use type elements in their provisos that were not defined in our OO model. One example is the *cast* type, used in Law 2—Move Method in (DUARTE, 2008). This type was not included in our OO model due to its complexity; for instance, type checkings would be required to be done simulating a programs execution. This is very difficult (maybe impossible) to do using the Alloy infrastructure.

Another point that limit some more detectable errors to be found is the Alloy limitation in dealing with recursion as explained thorough this work. For instance, specifications such as Rule 3 can not be analysed entirely due to this restriction (see Section 4.4.3.4). In general, when substitutions are required, recursion is necessary. Even when it is not necessary, the Alloy model become more complex because it is necessary to apply these substitutions in the elements (of the Alloy model) involved in the transformation, according to what is described in the specification.

In addition, we observe memory leak problem in the *sat4j* solver used by the Alloy Analyzer, that difficulties the detection of more transformation specification faults, mainly the ones related to behavioural problems. Thus, the solver was not able to finish the generation of all of the Alloy instances. As a consequence, we had to limit the number of instances analysed. Actually, we faced this problem in all of the specifications analysed and had to do the behavioural analysis manually, using the Alloy Analyzer tool itself (with our Alloy Validator model), instead of our Alloy–To–Java translator tool. A possible solution not to cause memory leak problems or decrease the analysis time would be to decrease the scope bound limitation of each type element involved in the transformation or even remove some of them. However, for each transformation specification we already worked with the minimal type elements (and their corresponding bounds) required for doing the analysis.

Memory problems are also observed when (1) specifications did not present static semantics problems or (2) specifications were fixed and seemed not to present static semantics problems. When there is an error in the specification (and as a consequence in the transformation–specific model that represent it), the Alloy Analyzer returns counter examples (in case of the Static Semantics Validator model) rapidly. However, when there is no counter example, Alloy Analyzer delays in returning that no counter examples have been found, mainly in the cases the models are more complex (i.e. with more types, facts and predicates). In these cases, after some days waiting, a memory error is returned by the Alloy Analyzer.

5

RELATED WORK

Most related works focus on providing transformation (more commonly, refactorings) implementations, and are usually available as plug–ins to an IDE tool. In these works, the transformations are typically validated using test suites, provided by the tool itself or by some IDE. Due to the absence of transformation precise specifications, some works such as (OVERBEY; JOHNSON, 2011; OVERBEY M. J. FOTZLER; JOHNSON, 2011; SCHAFER, 2010) take the initiative to also specify these refactorings in a way to ease implementation. On the other hand, the works in (SOARES, 2015; SABINO, 2016) focus mainly on evaluation of the implementations provided by (OVERBEY; JOHNSON, 2011; OVERBEY M. J. FOTZLER; JOHNSON, 2011; SCHAFER, 2010) based on test input generators using, for example, the Alloy Analyzer; but the initiatives in (SOARES, 2015; SABINO, 2016) do not address specification or implementation of refactorings. The major contributions reported in (SOARES, 2015; SABINO, 2016) were to find many bugs in refactoring implementations available in IDEs (or refactoring engines) such as JRRT (EKMAN; HEDIN, 2007), Eclipse, NetBeans and IntelliJ.

However, a validation of transformation specifications, regardless of their implementations in a source language, is not addressed. As discussed along this thesis, our strategy focused on transformation *specification* validations, considering that a transformation specification is given as input. Using Alloy and the Alloy Analyzer, we simulate all possible inputs (according to a given scope of elements) matching a specification template which enables a transformation to be applied. At the same time, all the respective instances of programs are also generated and it is checked if the transformation does not cause static semantics or behaviour problems in them. So our main goal is transformation validation, rather than providing a transformation implementation that can be plugged into an IDE and used by developers. As a future work, we aim at providing a tool, where our strategy can be embedded, that offers a standard format of transformation specification, that can be easily implemented in a programming language.

In addition to the related work mentioned above, in this chapter we consider some additional approaches closely related to ours. The considered works are summarised in Table 3, where the lines represent each work being compared, and the columns represent the perspectives being compared. For instance, works such as (OVERBEY; JOHNSON, 2011; SCHAFER, 2010) propose transformation (in their case, refactoring) *specifications* (see first column in Table 3), whereas

we focus on validating transformations. The focus on validation is also the case for the works in (SOARES, 2015; SABINO, 2016) but they only validate the transformation implementations. Concerning transformation *implementation* (second column), the works (OVERBEY; JOHNSON, 2011; SCHAFER, 2010) adopt concrete programming languages whilst we use the Alloy formal notation for providing an implementation of each transformation specification. This point is not addressed by the works (SOARES, 2015; SABINO, 2016). On the other hand, all the works address transformation *validation* (third column). However, as we already mentioned, except for our case, only the transformation *implementation* is validated (not the *specification*). In some works such as (SCHAFER, 2010), the specification is adjusted as a consequence of the implementation validation, since when a fault is found in the implementation, the adjustment is propagated back to the specification. This can be done in the case of the specification provided by the authors in (SCHAFER, 2010) since it is done as a pseudocode, to ease the implementation of the refactoring. However, what is actually (or directly) validated is the transformation implementation. For this reason, we indicate in the transformation validation column the symbols *S* or *I* referring to the validation of specification or implementation, respectively.

With regard to the fourth column, named *Dependency of a particular source language*, it captures whether the program where the transformation is applied needs to be in a specific source language. In our strategy, this perspective does not apply since we are validating transformation specifications.

The column *internal model representation* refers to the specific model used to evaluate a specific transformation, regardless whether what is being evaluated is a specification or an implementation. Hence, in our case we work with Alloy instances, whilst the *differential precondition checking* proposed in (OVERBEY; JOHNSON, 2011) works with a program graph (i.e. AST). The other strategies consider programs (in general) as input to a refactoring engine implementation. The column *internal model generation* is intuitive and refers to which tool is used to generate the internal model representation.

Finally, the columns *Static Semantics* and *Dynamic* checks indicate which mechanisms are used to do the static semantics and dynamic validations, respectively, of the transformations being evaluated. Each perspective/column is further detailed when describing each one of the related work.

Many authors refer to preconditions incompleteness as the main source of program transformation misbehaviour. In this direction, the work in (OVERBEY; JOHNSON, 2011) presented a technique called *differential precondition checking*. They classified refactoring preconditions as ensuring input validity, compilability, and behaviour preservation. The first one checks if all inputs from the user are legal, hence it is possible to apply the transformation. Compilability checks if the resulting program (RS) will compile after the transformation is applied (the RS will meet all the syntactic and semantic requirements of the target language, according to a given scope). This property is similar to what our Static Semantics validator does. Finally, the preservation property checks if, when the transformation is performed and the RS is compiled

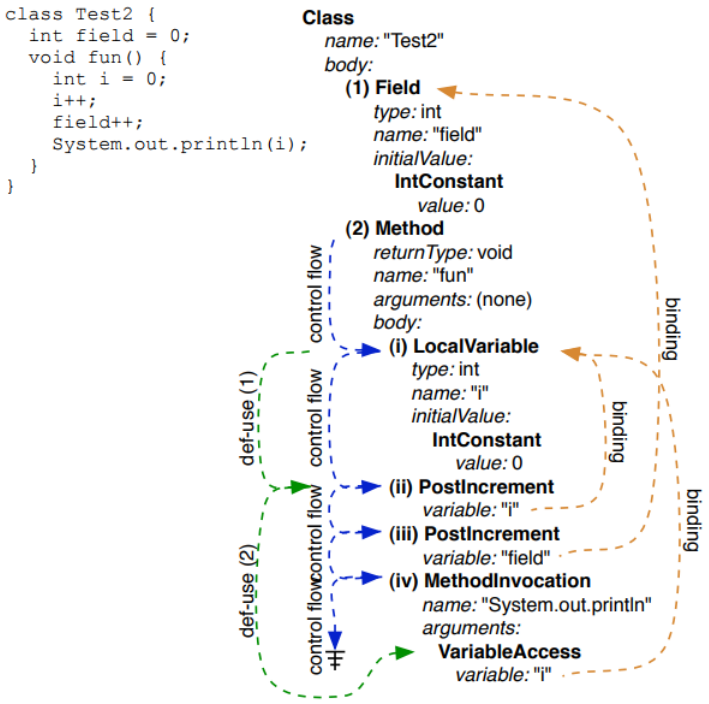


Figure 29 Example of the Java program and its corresponding program graph (OVERBEY; JOHNSON, 2011).

and executed, it will exhibit the same runtime behaviour as the original program. That is what our Dynamic validator does.

Overbey et al. (OVERBEY M. J. FOTZLER; JOHNSON, 2011) wrote detailed specifications of 18 refactorings in a technical report. Each specification describes both the traditional and the differential version of the refactorings, both at a level of detail sufficient to serve as a basis for implementation. They implemented a differential precondition checker and used it to implement differential refactorings in three refactoring tools—Fortran 95, PHP 5, and BC—, following their detailed specifications. Their checker can be placed in a library and reused in refactoring tools for many different languages. For the tools having traditional versions of the refactorings, they did a comparison between the results applying the traditional and differential versions—since they were able to reuse the existing test cases to test the differential implementations.

The referred checker builds a *semantic model* of the program prior to transformation, simulates the transformation, performs semantic checks on the modified program, computes a semantic model of the modified program, and then looks for differences between the two semantic models. The semantic model is represented by a *program graph* (see an example in Figure 29), which is an AST with semantic information such as name bindings, control flow, inheritance relationships and so forth. If the actual differences in the semantic models are all expected, then the transformation is considered to be behaviour preserving. The changes are applied to the user code only in this case.

Observe that our Alloy instances, which characterizes Java programs, have a similar (but more complete, see Figure 9) structure than the program graph presented in Figure 29. All of

our programs follow the pattern defined in our OO metamodel. Our SS or input programs are generated taking advantage of formal techniques through our Alloy infrastructure (see Chapter 3), that in turn benefit from the exhaustive generation of the Alloy Analyzer, according to a given scope. Likewise are our resulting (RS) programs, but these ones, in addition, are generated according to what is defined in the Alloy transformation–specific model and at the same time of SS programs (in a synchronized manner). Instead, the work in (OVERBEY; JOHNSON, 2011) generates an internal representation (an AST) to compare the two sides of a transformation, both from a syntactic and a semantic perspective. However, the AST representation with the respective annotations is not enough to ensure full semantic preservation since not all required semantic information is modeled in the AST. For instance, replace every instance of the constant 0 with the constant 1 would almost certainly break a program, but their analysis would not detect such a problem, since this change would not affect any edges in a typical program graph. They cite another example of bug (in this case a behaviour problem) in the Encapsulate Field refactoring that cannot be detected by their solution since doing so would require an interprocedural analysis and this cannot be modeled in their program graph. On the other hand, our *validate* step can detect behaviour problems by taking advantage of the exhaustive generation offered by the Alloy Analyzer (according to the corresponding Dynamic Alloy model defined) and an adequate testing campaign (our Java Test tool).

In addition, our checking whether the transformation causes any static semantics error in the resulting program is done through our Static Semantics Validator Alloy model (see Section 4.1). This model yields exactly the Alloy instances (characterizing programs) generated with static semantics errors. Instead of this, the technique defined in (OVERBEY; JOHNSON, 2011) detects the difference between the ASTs (representing the starting and resulting programs, respectively), as just mentioned, through the indication, in the first one, of which semantic information they expect to be preserved after the transformation. According to the authors themselves, this is another vulnerability point since it is up to the developer/user to determine what will be preserved or not and not all semantic information is modeled in a program graph. Besides, it is a challenge expressing which semantic information are expected to be preserved (same type and endpoint) because, since the AST has been modified and the endpoints are AST nodes, it is not easy determining what the same endpoints. The comparison between graphs is done by exploiting an isomorphism between graph nodes and textual intervals.

Our work stands out mainly due to the absence of transformation precise descriptions so one can validate or improve it using our strategy. In general, refactorings commonly implemented in modern IDEs are explained in terms of one or two examples. Describing a refactoring precisely is a difficult task and consequently the guarantee that the refactored program is always valid and behaves likewise the original program. This problem is also mentioned in (SCHAFER, 2010), which takes the initiative to specify some refactorings. The authors provide a high-level specification of common refactorings, in terms of an informal pseudocode notation, but precise enough to serve as the basis of an implementation. They use their own specification to implement

all of the refactorings specified—a case study is presented in which they specify and implement the refactorings Rename Variable, Inline Temp, and Extract Method. Whenever they discover a bug in their implementation, as it is based on a high–level specification (also provided by them), then they propagate the fix back to the corresponding specification. In this way, they argue that they not only improve a particular implementation but also deep their understanding of the refactoring itself. In this aspect, we do the same since we use our strategy to validate a transformation specification and consequently deep our understanding about it. The difference is that they validate their own specification (in pseudocode) and implementation whilst we validate the ones in various other works (in a language independent format as discussed in Chapters 2 and 3). The Alloy instances generated in our strategy help us to fix eventual bugs in our Alloy models and, when it is not the case, repair the specification and modify the corresponding Alloy model(s) accordingly in a continuous and interactive process. Note that after all our main goal is validating the transformation specification.

The refactoring specifications provided in (SCHAFER, 2010) do not follow the traditional precondition–based approach but are based on proposed concepts (and techniques) of dependency preservation, language extensions, and microrefactorings. The first is the notion of *dependencies*, which captures static semantics properties of the program to be refactored. For instance, there is a name binding dependency from every name to the declaration it accesses, and a flow dependency from every use of a variable to each of its reaching definitions. The authors provide a framework for tracking such dependencies over the course of a refactoring and for making sure that they are preserved in the output program. In our case, this is achieved using our Alloy infrastructure and the Alloy Analyzer since our resulting instances are generated at the same time as instances representing the starting programs (and according to the predicates specifying the specific transformation) and their preservation (concerning static semantics aspects) checked through our *wellFormedProgram* predicate. The second concept proposed in (SCHAFER, 2010) is created to overcome shallow problems caused by idiosyncrasies or lacking features of the object language, then their refactorings are formulated in terms of an enriched language with extensions that make the transformation easier to describe—in our case, Alloy is used as a language independent format to describe and specify the transformations and we structure the specifications in terms of Alloy types and predicates in a way that eases the understanding of transformations specified. Finally, they achieve reusability through their technique of microrefactorings since they decompose a complex refactoring into multiple smaller and simpler ones, which can help with verifying the correctness of critical parts of the specification—in our case the reusability is achieved through the use of common predicates in different Alloy transformation–specific models, since each predicate can represent provisos, conditions or substitutions that can commonly appear in different transformation specifications.

The refactoring engine in (SCHAFER, 2010) is implemented as an extension to the JastAddJ Java compiler (EKMAN; HEDIN, 2007) frontend. They can only process programs that successfully pass syntactic and semantic checks. They argue their implementation is more

compact than the Eclipse's one. This is partly due to JastAdd's aspect–oriented features that enable one to separate the essence of refactoring implementation from supporting code, but most of the reduction in size comes from the fact that complicated issues of static semantics preservation are handled by the underlying framework, not by every individual refactoring.

Besides, they do some adjustments in the input programs to enable the refactorings being applied, whilst Eclipse would refrain from doing so and instead reject the whole refactoring. Along their work, examples are given of the wrong output produced by refactoring engines such as Eclipse, IntelliJ and NetBeans. They argue that their refactoring engine is very reliable because, although not implementing all the additional features Eclipse provides, they can handle many programs where Eclipse produces wrong output and also they refactor many programs on which the Eclipse implementation has to give up.

However, the authors of (SCHAFER, 2010) point out two major weaknesses of their refactoring engine in comparison with Eclipse: it is currently not integrated into an IDE, and it is somewhat lacking in performance. The former is a design choice since their goal was to develop a working prototype, not an end–user tool. The latter they attribute partly to the general approach of decomposing refactorings into smaller units, but more importantly to the choice of implementation language.

In addition, they verify their implementation using both correctness proofs (SCHAFER, 2010; SCHAFER; EKMAN; MOOR, 2009) and their own test suite and the one for Eclipse and IntelliJ, which are publicly available. A general method for formalising reference attribute grammars in the theorem prover Coq is proposed. They present a verified implementation of the name binding framework for a subset of Java in the theorem prover Coq. Besides, a framework for verifying analyses and transformations on source language is presented, consisting of a specification formalism for source-level analyses and transformations, namely Circular Reference Attribute Grammars, and an embedding of that formalism in Coq. Their embedding is supported by tools that reduce the tedium work with a complex language definition and allow proofs to be conducted at a high level of abstraction. Later, the work in (SOARES, 2015) found bugs in the engine presented by (SCHAFER, 2010). The bugs were already fixed.

In (SELIM; CORDY; DINGEL, 2012), a model transformation testing process is surveyed and they conclude that the transformation testing process is composed by four phases: (1) a test case generation, where there is a test suite or a set of test models conforming to the source metamodel for testing the transformation of interest; (2) the assessment of the generated test suite; (3) the oracle function, which is the function that compares the actual output of a transformation with the expected output to evaluate the correctness of the transformation; (4) the assessment of the transformation—if discrepancies are found, then the tester can analyse the transformation and fix any faults accordingly.

The works in (SOARES, 2015; SABINO, 2016) seem to follow the transformation testing process defined in (SELIM; CORDY; DINGEL, 2012). The main goal of both works is checking implementation of refactoring engines. In the former, a tool, named JDolly, works as a test case

generator. Actually, it is a Java program generator, which involves an Alloy specification model (that can be mapped to a source metamodel in (SELIM; CORDY; DINGEL, 2012)) representing a subset of Java, from where Alloy instances (characterizing Java programs) are generated by the Alloy Analyzer. Furthermore, these Alloy instances are translated into Java programs. The ones not compiling are discarded and thus not submitted to the tools containing the refactoring engine implementations. It is important to emphasize that each one of these Java programs represents only the starting–hand side program (the program before the transformation) and the IDE plugin implements the transformation itself to generate the resulting–hand side. On the other hand, in our strategy, starting and resulting programs are generated concomitantly as Alloy instances by the Alloy Analyzer according to our Alloy models, as already discussed throughout this thesis. Afterwards, our Alloy–To–Java translator transforms these instances into Java programs. Besides, our OO metamodel was useful to validate not only Java transformation specifications, but specifications in other languages (rCOS, ROOL, and an object–oriented language with reference semantics presented in (PALMA, 2015)) that use, or have in common, OO features defined in our metamodel. Even considering a subset of the language and some simplifications, we observe that all the starting–hand side programs (where the predicate *wellFormedProgram* is applied to) used in the transformations analysed are 100% compilable, different from the model presented in (SOARES, 2015), which generates only 68,8% compilable programs.

In addition, in (SOARES, 2015), tests are generated by the SafeRefactor tool (SOARES *et al.*, 2010) to assert whether there are any behavioural discrepancies after the transformation—that is, the assessment phase in (SELIM; CORDY; DINGEL, 2012). SafeRefactor checks the common methods of the programs being compared and generates many tests, each one with some possible calls to these methods. The results are then compared and behavioural problems can be detected—the oracle function in (SELIM; CORDY; DINGEL, 2012).

Later, the work in (SABINO, 2016) enhances the one in (SOARES, 2015) and JDolly becomes Dolly. Besides providing a Java program generator with a better expressiveness due to the inclusion of abstract classes, abstract methods, and interfaces, which enabled more bugs to be discovered, Dolly also works as a C program generator. In addition, besides compilation errors and program behaviour changing, they also find overly weak and overly strong conditions in transformations as well as what they call as transformation issues. A transformation issue refers to incorrect transformations, regarding the refactoring transformation definition, that appears in the resulting program. As an example of a refactoring definition for the *Push Down Method* refactoring, we have: the transformation must remove the method from its original class, add the removed method in the subclass(es) of its original class, and update all calls to the refactored method; on the other hand, the transformation must not move the method when there is another method with the same signature in the target class, and apply the transformation when there is no subclass to push down the method.

The authors argue that transformations can be global and change parts of the code that they are not supposed to. When this scenario happens in practice, mainly for large subjects, the

user cannot be aware of all transformation changes. They also analyse additional transformations that must not be performed, such as removing an entity from the program. Thus, this is neither the case of an overly strong condition nor an overly weak condition, but it is a transformation issue in the refactoring implementation. Transformation issues are only checked when the resulting program compiles and preserves behaviour.

The work in (SABINO, 2016) proposes two oracles to identify transformation issues in refactoring implementations: Differential Testing (DT) and Structural Change Analysis (SCA) oracles. DT oracle compares the outputs of two refactoring implementations. For this, they implement a program that compares two Java programs concerning their Abstract Syntax Tree (AST). When the outputs compile and preserve the program behaviour, they check if they are different. If some difference is identified, they manually inspect the transformations to analyse if one of them (or both) has issues.

SCA detects transformation issues related to the refactoring definition. The way this oracle works is very similar to what is done in (OVERBEY; JOHNSON, 2011), where the ASTs of programs before and after the transformations are compared concerning their expectations. These expectations are named as refactoring definitions in (SABINO, 2016), as just explained. This is also very similar to what is done in (SCHAFER, 2010) but, in this last case, besides the process being performed just after the refactoring is applied, it is done to detect if the transformation caused any kind of problem, not restricted to transformation issues. Still, in (SCHAFER, 2010) their own implementation is checked and done according to their own transformation specification. If an unexpected result is found in the refactored program, then the implementation is fixed and the modification is propagated back to the specification.

Behaviour preservation is checked by them through SafeRefactorImpact, which is an oracle (enhanced from SafeRefactor) to identify failures related to behavioural changes. It generates test cases only for the methods impacted by a transformation. The time to test the refactoring implementation is reduced (if we compare to SafeRefactor in (SOARES *et al.*, 2010)) by proposing a technique to skip some consecutive test inputs, which enables finding some initial failures in a few seconds. However, some bugs are missed.

New subjects, including real case studies, considering Object-Oriented (OO) and Aspect-Oriented (AO) constructs are evaluated with respect to two new defined metrics (change coverage and relevant tests), time to evaluate a transformation, and detected behavioural change transformations. The evaluation in the context of Aspects showed evidence that the technique is useful to evaluate transformations in AspectJ programs. SafeRefactorImpact is also used to detect faults related to overly weak and overly strong preconditions as well as transformation issues.

Overly strong preconditions are detected in (SABINO, 2016) by disabling some preconditions. For each program generated by Dolly, they apply the transformation using the refactoring engine under test. Next, they collect the different kinds of messages reported by the refactoring engine when it rejects transformations. For each kind of message, they inspect the refactoring engine and manually identify the refactoring preconditions that can raise it. Thus, the refactoring

engine code is changed to allow disabling the preconditions that prevent the refactoring. If the engine, with some preconditions disabled applies the transformation, and preserves the program behaviour (according to SafeRefactorImpact), then they classify the set of disabled preconditions as overly strong. Finally, the authors in (SABINO, 2016) implement an automated issue categorizer to classify the outputs of DT and SCA oracles into different kinds of issues. It is based on the kinds of differences between the outputs (for DT oracle) and the kinds of refactoring definitions that the transformations do not follow (for SCA oracle).

However, in both works (SOARES, 2015; SABINO, 2016), the Alloy infrastructure is only used to generate the SS program inputs, if we consider the nomenclature used in our strategy (see Chapter 1). This is done through the Alloy model defining the subset of the corresponding language used to implement refactorings. On the other hand, the RS programs (using our nomenclature) are generated by the refactoring engines implementations instead of in the Alloy infrastructure as ours, likewise the SS programs. In addition, when a test fails, it is not detected if its origin is in the specification or in the implementation engine. In our approach, an Alloy instance, generated by the Alloy Analyzer, represents not only a Java program used as an input but a pair of programs, before and after the transformation, following what was specified in the transformation specification, translated into an Alloy transformation–specific model. Thus, it is possible to obtain program examples of the transformation failure through its own specification. Our strategy can be used in a complementary manner to first validate the respective transformation specification that captures the program transformations, and then provide a more reliable input for a possible implementation.

The work in (BECKER *et al.*, 2011) focuses on the consistency preservation of the refactorings. In their approach, the language's metamodel, the set of well-formedness constraints and maintenance rules are language-specific and they are formalized as graph transformations. A model is considered consistent if no maintenance rule is applicable to it and if it satisfies each well-formedness constraint. In our strategy, this well-formedness constraint can be represented by the *wellFormedProgram* predicate where the model being checked is the one that represents the resulting program. In (BECKER *et al.*, 2011), a refactoring specification S consists of a set of rules, formalized by graph rules, with priorities —when more than one rule is applicable to the same graph, the rule with the highest priority is applied first. There is an invariant checker which proceeds for statically verifying that the absence of forbidden patterns is preserved by a set of graph transformation rules with priorities; it analyses statically which kind of graph elements may be produced by a rule, and then it checks how these created graph elements may be overlapped with the forbidden pattern. To have a consistency-preserving refactoring they use the invariant checker to show that the refactoring rules do not produce any forbidden patterns. In addition they check that no maintenance rule is done during the refactoring phase. Otherwise, the addition of some predicate by a maintenance rule could lead to a violation of some forbidden pattern during the refactoring phase as well. If violations are found, counterexamples are produced and the developer is then able to inspect and change the refactoring specification accordingly. Similarly,

this can be done in our work through the counterexamples returned by our Static Semantics Validator. Besides, our Dynamic Semantics Validator presents the behaviour problem(s) along with the corresponding pair of programs (see our Validate step, Figure 23).

Table 3 Comparison of the main works presented

Works	specification*	implementation*	validation (S/I)*	Dependency of a particular source language	Internal model representation	Internal model generation	Static semantics check	Dynamic check
Ours	No	Yes (Alloy)	Yes (S)	No	Alloy instance	Alloy Analyzer	model finding	model finding + testing
Overbey and Johnson (OVERBEY; JOHNSON, 2011)	Yes	Yes (various)	Yes (I)	Yes (various)	program graph (ASTs)	their refactoring engine implementation translates from the source language into ASTs	ASTs comparison+testing	ASTs comparison (with reservations) + testing
Soares (SOARES, 2015)	No	No	Yes (I)	Yes (Java)	Java Program	some refactoring engine implementation	Java Compiler	SafeRefactor
Mongiovi (SABINO, 2016)	No	No	Yes (I)	Yes (Java, C, AspectJ)	Java, C or AspectJ Program	some refactoring engine implementation	Java Compiler	SafeRefactor-Impact
Schafer (SCHAFER, 2010)	Yes	Yes (Java)	Yes (I)	Yes (Java)	Java Program	their refactoring engine (with JastAdd)	correctness proofs + testing	correctness proofs + testing

6

CONCLUSION AND FUTURE WORK

In this work we propose a strategy intended to help with the validation of transformation specifications (following a perfective, corrective or adaptive model evolution) relying on a combination of formal verification techniques, Alloy and the Alloy Analyzer, and testing. For doing this, we firstly build Alloy metamodels. The first one is the OO metamodel which groups OO features. Although based on JLS elements, this model is used to validate specifications in languages other than Java (i.e., rCOS, ROOL, and an object–oriented language with reference semantics presented in (PALMA, 2015)).

In our strategy, it is possible to completely carry out the static semantics validation of a transformation specification using only our Alloy infrastructure. However, this is not the case for our dynamic validation. There are some reasons for this. Firstly, in most cases, the semantics of the programming language, in which the transformations are expressed, is not even fully defined in a precise way. Even if a formal semantics is available, the effort for encoding it and carry out a full semantic analysis in a framework like Alloy would probably not scale. This is the main motivation for our choice to combine formal analysis and testing. It is important to emphasize that the dynamic validation can be discarded in cases of transformation specifications following a adaptive evolution model. Hence, for this specific case, it is possible to completely carry out the specification validation using only our Alloy infrastructure.

Although our long term goal is to develop a language agnostic framework, we choose Java as the reference language for the semantics validation since this language owns a compiler whilst the other languages where transformations were specified, do not. Some of them such as rCOS are specification, as opposed to programming languages. Others like ROOL have not been implemented. Although these languages have a similar semantics, it is important to emphasize that an error pointed by our strategy (represented by a pair of programs denoting the transformation application) is actually a possibility of an error, which needs to be checked with regard to the semantics of the language (for which the transformation is specified) in question.

A Java tool was built, which is comprised by some other tools: (1) the Alloy–To–Java translator, which translates the Alloy instances (generated by our dynamic validator) into Java programs. For doing this, we benefit from JDolly (SOARES, 2015) that, among other things, also translates Alloy instances into Java programs. However, we had to change the way this

translation is done since the Alloy metamodel used in JDolly was completely different from ours; in our case, we have a completely different structure since, besides the OO metamodel, there are the transformation–specific and validators models, whose generated instances characterize programs before and after the transformation application; (2) a Java Test tool, where tests are generated to be executed by the generated Java programs. Actually, these two tools are integrated and tests are generated at the same time the programs are translated, benefiting from the ASTs of the programs before and after the transformation. These tests exercise methods of each side of the transformation. The corresponding *main* method of each Test class is applied in the context of the classes generated for the starting– and the resulting–hand sides of a transformation. This enables detecting behavioural errors.

The second Alloy model is actually a family of specific metamodels, one for each transformation specification being investigated. This model is comprised by predicates defining the transformation from an SS to an RS program. It uses the elements defined in the first model. Finally, the last Alloy metamodels are our Validator Models; in the case of the Static Semantics Validator, Alloy instances where the RS programs present static semantics problems are returned as counterexamples by the Alloy Analyzer. In the case of the Dynamic Validator, the Alloy Analyzer produces Alloy instances that represent a set of classes in Java involved in a transformation so that it is possible to identify the ones *before* and *after* the transformation. The programs resulted from the Alloy–To–Java translation always compile since our Dynamic Validator only runs *after* the Static Semantics one—thus they are always compliant to the subset of the JLS defined in our strategy and discussed in Section 3.1.1.

Although the counterexamples returned by the Static Semantics Validator can be analysed using the Alloy evaluator tool, the Alloy–To–Java translator can be used to ease and speed up the detection of which static semantics error is appearing in the RS program, specially in the cases when the user is not an Alloy developer with experience or desires to fastly know all of the static semantics errors of all of the Alloy instances (counterexamples returned). As a future work, discussed further, we can even categorize the errors and have an idea of which kind of errors appear (and the corresponding count of each one) in the transformation specification.

Results showed that transformation static semantics failures can be detected by the Alloy Static Semantics Validator without the need to implement them in a source language or submitting them to a more elaborate test campaign as done in (SOARES, 2015; SABINO, 2016). Actually the Alloy Analyzer acts, with an adequate model, as a powerful test generator since it generates, according to the transformation scenario, different possibilities of classes, their relationships and properties such as methods and attributes, elements inside method bodies as well as the other properties included in our OO metamodel (see Section 3.1).

However, we are not underestimating oracles or testings done by tools in the literature (SCHAFER, 2010; SOARES, 2015; SABINO, 2016). With regard to our Dynamic Validator, we know it can be enhanced, or maybe more elaborated testing tools can be used, instead of tests based on common methods invoked randomly and with structural comparison of the results in

SS and RS programs. This can be cited as a limitation of our work. If we had a complete formal semantics of our OO subset, coded in an Alloy model (or maybe a formal semantics for each language in which the transformations are specified), the dynamic validation would be done using the Alloy Analyzer as well, similarly to what we do with the static semantics validation (subject to scalability, as discussed before).

The Validate step is very valuable and also speeds up a more complete dynamic validation (if desired) since the pair of programs already characterizing the transformation are generated based on a formal infrastructure. Thus, this can be faced as a previous step before the dynamic validation itself, done with test tools in the literature. Commonly, the dynamic validation in previous works is done considering a program refactored but this refactoring is done by IDEs implementations. In our case, the representation of the program refactored is generated by our formal infrastructure and then translated to Java. Thus we have a more confidence in the transformation specification, considering the resulting program does not contain any static semantics problems and that a dynamic validation is also done, even if partially.

Another contribution of our work is that we specify the models in Alloy following a pattern in terms of elements and predicates—see Figure 18 in case of OO model and Figure 21 in case of transformation–specific model. In this regard, the structure of our Alloy models is predictable, intuitive, thus easing their future automatic generation (from a source transformation) and extension. For instance, if a new kind of *Statement* is included in the OO model such as the *SwitchStatement* (see Figure 13), a predicate similar to the *wellFormedMethodInvocation* predicate in Figure 18, but for a *SwitchStatement*, should be added. As long as it is necessary to include new elements, the pattern of the grammar in JLS is followed, with simplifications, according to the scope of the transformation, as discussed in Section 3.1.1.

However, when the OO model is enhanced to include new elements of the transformation specifications, the Alloy–To–Java translator has to be adapted likewise. Nevertheless, for some elements, some difficulties are introduced. For instance, the concurrency laws reported in (DUARTE, 2008) have elements such as *Threads*, *joins*, typical of a concurrency environment, that requires not only a careful mapping into our OO model, introducing the adequate types and predicates to guarantee the well–formedness of the programs, but also a correct translation from the Alloy instances into Java programs. This can be very challenging specially considering a concurrent context.

Besides, we observe that the predicates specifying the transformations from SS to RS programs (in the Alloy transformation–specific model) also follow a structure, where firstly some variables are declared (these ones are almost the same in all the transformation predicate specifications). Secondly, the code establishes the elements in both sides of the transformation template: left and right. Thirdly, predicates corresponding to each proviso, condition or substitution in the transformation appear. Finally, the equivalence between the SS and RS programs is established. We also defined a mapping between each predicate in the transformation–specific model and its corresponding proviso, condition or substitution (see Figures 19 and 20). As these

provisos, conditions or substitutions commonly appear repeatedly along different transformation specifications, these predicates can be reused. The reuse of the predicates minimizes the possibilities of errors and helps the users to specify the transformations. In addition, as a future work, in cases the Static Semantics Validator generates counterexamples, we could suggest the possible predicate that could be potentially related to it.

The effort to code the metamodels contemplating each transformation as well as doing its validation using predominantly a formal technique demands significant effort. However, it is compensated by the benefits obtained from the exhaustive generation of the Alloy Analyzer, according to a given scope and specific for each transformation, in a way that a significant part of the transformation validation can be done using a formal technique. Our experiments helped to validate our own OO metamodel and some transformation specifications as shown in Chapter 4.

6.1 Future Work

Despite the results already achieved, there are some interesting directions for extending our strategy. Firstly, we plan to consider validating more transformation specifications. In addition, the scalability of our strategy should be evaluated. So far we realized that, when there is a problem in the specification related to a static semantics error in the resulting program, then the output of our strategy is fast. On the other hand, if the specification yields only a dynamic error in RS, then the performance tends to be unsatisfactory. This occurs because, firstly, the Static Semantics validator does not find any counterexamples (and this might take a long time since all possibilities—according to a given scope—are explored by the Alloy Analyzer) and, secondly, the Dynamic validator run until all possible Alloy instances (characterizing SS and RS programs involved in the transformation) are generated, which also tend to involve an intense computation. Afterwards, these Alloy instances have to be translated into Java programs for the semantic preservation to be checked via (automatic) testing.

Concerning the effort related to testing, a direction we plan to explore a more elaborate strategy to guide generation of the instances to be tested based on the transformation side conditions. Currently, the Alloy Analyzer is used without any guidelines. One possibility is a kind of ”intelligent” method *main*, that would be generated by our Alloy infrastructure and would be able to catch dynamic problems in the transformation. This method *main* can be represented by the relation *main* of type *Program* in our OO model and can be used in the current strategy as the method to be applied in both sides of the transformation, instead of the current tests generated. More elaborated testing tools (existent in literature) can also be used, instead of the current approach. At the end, we can even compare the results from using each of these possibilities (or some combination of them), according to well–defined metrics.

Another topic for further research is to encode an Alloy semantics for the language in which the transformations are expressed, and carry out semantic check using the Alloy Analyzer, as we do now for the static semantics verification. The main challenge for carrying out a full

dynamic semantics analysis using the Alloy Anayzer is scalability, as already discussed.

Although we have an Alloy infrastructure (with an OO and transformation–specific models and Alloy validators as well), using our strategy as it currently stands requires specialized knowledge of the Alloy language since a transformation–specific Alloy model is required for doing the evaluation of a specific transformation. Eventually, some predicates can be reused depending on the similarity among the provisos, conditions or substitutions of the transformation and the ones already encoded. The requirement for the knowledge of a formal language such as Alloy can be a barrier to some developers. Actually, every solution that requires a specific language or technology is not the ideal solution. Thus, it is also in our agenda providing a standard way of specifying transformations in a more accessible notation, as well as a translator from such a notation into the corresponding Alloy models, in a systematic way.

Although we have not yet automated the translation of a transformation specification into Alloy, we follow a pattern that will hopefully help in this direction. This pattern can be clearly seen in the way transformation–specific predicates were specified as well as how the OO model is structured and can be enhanced, as described in Chapter 3, and also mentioned in the beginning of this chapter. In addition, we started to define a mapping (see Chapter 3) from provisos, conditions or substitutions to the corresponding predicates. It can be enhanced as long as additional elements appear in new transformations.

As mentioned in Chapter 5, another future work is providing a tool where our strategy would be embedded, with traceability to the original transformation specification, hiding the encoding in Alloy.

Yet related to the just described tool, another future work is establishing a more elaborate traceability information related to the counterexamples generated by the Static Semantics validator and the transformation specification in a way that from a counterexample the tool could suggest which conditions are missing in the specification. The user can then accept the suggestion from the tool in an interactive and iterative process.

We also plan to explore the integration of our strategy with refactoring engines such as Eclipse. By applying the engine refactoring implementation to an SS program generated from an Alloy instance, we can obtain the result of the transformation and translate it back to Alloy and compare it with SS within the Alloy Analyzer. This allows a way of checking the soundness of refactoring engines in a more rigorous way than testing, but this also requires the encoding in Alloy of the semantics of the language in which the transformations are expressed.

REFERENCES

AL., C. H. et. Laws of programming. In: . [S.l.]: ACM, 1987. p. 30(8):672–686.

ALANEN, M. *et al. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*. [S.l.], 2003. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc-/summary?doi=10.1.1.138.8536>.

ATKINSON, C.; KüHNE, T. Model–driven development: A metamodeling foundation. In: . [S.l.]: IEEE Xplore Digital Library, 2003. p. 36–41, Volumn 20, Issue 5. ISSN 0740-7459.

BECKER, B. *et al.* Iterative development of consistency–preserving rule-based refactorings. In: . Springer–Verlag, 2011. p. 123–137. Disponível em: <http://link.springer.com/chapter/10-.1007%2F978-3-642-21732-6_9>.

BORBA, P.; SAMPAIO., A. Basic laws of rool: an object–oriented language. In: . [S.l.]: Revista de INFORMÁTICA TEÓRICA E APLICADA, 2000. p. 49–68.

BORBA, P. *et al.* Algebraic reasoning for object-oriented programming. *Science of Computer Programming, Special Issue on Program Transformation*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 52, n. 1-3, p. 53–100, ago. 2004. ISSN 0167-6423. Disponível em: <http://dx.doi.org/10.1016/j.scico.2004.03.003>.

BORBA, P.; SAMPAIO, A.; CORNELIO, M. A refinement algebra for object-oriented programming. In: . [S.l.]: Springer, 2003. p. 457–482.

CORNELIO, M. *Refactorings as Formal Refinements.* Tese (Doutorado) — Federal University of Pernambuco, UFPE, 2004.

DIAZ V., L. J. G.-B. B. M. O. *Advances and Applications in Model–Driven Engineering.* Hershey, PA, USA: Information Science Reference., 2014.

DUARTE, R.

Parallelizing Java Programs Using Transformation Laws. — Federal University of Pernambuco, UFPE, 2008.

DUARTE, R.; MOTA, A.; SAMPAIO, A. Introducing concurrency in sequential java via laws. In: . elsevier, 2011. p. 129–134. Disponível em: <http://www.sciencedirect.com/science/article-/pii/S0020019010003327>.

EKMAN, T.; HEDIN, G. The jastadd extensible java compiler. In: . New York, NY, USA: ACM Press, 2007. p. 1–18. ISBN 978-1-59593-786-5. Disponível em: <http://dl.acm.org/citation-.cfm?id=1297029>.

FOWLER., M. *Refactoring, Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 2002. Disponível em: <https://www.csie.ntu-.edu.tw/~r95004/Refactoring_improving_the_design_of_existing_code.pdf>.

FRANCE, R. *et al.* A metamodeling approach to pattern–based model refactoring. In: . Colorado State Univ., Colorado Springs, CO, USA: IEEE Xplore Digital Library, 2003. p. 52–58. ISSN 0740-7459. Disponível em: <http://cs.ecs.baylor.edu/~song/publications/IEEE-Software.pdf>.

GHEYI, R.; MASSONI, T. Formal refactorings for object models. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2005. (OOPSLA '05), p. 208–209. ISBN 1-59593-193-7. Disponível em: <http://doi.acm.org/10.1145/1094855.1094938>.

GROUP, O. M. *MetaObject Facility*. 2016. Disponível em: <http://www.omg.org/mof>.

JACKSON, D. *Software Abstractions: Logic, Language, and Analysis.* New York, NY, USA: MIT Press., 2006.

JUDSON, S.; CARVER, D.; FRANCE, R. A metamodeling approach to model refactoring. *IEEE Software*, IEEE, v. 20, n. 5, p. 52–58, 2003. ISSN 0740–7459. Disponível em: <http://www.cs.colostate.edu/~france/publications/Judson_UML2003.pdf>.

KLEPPE, A. G.; WARMER, J.; BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 2003. Disponível em: <http://dl.acm.org/citation.cfm?id=829557>.

LONG Q., H. J. L.-Z. *Refactoring and pattern-directed refactoring: A formal perspective. Technical Report 318.* [S.l.], 2005.

MENS, T.; TOURWÉ, T. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, IEEE, Piscataway, NJ, USA, v. 30, n. 2, p. 126–139, 2004. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2004.1265817>.

MILLER, J.; MUKERJI, J. *MDA Guide Version 1.0.1.* OMG., 2003. Disponível em: <http://www.enterprise-architecture.info/Images/MDA/MDA%20Guide%20v1-0-1.pdf>.

MORGAN, C. *Programming from Specifications (2Nd Ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1994. ISBN 0-13-123274-6.

NAUMANN, D.; SAMPAIO, A.; SILVA, L. Refactoring and representation independence for class hierarchies. In: . elsevier, 2012. p. 60–97. Disponível em: <http://www.sciencedirect.com-/science/article/pii/S0304397512001284>.

OMG. *UML 2.0 Infrastructure Specification.* OMG., 2015. Disponível em: <http://www2-.informatik.hu-berlin.de/sam/lehre/MDA-UML/UML-Infra-03-09-15.pdf>.

OPDYKE, W. *Refactoring object-oriented frameworks.* Tese (Doutorado) — University of Illinois at Urbana- Champaign, 1992.

ORACLE. *Java Language Specification*. 2016. Disponível em: <http://docs.oracle.com/javase-/specs/jls/se8/jls8.pdf>.

OVERBEY, J.; JOHNSON, R. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In: . New York, NY, USA: IEEE, 2011. p. 303–312. ISBN 978-1-4577-1638-6. ISSN 1938-4300. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all-.jsp?arnumber=6100067&tag=1>.

OVERBEY M. J. FOTZLER, A. J. K. J. L.; JOHNSON, R. E. *A collection of refactoring specifications for Fortran 95, BC, and PHP 5*. [S.l.], 2011. Disponível em: <http://jeff.over.bz-/papers/2011/tr-refacs.pdf>.

PALMA, G. *Algebraic Laws for Object Oriented Programming With References.* Tese (Doutorado) — Federal University of Pernambuco, 2015.

QUAN, L.; ZONGYAN, Q.; LIU, Z. Formal use of design patterns and refactoring. In: . Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 323–338. ISBN 978-3-540-88479-8. Disponível em: <http://dx.doi.org/10.1007/978-3-540-88479-8_23>.

SABINO, M. *Scaling Tests of Refactoring Engines.* Tese (Doutorado) — Federal University of Campina Grande, 2016.

SAMPAIO, A. *Algebraic Approach to Compiler Design (Book 4)*. [S.l.]: World Scientific Publishing Company; 1st edition (January 15, 1997), 1997. ISBN 9810223919.

SCHAFER, M. *Specification, Implementation and Verification of Refactorings*. Tese (Doutorado) — Oxford University Computing Laboratory, 2010.

SCHAFER, M.; EKMAN, T.; MOOR, O. Formalising and verifying reference attribute grammars in coq. In: . Springer–Verlag, 2009. p. 143–159. Disponível em: <http://researcher-.watson.ibm.com/researcher/files/us-mschaefer/agcoq.pdf>.

SELIM, G.; CORDY, J.; DINGEL, J. Model transformation testing: The state of the art. In: . New York, NY, USA: ACM Digital Library, 2012. p. 21–26. ISBN 978-1-4503-1803-7. Disponível em: <http://dl.acm.org/citation.cfm?id=2432502>.

SILVA, L.; SAMPAIO, A.; LIU, Z. Laws of object-orientation with reference semantics. In: . Washington, USA: IEEE, 2008. p. 217–226. ISBN 978-0-7695-3437-4. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167642304000474>.

SOARES, G. *Automated behavioral testing of refactoring engines.* Tese (Doutorado) — Federal University of Campina Grande, 2015.

SOARES, G. *et al.* Making program refactoring safer. In: . New York, NY, USA: IEEE, 2010. p. 52–57. ISSN 0740-7459. Disponível em: <http://ieeexplore.ieee.org/xpl/login-.jsp?tp=&arnumber=5440166&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all-.jsp%3Farnumber%3D5440166>.

APPENDIX A – OO METAMODEL

Code A.1 OO Metamodel

```
1  module javametamodel
2
3  open util/relation
4
5  abstract sig Id {}
6
7  sig ClassId, MethodId, FieldId, VarId extends Id {}
8
9  abstract sig Accessibility {}
10
11 one sig public, private_, protected extends Accessibility {}
12
13 abstract sig Type {}
14
15 abstract sig PrimitiveType extends Type {}
16
17 one sig Long_ extends PrimitiveType {}
18
19 sig ClassType extends Type{
20    classIdentifier: one ClassId
21 }
22
23 abstract sig Class {
24    extend: lone ClassId,
25    methods: set Method,
26    fields: set Field
27 }
28
29 sig Field {
30    id: one FieldId,
31    type: one Type,
32    acc :one Accessibility
33 }
34
35 sig VarDec {
36    varName: one VarId,
37    type: one Type
38 }
39
40 sig Method {
41    id: one MethodId,
```

```
42    param: lone VarDec,
43    return: lone Type,
44    acc: one Accessibility,
45    body: seq Statement
46 }
47
48 abstract sig Expression {}
49 abstract sig Statement extends Expression {}
50
51 sig AssignmentExpression extends Statement {
52    pExpressionLeft: one LeftHandSide,
53    pExpressionRight: one {Expression - PrimaryExpression -
         AssignmentExpression}
54 }
55
56 abstract sig PrimaryExpression extends Expression {}
57
58 sig this_, super extends PrimaryExpression {}
59
60 sig newCreator extends PrimaryExpression {
61    id_cf :one ClassId
62 }
63
64 abstract sig LeftHandSide{}
65
66 sig FieldAccess extends LeftHandSide {
67    pExp: one PrimaryExpression,
68    id_fieldInvoked: one FieldId
69 }
70
71 sig ExpressionName extends LeftHandSide{
72    name: one VarId
73 }
74
75 sig MethodInvocation extends Statement{
76    pExp: one PrimaryExpression,
77    id_methodInvoked: one MethodId,
78    realParam: lone LeftHandSide
79 }
80
81 sig LiteralValue extends Expression {}
82
83 sig Program {
84    classDeclarations: ClassId -> one Class,
85    main: Method
86 }
```

```
87
88 pred wellFormedProgram [p:Program] {
89     all c:ClassId | c in (p.classDeclarations).univ =>
          wellFormedClass[p,c]
90 }
91
92 pred wellFormedClass[p:Program, c:ClassId] {
93     noCycleInExtends[p,c]
94
95     no disj f1,f2: p.classDeclarations[c].fields |
96     f1.id = f2.id
97
98     no disj m1,m2: p.classDeclarations[c].methods |
99     m1.id = m2.id &&
100    #(m1.param) = #(m2.param) &&
101    (m1.param.type = m2.param.type)
102
103    let class = c.(p.classDeclarations) {
104        superClassIsDeclared[p,class]
105        all m:Method | m in class.methods =>
106        wellFormedMethod[p,class,m]
107    }
108 }
109
110 pred noCycleInExtends[p:Program, c:ClassId] {
111    c !in c.^((p.classDeclarations).extend)
112 }
113
114 pred superClassIsDeclared[p:Program,c:Class] {
115    c.extend in (p.classDeclarations).univ
116 }
117
118 pred wellFormedMethod[p:Program, class:Class, m:Method]{
119
120    let body = (m.body).elems
121    {
122    all stm: Statement | stm in body =>
123    wellFormedStatement[p, class, stm, m]
124    }
125 }
126
127 pred sameSignature[m1,m2:Method]{
128    #(m1.return)= #(m2.return)
129    m1.id = m2.id && #(m1.param) = #(m2.param) &&
130    (#(m1.param) = 1 => (m1.param.type = m2.param.type))
131 }
```

```
132
133 pred lessOrEqualAccessibility[m,mOverriden: Method]{
134 //check accessibility of methods involved in overriding
135     (m.acc in protected && mOverriden.acc !in private_ &&
136     #(mOverriden.acc) != 0) ||
137     (m.acc in public && mOverriden.acc in public ) ||
138     (#(m.acc) = 0 && #(mOverriden.acc) = 0 )
139 }
140
141 pred wellFormedStatement[p:Program, class:Class, st:Statement,
       m:Method]{
142     st in AssignmentExpression => wellFormedAssignment[p,class,
          st,m]
143     st in MethodInvocation => wellFormedMethodInvocation[p,
          class,st,m]
144 }
145
146 pred wellFormedAssignment[p:Program, class:Class, stm:
       AssignmentExpression, m:Method] {
147     wellFormedLeftHandSide[p,class,stm.pExpressionLeft]
148
149     let rightExp = stm.pExpressionRight {
150         rightExp in MethodInvocation =>
              wellFormedMethodInvocation[p,class,rightExp,m]
151     }
152 }
153
154 pred wellFormedLeftHandSide[p:Program, class:Class, stm:
       LeftHandSide] {
155     stm in FieldAccess => wellFormedFieldAccess[p,class,stm]
156     stm in ExpressionName => wellFormedExpressionName[p,stm,m]
157 }
158
159 pred wellFormedFieldAccess[p:Program, class:Class, stm:
       FieldAccess] {
160
161     let target = stm.pExp {
162         wellFormedPrimaryExpression[p,class,target]
163         target in newCreator => stm.id_fieldInvoked in
164         ((target.id_cf).(p.classDeclarations).*(extend.(p.
              classDeclarations)).fields).id
165
166         target in this_ => stm.id_fieldInvoked in
167         (class.*(extend.(p.classDeclarations)).fields).id
168
169         target in super => stm.id_fieldInvoked in
```

```
170         (class.^(extend.(p.classDeclarations)).fields).id
171
172         target in ExpressionName => some f,f2:Field |
173         ((f.id = target.name && f in class.fields && f.type !in
               Long_) ||
174          (f.id = target.name && f in
175          class.^(extend.(p.classDeclarations)).fields &&
176          f.type !in Long_ && f.acc !in private_)) &&
177
178         (f2.id = stm.id_fieldInvoked &&
179          f2 in (f.type.classIdentifier).(p.classDeclarations).*(
                extend.(p.classDeclarations)).fields
180          && f2.acc !in private_)
181
182         ((f.type.classIdentifier).(p.classDeclarations).*(extend
               .(p.classDeclarations)).fields).id)
183     }
184 }
185
186 pred wellFormedExpressionName[p:Program, variable:
       ExpressionName, m:Method]{
187     variable.name in (m.param).varName
188 }
189
190 pred wellFormedMethodInvocation[p:Program, class:Class, stm:
       MethodInvocation, m:Method]{
191     stm.pExp in PrimaryExpression =>
           wellFormedPrimaryExpression[p,class,stm.pExp]
192
193     let target = stm.pExp {
194         target in newCreator => (some m': Method |
195         m' in (target.id_cf).(p.classDeclarations).*(extend.(p.
               classDeclarations)).methods &&
196         m'.id = stm.id_methodInvoked &&
197         #(stm.realParam) = #(m'.param) &&
198         ( ((target.id_cf).(p.classDeclarations) != class ||
199           m' !in class.methods) => m'.acc !in private_)
200           &&
201           wellFormedRealParam[p,stm,class,m'])
202
203         target in this_ => (
204         (some m': Method |
205         m' in class.^(extend.(p.classDeclarations)).methods &&
206         m'.id = stm.id_methodInvoked &&
207         #(stm.realParam) = #(m'.param) &&
```

```
208         m'.acc !in private_ && wellFormedRealParam[p,stm,class,m
               ']) 
209         ||
210         (some m': Method | m' in class.methods &&
211          m'.id = stm.id_methodInvoked &&
212          #(stm.realParam) = #(m'.param) &&
213          wellFormedRealParam[p, stm, class, m']) )
214 
215 
216         target in super => some m': Method |
217         m' in class.^(extend.(p.classDeclarations)).methods &&
218         m'.id = stm.id_methodInvoked &&
219         #(stm.realParam) = #(m'.param) &&
220         m'.acc !in private_ &&
221     wellFormedRealParam[p,stm,class,m']
222 
223         target in ExpressionName =>
224         wellFormedExpressionNameInMethodInvocation[target,class,
               p,stm]
225     }
226 }
227 
228 pred wellFormedExpressionNameInMethodInvocation[target:
        ExpressionName, class:Class, p:Program, stm:MethodInvocation
        ]{
229 
230     some f:Field | some m2:Method |
231     ((f.id = target.name && f in class.fields && f.type !in
           Long_)
232      ||
233      (f.id = target.name &&
234      f in class.^(extend.(p.classDeclarations)).fields &&
235      f.type !in Long_ && f.acc !in private_)) &&
236 
237      ((m2.id = stm.id_methodInvoked &&
238       m2 in (f.type.classIdentifier).(p.classDeclarations).
              methods)
239       ||
240       (m2.id = stm.id_methodInvoked &&
241        m2 in (f.type.classIdentifier).(p.classDeclarations).^(
              extend.(p.classDeclarations)).methods &&
242        m2.acc !in private_ )) &&
243       wellFormedRealParam[p,stm,class,m2]
244 }
245 
```

```
246 pred wellFormedRealParam[p:Program, stm:MethodInvocation, class
       :Class, m:Method]{
247     #(stm.realParam) = #(m.param)
248     #(stm.realParam) = 1 => some f:Field |
249     wellFormedFieldAccess[p,class,stm.realParam] &&
250     f.id = stm.realParam.id_fieldInvoked &&
251     (m.param.type in Long_ => f.type in Long_) &&
252     (m.param.type in ClassType => (f.type in ClassType &&
253      firstIsSubtypeOfTheSecondOneClass[p,f.type.classIdentifier,
           m.param.type.classIdentifier]))
254
255 }
256
257 pred wellFormedPrimaryExpression[p:Program, c:Class, stm:
       PrimaryExpression] {
258      stm in newCreator => classIsDeclared[p,stm.id_cf]
259 }
260
261 pred classIsDeclared[p:Program, c:ClassId] {
262     let cds = p.classDeclarations {
263         c in cds.univ
264   }
265 }
266
267 pred firstIsSubtypeOfTheSecondOneClass[p:Program, first,second:
       ClassId] {
268     let secondSubClasses = second.*~((p.classDeclarations).
          extend) {
269         first in secondSubClasses
270     }
271 }
```