UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Pós Graduação em Ciência da Computação

Tese de Doutorado

A Model-driven Approach to Formal Refactoring

por

Tiago Lima Massoni

Recife, março 2008

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Tese de Doutorado

A Model-driven Approach to Formal Refactoring

Tiago Lima Massoni

Esta tese será apresentada à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

This thesis will be presented at the Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Doctor (Dr.) in Computer Science.

Orientador (Supervisor):
Prof. Dr. Paulo Henrique Monteiro Borba

Recife, março 2008

Massoni, Tiago Lima
A model-driven approach to formal refactoring / Tiago Lima Massoni. – Recife: O Autor, 2008.
xi, 213 p.: il., fig., tab.

Tese (doutorado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2008.

Inclui bibliografia e apêndice.

1. Engenharia de software. Título.

005.1 CDD (22.ed.) MEI2008-037

To the wonder of life.

Acknowledgements

I'd like to thank all you guys
For helping me out
Andreza who brings about
an everyday surprise

Cheer mom and dad
my brother and sister
who, better than any other
always made me glad

Rohit and Sérgio my dear friends
Standing by me on the trouble
Giving me the double
of what I'll ever do when it ends

My advisor, Paulo and household
who for seven years now
gave much to learn how
and shared friendship in bold

Augusto and other great professors
Ana, Marcel, Juliano and Alexandre mainly
For discussing plainly
big research endeavors

Praises to my DSC, The Department
Which gave me a place
To dedicate and engage
For pursuing great accomplishment

I cannot forget
the friends I made there
who made me aware
we'll be big, I bet

as well the greatest research group
and friends in this university
whose brightness and grandiosity
make it an elite troop

Also I have to thank
the doctors I guess
who gave me the bless
and energy of a think tank

I cannot forget the blow
the rock n' roll ground
who made up the background
where these ideas could flow

Resumo

Como qualquer outra tarefa evolucionária, a aplicação de refatoramentos em *software* orientado a objetos normalmente afeta código-fonte e seus modelos relacionados, aumentando a dificuldade de manutenção de artefatos corretos e consistentes. Devido à distância de representação entre artefatos de modelagem e programação, o esforço ligado a refatoramentos logo torna-se duplicado e custoso. Neste contexto, suporte de ferramentas utilizado atualmente, em especial ferramentas de *Round-Trip Engineering (RTE)*, falha em automatizar tarefas de evolução. Conseqüentemente, a maioria dos projetos de *software* descarta artefatos de modelagem precocemente, adotando abordagens centradas unicamente em código-fonte. Esta tese propõe uma abordagem formal para consistentemente refatorar modelos de objeto e programas orientados a objetos, baseando o refatoramento apenas em modelos de objetos. Refatoramento de modelos é fundamentado com transformaçoẽs formais primitivas – leis de modelagem – que são garantidamente preservadoras de semântica. Cada refatoramento aplicado a um modelo de objetos é associado a uma seqüência semi-automática de aplicações de leis de programação preservadoras de comportamento, chamadas estratégias. Estratégias são aplicadas na dependência de um relacionamento específico de conformidade entre modelos de objetos e programas, que devem satisfazer também um dado grau de confinamento. Este trabalho formaliza 14 estratégias, duas para cada lei de modelagem que afeta estruturas do programa. Estas estratégias são formalizadas como táticas de refinamento. Desta forma, refatoramento correto de programas pode ser realizado com reduzida intervenção manual do desenvolvedor, baseado apenas nas transformações que o mesmo aplicou ao modelo. Neste cenário, refatoramentos complexos que afetam as principais estruturas do programa podem ser aplicados a um artefato de mais alto nível de abstração, deixando a atualização semi-automática dos detalhes de implementação para as estratégias. Além disso, invariantes do modelo podem ser usados para aprimorar ferramentas especializadas em refatoramento, já que modelos de objetos oferecem informação semântica que permite refatoramentos automáticos mais poderosos. Esta tese considera Alloy como linguagem de modelagem formal, além de uma linguagem de programação similar a Java que chamamos BN. Para esta linguagem, introduzimos quatro novos refatoramentos e leis de programação orientada a objetos, com suas provas e derivações correspondentes. Adicionalmente, as leis de programação foram aplicadas em uma semântica de referências, mais próxima de linguages de programação utilizadas na prática. Com o intuito de delimitar a aplicabilidade desta abordagem, formalizamos uma noção de conformidade entre modelos de objetos e programas, a partir de um *framework* formal para definição de relacionamentos de conformidade; as definições formais relacionadas foram especificadas e checadas quanto ao tipo na ferramenta PVS. Além disso, estabelecemos e provamos manualmente um teorema para a corretude das estratégias, definindo que elas preservam comportamento e conformidade dos programas refatorados. Mesmo sendo uma abordagem formal, temos a preocupação de discutir sua utilização prática, além de aplicá-la em três estudos de caso. Os problemas apresentados nesta tese certamente serão enfrentados em qualquer abordagem de desenvolvimento dirigida por modelos, no momento em que se lida com evolução.

Palavras-chave. Alloy, refatoramento, confinamento, desenvolvimento dirigido por modelos.

Abstract

Refactoring object-oriented software, as any other evolutionary task, usually affects source code and object models, burdening developers to keep those artifacts correct and up to date. Due to the gap between modeling and programming artifacts, refactoring efforts soon become duplicate and considerably expensive. In this context, currently used tool support, in special Round-Trip Engineering (RTE) tools, fails to fully automate evolution tasks. Consequently, most projects discard object models early in the life cycle, adhering to code-driven approaches. This thesis proposes a formal approach to consistently refactor object models and object-oriented programs of a system in a model-driven manner. Model refactoring is backed by formal laws of modeling, which are guaranteed to be semantics preserving. Each refactoring, a composition of laws, applicable to an object model, is associated with a semi-automatic sequence of applications of laws of programming, called strategy. Strategies are applied by relying on a specific conformance relationship between object models and programs, which must fulfill a specific degree of confinement. We formalized 14 strategies, two for each law of modeling that affects program structures. These strategies have been formalized as refinement tactics. In such scenario, complex refactorings that affect main program structures can be applied abstractly, leaving the update of implementation details to strategies. Also, model invariants can be used to improve refactoring automation, as they provide runtime information that allows automation of more powerful program refactorings. This thesis considers Alloy as the formal modeling language, along with a simplified Java-like programming language that we call BN. For this programming language, we introduced four new refactorings and laws of object-oriented programming, with their correspondent derivations and proofs. Also, the laws of programming have been used in a reference semantics context, which is closer to current mainstream programming languages. In order to delimitate the applicability of model-driven refactorings, we formalize a specific conformance relationship, using a underlying general framework for formalizing conformance relationships; the related formal definitions have been specified and type-checked with PVS. In addition, we establish and manually prove a soundness theorem for strategies, guaranteeing that they preserve the target program's behavior and conformance. Despite of its formality, we also regard the utilization of this theory in practical object-oriented development, by discussion and three case studies that simulate refactoring situations for object models and programs. The results presented here shows evidence on issues that will surely recur in other Model-Driven Development contexts.
Keywords. Alloy, refactoring, confinement, model-driven development.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During software development, *evolution* is acknowledged as a demanding activity. Software is changed more often than it is written, and these changes can be notably costly [69]. Evolution is further complicated by the adoption of models, which constrain the behavior of software with decisions made by designers in a more abstract context. For instance, *object models* [58, 70] specify structures, relationships and invariants over the state of object-oriented programs. In this scenario, it is useful that abstractions in models and source code evolve consistently; however, this task is time-consuming and costly in practice.

In particular, the originally defined structure of software usually does not smoothly accommodate adaptations or additions, demanding new ways of reorganization, in order to allow smoother and cheaper evolution. Modern development practices, such as *refactoring* [37, 82], improve the design of programs while maintaining its observing behavior, mainly for preparing software for changing requirements. Refactoring must be executed with care, as the task must maintain the observable behavior of the target program. In the context of model-driven development, these tasks become even more complex, with additional forces to consider, such as code generation, evolution and conformance.

1.1 Problem

When evolving programs or object models, maintaining those artifacts consistent is usually hard, requiring manual updates, even with state-of-the-art tool support. As a consequence, projects commonly discard models during development, adhering to code-driven approaches. The same issues are observed when refactoring multiple artifacts, limiting the benefits of working with models in restructuring tasks.

Currently used tool support, in special Round-Trip Engineering (RTE) tools [90], fails to fully automate evolution tasks due to the abstraction gap between models and programs. For instance, if programs are refactored, models can be generated by reverse engineering. Producing correct and abstract models from modified programs – as depicted in Figure 1.1(a) – usually yields undesired results, in which the generated models usually correspond to source code visualizations, hindering comprehension of key design decisions. Consistency conditions between object models and programs – hereafter called conformance relationship – are established but not detailed here, for simplicity.

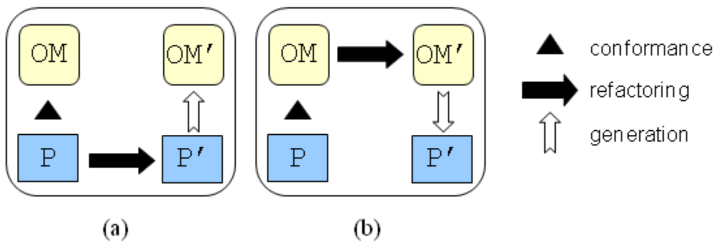


Figure 1.1: Software evolution approaches with RTE tools

In an alternative situation, source code generation is often applied to accelerate implementation tasks with code skeletons. Nevertheless, in the context of refactoring, code generation-based development is ineffective. Assuming that the model is refactored, structures may be modified; for instance, one of these modified structures may be implemented as a class in the program. A usual technique in RTE tools marks fragments edited by programmers as immutable, in the original program, in order to avoid overwriting during code regeneration. However, this immutable code may rely on design structures that were modified in the model, resulting in an incorrect program. From the refactored model, code is generated, and the class relating to the changes structure is also changed. In this scenario, the program changes make the previously user-edited source code incorrect – as depicted in Figure 1.1(b) –, which requires manual updates for conformance.

As such, the artifacts quickly end up outdated, as manual updates gradually become more expensive. Therefore the problem statement is established: refactoring when models and programs are involved is mostly a manual and nonproductive task; little is known about the relationship between object model and object-oriented program constructs, and current tool support often fails in achieving conformance maintenance.

Alternative approaches for using models as primary development artifacts are linked to Model-Driven Development (MDD) [49]. In special, the Model-Driven Architecture (MDA) [66] encourages the use of UML as a development language with executable semantics [7], using action semantics [1] or statecharts, which generates code for several implementation technologies. This executable models is realized either in a high-level programming language or by direct compilation to executable representations like assembly languages. Therefore, the accepted idea of abstraction in MDA is linked to implementation platforms, not programming logic – the latter is included as part of the models itself. If source code is manipulated somehow, the aforementioned problems will recur. Differently, Domain-Specific Languages (DSL) [35] are used as modeling languages for constrained contexts, such as embedded systems for a given domain, such as avionics or computer games. In this case, developers are not supposed to change source code, as seen in tools exemplified by SCADE [33] and Perfect Developer [32].

This thesis focus on two aspects of this problem: conformance between models and source code, and how to feasibly co-evolve these artifacts. In terms of conformance, research approaches have been investigating issues in terms of automating conformance checking and generation [53, 84, 27], with little on formalizing these conformance relationships.

Into evolution among several conforming artifacts, coupled [67] and bi-directional transformations [14] establish theories for elements that must evolve in conjunction, lacking a more in-depth solution for specific modeling artifacts, as object models (subject to study in this thesis). Other approaches [76, 55, 85] provide more concrete guidelines for maintaining conformance during evolution, although considering rather concrete models (design models).

1.2 Relevance

The use of abstract information is acknowledged as a powerful instrument for coping with complexity during software development. Software modeling offers means to exploit the benefits of abstract information in tackling complexity and size. However, as models are not source artifacts, evolution makes them obsolete very quickly. This is the manifestation of the tension between dealing with complexity and dealing with change [49]. Methods and tools for partially or even totally removing human interaction in the process are invaluable to the refactoring practice, due to productivity needs in software development.

Furthermore, discussions over model and program conformance often reduce to the problem of defining a specific conformance relationship between models and source code elements. The correspondence between what changes in the source code from what changes in the model is in the core of automatic evolution tasks. For that, several aspects must be considered; primarily, the level of abstraction must be established, in order to define a policy for constraining programs that are in conformance with a model. For instance, changing a set of objects in an object model may affect only a single class, if there is a direct implementation of the set as a class in the program. For model refactoring, this seems appropriate, as refactoring aims at restructuring design decisions. Although automation in this case is relatively simple, similar one-to-one correspondence may restrain abstraction. The problem of establishing the right level of abstraction in a model-driven context permeates software evolution in general, defining how two artifacts can evolve in synchronization.

1.3 Research Questions

This thesis aims to provide answers to the following research questions; these questions are enumerated for later reference:

1. Is there a satisfactory conformance relationship between object models and object-oriented programs that allows both refactorings with some degree of automation and a high level of abstraction? What is its formal definition?

2. For this conformance relationship, how can we deal with source code synchronization in an incremental way, when part of the model has been refactored? What is the degree of automation that is possible in this approach?

3. Is this approach sound in terms of, among other properties, behavior and conformance preservation?

4. What are the consequences of applying this approach in real-world contexts?

1.4 Solution

This thesis proposes a formal approach to consistently refactor object models and corresponding object-oriented programs in a *model-driven manner*. A sequence of formal behavior-preserving program transformations, which we call *strategy*, is associated to each predefined model refactoring. Applying a model refactoring triggers the corresponding strategy, which, in a semi-automated way, (1) updates code abstractions as refactored in the model and (2) adapts implementation details according to the modified abstractions; this is accomplished with *model invariants*, which are assumed throughout all program's executions, through a conformance relationship. Although developers only apply refactorings to object models, in this approach both artifacts get refactored, reducing manual updates on source code.

Our approach is a formal investigation of evolution tasks, showing evidence about issues with keeping object models and their implementations in conformance during refactoring. We establish our approach on formal languages relying on previous work in object modeling and program refinement. For instance, we consider object models in Alloy [58], which includes structures for expressing objects, relations and invariants equivalent to the core concepts of UML class diagrams and OCL [98]. For programs, we consider a small object-oriented language inspired by the language presented in Banerjee and Naumann's work [8], which we call *BN* (from the authors' last name initials), developed for reasoning about object-oriented programming.

In order to precisely determine what changes in the program for each model refactoring, this thesis offers a formal conformance relationship, based on syntactic and semantic constraints. From this result, a formal framework for defining conformance relationships between object models and programs was devised; its formal definitions were specified and type-checked in the (Prototype Verification System (PVS) [83]. Also, the soundness of each strategy is enunciated as a theorem, manually proved for a representative subset of the strategies proposed in this thesis. In this soundness proof, we ensure that strategies are program refinements (as expected for refactorings), and the refactored program is in conformance with the refactored model. In addition, we further evaluate our approach in three case studies, in order to investigate its effect in software artifacts.

1.5 Summary of Contributions

Accordingly, we summarize the contributions of this thesis as follows:

- A formal conformance relationship between Alloy models and object-oriented programs for refactoring purposes;
- An approach that relates object model refactorings with strategies of program transformations, offering semi-automatic program refactoring based only on the available model information (they cannot be fully automatic, mainly due to class refinement and refactoring quality issues);

- Formalization of strategies as refinement tactics;
- Establishment and proof of soundness theorems for strategies, which ensure refinement and conformance;
- Three case studies using our approach;
- A formal framework for defining conformance relationships, specified and typechecked in PVS;
- Four new refactorings and laws of object-oriented programming, with the corresponding derivations and proofs;
- Proofs, in a reference semantics, for laws originally developed for copy semantics. The proofs are backed by a theory of confinement;
- Formalization, as refinement tactics, of critical steps of normal form reduction for an object-oriented program.

1.6 Organization

In the next chapter, we survey the state of the art on refactoring and model-driven evolution. We provide some background on primitive transformations and the involved object modeling language in Chapter 3, while Chapter 4 establishes our contributions for defining a programming language and a catalog of laws over which our approach is based. Chapter 5 builds the foundation of our approach, establishing a notion of conformance between Alloy models and object-oriented programs, followed by the general formal framework that allows definition of conformance relationships between object models and programs. In Chapter 6, we present our approach in detail, including strategy definitions as refinement tactics. The general theorem and soundness proofs are presented in Chapter 7, while Chapter 8 evaluates our approach in small case studies. Finally, Chapter 9 summarizes the contributions of the thesis, related and future work (Appendices A, B, C, D, E provide additional content for the thesis, including catalogs of modeling and programming laws, strategy descriptions and complete proofs).

Chapter 2

Background

This chapter presents some efforts related to the state of the art in evolution and refactoring considering programs and models, laying the ground for this thesis.

2.1 Software Evolution

Software development aims at the delivery of a software product which satisfies user requirements. Accordingly, the software product must change or evolve. Once in operation or still during development, defects are uncovered, operating environments change, and new user requirements surface. As a consequence, software maintenance can be considered one of the most expensive development task. Since most domains depend on software today, it must operate in complete synchronism with the ever-changing business to be automated [69]. In the context of evolution supported by tools, transformations – the act of changing one program to another, not necessarily written in distinct languages – are usually applied to operational software, in order to perform the desired changes and generate a new version of the software system.

A software process usually contains a comprehensive set of activities for software evolution. The procedure for dealing with evolution determines a classification criterion for software processes, as follows.

- Up-front design: in a process driven by this approach, most of the software architecture is defined previously during the software life cycle, exploring opportunities for reuse and extensibility. Risks are mitigated as early as possible, defining most requirements in early phases of the project. In this context, most evolutionary changes are expected to occur in this architectural phase. Industrial Processes based on the Unified Process [61] strongly rely on these principles.

- Evolutionary design: this approach guides software developers to design software for the requirements at hand at a given moment. Design and implementation activities take into consideration the concerns involving the current requirements, simplifying design decisions. However, as new requirements arise and changes are needed, developers apply techniques for improving the existing design – refactoring [37, 82] –, in order to smoothly apply changes and maintain the quality properties. Agile methods such as Extreme Programming [12] employ this approach as one of their main guidelines.

Developers involved in evolutionary practices usually perform activities comprising four key characteristics, according to Pfleeger [86]:

- Maintaining control over the software system's day-to-day functions;
- Maintaining control over software modification;
- Perfecting existing functions;
- Preventing software performance from degrading to unacceptable levels.

2.1.1 Issues Related to Evolution

A few technical issues related to evolution recur as developers face the task of evolving systems, either during development or maintenance activities. For instance, *limited understanding* is often present, referring to how quickly a software engineer can understand where to modify or correct software which this individual did not develop. Comprehension is more difficult in text-oriented representation (in source code, for example), where it is often difficult to trace the evolution of software through its releases if changes are not documented, especially when the developers are not available to explain it, which is often the case.

Testing is also a critical concern in evolution. The cost of repeating full testing on a major piece of software can be significant in terms of time and money. Regression tests – the selective re-testing of a software or component to check whether modifications have not caused unintended effects – are crucial to evolution, as modifications might affect more than one software component in unexpected ways, given strong dependencies between modules.

In evolution contexts, *impact analysis* describes how to conduct, cost-effectively, a complete analysis of the impact of a change in existing software. Developers must possess an intimate knowledge of the software's structure and content [86]. They use that knowledge to perform impact analysis, which identifies all software artifacts affected by a change request, and develops an estimate of the resources needed to accomplish the change [5]. Several potential solutions are provided and then a recommendation is made, as the likely best course of action.

Also an important evolution-related aspect, *maintainability* can be defined as the ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements. Maintainability factors must be specified, reviewed, and controlled during software development activities in order to reduce evolution costs. Nevertheless, developers are normally concerned with other aspects and often disregard the evolutionary requirements.

2.2 Program Restructuring

As software evolves, it usually becomes more complex, especially due to unanticipated changes. In general, addition of new requirements that were not foreseen by the original design tend to degenerate structure [13]. When the need for change arises, some decisions motivating the original structure may no longer be valid. Some changes may cross modules' boundaries, demanding non-local modifications to module interfaces and implementations, which may be troublesome modifications that compromise software reliability, extensibility and reusability, among others [51]. In some cases, although seeking productivity, developers try to evolve or maintain software in the fastest possible way, ignoring early design decisions and adding brittleness.

A former technique for addressing evolution-related problems was named *program restructuring*, which intends to reduce software complexity by incrementally improving its structure. Restructuring is applied for making programs easier to understand and change, as they become less susceptible to errors when future changes are made. This technique is not supposed to modify functionality from the previously working system, as they only enhance program structure to better accommodate later increments. Arnold [4] refers to the process of reorganizing the logical structure of existing software systems in order to improve particular quality attributes of programs. Some examples of software restructuring are: improvement of coding style, documentation edition, transformation of program components (by renaming variables or moving expressions) and enhancement of functional structures (like relocating functional components into other modules).

Concerned with how programmers perform restructuring tasks, Griswold [51] explores behavior-preserving transformations in order to automatically restructure programs, arguing that global restructuring can be cost-effective, but only if automated and separated from other qualitatively different evolution tasks. His transformations deal with program restructuring for aiding maintenance, significantly influencing its cost. As such, restructuring can isolate a design decision in a module so that changing it will avoid non-local complex changes.

2.3 Program Refactoring

In the context of object-oriented programming, restructuring is widely known as *refactoring*. The term was coined in Opdyke's Ph.D. thesis [82]. With refactorings, programmers rewrite part of an object-oriented program in order to improve certain qualities, such as reusability and extensibility, making the system structure easier to understand and change [37]. Refactoring initiatives tend to enforce coding standards, reduce duplication and the size of the program, improve modularity, and cause new abstractions to emerge from the code. Also, refactoring may minimize impact of changes, easing the task of impact analysis on programs, in addition to overcoming some technical issues related to evolution, such as limited understanding and maintainability.

Fowler, in his popular book [37], presents an extensive catalog of refactorings, primarily based on industrial experience. These refactorings include their motivations and step-by-step guidelines for their application, including the small changes needed to accomplish a refactoring effort in a practical way. Representative refactorings from Fowler's book include extracting a new class from a low-cohesive class, moving attributes between class hierarchies and replacing inheritance or delegation by one another, depending on the context.

Refactoring is often applied during maintenance activities, in which the lack of structure may be more evident. Nevertheless, agile methodologies, such as Extreme Programming [12], bring about refactoring as a constant practice in early design and implementation activities. Mostly it depends on programmers being taught to tell good code from bad code and being told to correct bad code whenever they see it. XP practitioners also have to be taught to balance time refactoring with time spent adding new features.

2.3.1 Refactoring Tools

Refactoring tasks can be performed by hand, by following a checklist of tasks to be accomplished in order to apply the refactoring as safely as possible, in the style of Fowler's *mechanics* [37]. In fact, these tasks can either be mixed with usual programming, performed in an *ad hoc* way, or separate from activities that add new features, as indicated by XP [12]. However, manual refactoring is tedious and error-prone, sometimes aided only by some primitive tools, featuring search and replace functions. This scenario may intimidate refactoring, restraining its benefits, such as high-quality and correct source code.

Program refactoring tools make it easier to refactor programs, being highly desirable to minimize the tiresome debugging and testing that must be performed, as the confidence in the final result of the refactoring is increased. These tools leave the developer in control of the subjective activities of choosing the appropriate structure for the redesign, which is the intelligent part of the refactoring process. While a catalog of refactorings can help developers choose the correct new design to apply, a refactoring tool helps the developer to apply the chosen design, especially performing *global changes*, affecting several parts of the program that depend on the changed structures (classes, interfaces, attributes). This feature can help developers overcome the technical issue of testing.

Significant research work has been carried out on the subject of refactoring tools. Roberts' Ph.D. thesis [88] pioneered in describing a refactoring tool, for the Smalltalk programming language. This tool offers a number of predefined refactorings, each of which including a number of preconditions. If the program subject to refactoring satisfies such preconditions, the tool guarantees that the transformation preserves the correctness and behavior of the resulting program. Roberts also argues that refactoring performed by a tool may even depreciate up-front design as consolidated in modern methodologies, since it reduces the cost of the overall process, leading to cheaper design changes after implementation has taken place.

In the context of Java, foremost tools, such as Eclipse [30] and IntelliJ IDEA [62], offer advanced support for refactoring programs, from changes as simple as moving and renaming members to more elaborate modifications, such as Extract Interface and Generalize Type. For the Eclipse IDE, Tip et al. [97] devised a novel approach for implementing refactorings with subtyping, using static properties of class hierarchies. The authors realized that, even though some refactorings having preconditions not fully satisfied by a program, they may be applied to program fragments presenting properties related to generalization. These properties are type constraints, which provide formal support so that some refactorings can be applied safely, such as Extract Interface and Pull up Members [97]. The type constraints – statically extracted from source code –

help the tool in identifying the correct program fragments that may be changed, ruling out the ones that might produce incorrect results. This approach is an example of how research is being carried out for improving confidence on refactoring tools.

2.3.2 Behavior Preservation in Refactoring

In refactoring, the observable behavior of the program must be maintained, outputting the same results for specific input data. Usually tool support approaches are not concerned with mathematical guarantee that the implemented refactorings preserve behavior (program semantics). There is no proof that, by satisfying its preconditions, a refactoring preserves semantics. This is important for the refactoring practice and for justifying the validity of changes accomplished by refactoring tasks. In the context of formal methods, laws of programming address this issue of behavior preservation. Laws include two transformations, and one of these can be considered a refactoring, in the sense that they may be a step to improve quality factors in a program. In fact, laws formalize some of the primitive refactorings shown in Roberts' work [88].

A notion of equivalence or refinement between programs is important in the context of laws of programming [56, 16], in order to refactor programs or support stepwise development. This establishes an algebraic approach, which consists in postulating general properties of the language constructs, typically as laws that relate language constructs. Further, equational reasoning, which can be easily automated by term rewriting, is immediately available as a framework for reasoning and transforming programs.

However, in order to avoid postulating an unsound set of laws, these laws must be linked to a formal semantics in which the laws can be proved. Once the laws have been proved, in whatever model, they should serve as tools for carrying out program transformations. Laws are called *algebraic*, since they are presented as *equations*, possibly with *provisos* (that must be satisfied for the successful application of the law). Two program transformations are defined by each law, taking a program to a (possibly) alternative program that presents the same semantics, in each direction. Such laws can be valuable not only for reasoning about programs, but also for designing correct compilers [89] and supporting informal programming practices, such as refactoring.

In order to illustrate the use of the algebraic approach in imperative programs, consider two laws related to the assignment command (:=) in an imperative language. **skip** denotes a command having no effect, always terminating, whereas x and y denote sets of variables and e and f equal-length list of expressions. The following law states that the assignment of the value of a variable to itself has no effect.

Law $\langle$*void assignment*$\rangle$

$$(x := x) = \textbf{skip}$$

As a consequence, this void assignment can also occur as part of a multiple assignment, as formalized in the following law.

Law $\langle$*identity assignment*$\rangle$

$(x, y := e, y) = (x := e)$

Since they are extensively used in this thesis, other examples of laws of programming – mainly for object-oriented programs – can be seen in Section 4.2.

2.4 Refinement Calculus

In addition to using equivalence laws, program refactoring may require programs that do not behave exactly as the original ones, but possibly better (refinement). For this purpose, an ordering relation on programs is considered: $p_1 \sqsubseteq p_2$ holds when p_2 is at least as good as p_1 in the sense that it will meet every purpose and satisfy every specification satisfied by p_1. This relation may also be regarded as p_2 possibly reducing nondeterminism in p_1.

Nondeterminism can be understood as allowing choices to be made. Formal program development usually starts with abstract specifications which leave several design decisions for the programmer to take. In order to explore the power of nondeterminism for specifying program behavior, it is suitable to embed a more abstract specification notation into a programming language. As a single notation is used both for programming and specification, program development reduces to transformations of specifications within a uniform framework. Examples of these approaches are the refinement calculi by Back [6] and Morgan [77]. An interesting feature from the latter is the specification statement, $w : [pre, post]$, which describes a program that, when executed in a state satisfying the precondition predicate *pre*, terminates in a state satisfying the postcondition predicate *post*, possibly modifying the values of variables in the list (frame) w.

Morgan's calculus includes several laws that allow transforming specification statements into executable programs. Some laws relate specification statements, defining a process known as algorithmic or control refinement. Two of these laws formally capture the notion of refinement in program development. One states that a program can be made more applicable (defined for a larger domain or set of states) when refined, weakening its precondition.

$w : [pre, post] \;\sqsubseteq\; w : [pre', post]$
provided $pre \;\Rightarrow\; pre'$

Concerning the postcondition, refinement might lead to a more deterministic or predictable program, closer to possible execution. The variable w must be replaced by w_0 in the formulation due to the convention that initial values of framed variables in the postcondition are subscripted.

$w : [pre, post] \;\sqsubseteq\; w : [pre, post']$
provided $pre[w\_0/w] \;\wedge\; post' \;\Rightarrow\; post$

2.4.1 Data refinement

Complementary to algorithmic refinement, a different type of refinement involves change of data representation, which is particularly useful for refactorings [26]. As a program evolves, the initially defined abstract data types, which may not be even available in the target programming language, give rise to more concrete representations, supporting stepwise development.

Intuitively, the program encompassing the concrete representation refines its abstract counterpart, although attempting to prove this refinement using the laws for algorithmic refinement reveals that a direct comparison between the two programs is not possible at all, due to its different data spaces. The missing connection is a relation between the abstract and the concrete states; if this relation is functional, it is known as abstraction function.

As a simple example, consider the specification statement that adds a new element to a set.

$$s : [e \notin s, s = s_0 \cup \{e\}]$$

Then consider a possible implementation using a sequence, as a concrete representation of a set.

$$t : [e \notin \mathsf{set}\ t, t = t_0 \frown \langle e \rangle]$$

The operation $\mathsf{set}\ t$ yields a set with the elements of the sequence t, $\langle e \rangle$ stands for the singleton sequence with element e, and $\frown$ represents sequence concatenation. In order to guarantee that this refinement is correct, the following abstraction function is appropriate.

$$s = \mathsf{set}\ t$$

As exemplified by Morgan's refinement calculus [77], data refinement is formulated at the level of programming modules. A module includes state variables, a state initialization, and procedures which act on the module state. Broadly, the technique involves adding the concrete variables to the module being data refined, making the abstract variables auxiliary, and then removing the auxiliary (abstract) variables. Encapsulated variables can then be added or removed, based on a *coupling invariant*, which relates the new and old variables, guaranteeing that the previous behavior is maintained.

2.5 Model Refactoring

As in other engineering fields, modeling can be a useful activity for tackling significant problems in software design, raising the level of abstraction. With models, developers can explicitly express the intent of how to address problems, information that is usually captured in an informal way, if captured at all, by traditional code-driven approaches. For instance, *object models* [70] structurally classify objects and their relationships, enriched by constraints over those objects. They are specially useful for analyzing system domains, giving a design for their main abstractions. As an example, we use a simple object model of a file system, as implemented in mainstream operating systems. We depict these model elements in Figure 2.1 as a class diagram in the Unified Modeling Language (UML) [15]. Directories and files are classes, representing two types of file system objects. `name` is a binary relation from file system objects to names, constrained to one multiplicity. Likewise, file system objects may have contents. `Root` is a subtype of `Directory`.

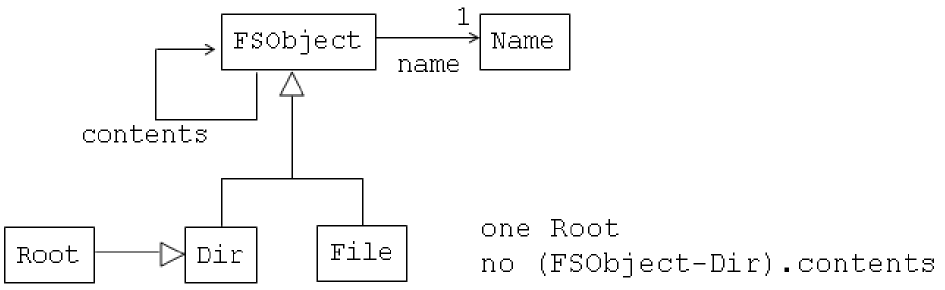


Figure 2.1: Object model for a file system

The invariants for the file system are represented with a logic-based syntax based on the Alloy language [58]. The formulas respectively state that there is only one root directory, and only directories have contents. The `one` keyword states that the expression yields exactly one object. The (-) symbol denotes set difference. The join operator (.) represents relational dereference. For instance, the expression `o.contents` yields the set of all file system objects for which there exists an element of `o` related to it by the `contents` relation (the `no` invariant states the emptiness of this set).

Many notations and languages exist to represent software models on different stages of the life cycle. Some notations may be visual, others textual, as long as they apply abstraction as a principle. Visual modeling notations are usually effective in providing documentation for human consumption. In terms of modeling languages, the UML has become a standard for communication and documentation during the software process. UML offers a notation for class diagram, in order to express object models; the language introduces constructions for classes of objects and relationships. Furthermore, UML includes a logical language, the Object Constraint Language (OCL) [98], which can be used to express logical constraints on models, such as preconditions, postconditions and invariants. OCL was introduced into UML as part of the standardization process, although not supported by tools as expected. For instance, the two invariants showed in Figure 2.1 may be written in OCL as well. On the other hand, Alloy [58] is a formal object-oriented specification language, used for specifying, verifying and validating (with tool support) properties about object models. As the language in our solution, Alloy is described in Chapter 3.

Similar to source code, model evolution may also lead to software entropy, lowering quality factors. In order to address this situation, refactorings can also be applied to models. Applying refactoring to models rather than to source code can encompass a number of benefits [38]. First, software developers can simplify design evolution and maintenance, since the need for structural changes can be more easily identified and addressed on an abstract representation. Second, developers are able to address deficiencies uncovered by model evaluation, improving specific quality attributes directly on the model. Third, a designer can explore alternative design decisions in a cheaper way. For instance, the introduction of patterns [39] can be approached as model refactoring.

As a recent research trend, several approaches have been proposed for refactorings targeting UML models [68, 47, 94]. Among those, a well accepted approach is to break a complex model refactoring into a set of small primitive semantics-preserving transformations. If the semantics holds during each transformation, the composite refactoring is considered to be semantics-preserving. In this thesis, the method applied for model refactoring is based on laws of modeling, as detailed in Chapter 3.

2.6 Conformance

Software models include system objects and mechanisms which implementations should conform to. For the context of this thesis, informally conformance consists in design decisions being fulfilled by all executions of a program. Regarding object models, we consider that a program is in conformance with a model when it meets all of the specified constraints throughout every execution. For the example in Figure 2.1, a brief description of a conforming program consists in not having more than one instance of root directories, and no file objects have contents. This relationship between models and programs can be further constrained with syntactic requirements. Also in the file system example, a syntactic conformance constraint could enforce that a program class `FSObject` declares a `contents` field, as specified in the object model.

In general, the challenge in co-evolution is to maintain models and implementations in conformance through evolving changes such as requirement shifts and refactorings. *Conformance checking* consists in automatically verifying whether a program is a valid implementation for a given model, complementing traditional co-evolution support. This approach is becoming practical in several contexts, as conformance checking tool support is in constant evolution. Several techniques have been applied for checking conformance, including code inspection, testing and formal verification. These techniques can be separated into two categories: static or runtime checking.

Static checking only applies to the implementation's source code. In other words, static conformance checking can be used to check the consistency of a program's code against models. Such an approach has two key elements: the language for the models themselves and the analysis mechanism by which the constraints are checked. Alternatively, runtime checking makes use of information available during the implementation's execution; it is not limited to artifacts available at compile time. An execution may not cover all functionality of an implementation, leading to a less comprehensive conformance checking. In addition, this analysis may interfere with the implementation's execution, which is slowed down by the checking itself. The analysis may be limited to specific user-defined breakpoints.

As an alternative to keeping models and programs as distinct artifacts and facilitate conformance checking, annotation languages combine invariants and source code into one artifact. The Java Modeling Language (JML) is probably the most important exemplar [18]. JML is a language for specifying assertions about the design of Java classes and interfaces, including pre- and post-conditions on methods, using a Java-like syntax. The following code fragment shows examples of JML annotations as commentary within a Java class named `Person`. An invariant is a conjunction of properties that must hold for every `Person` object, indicated by the JML identifier `invariant` – in this case, no person may present an empty string as name or a negative weight. In addition, pre- and post-condition for the `addKgs` method, which increases a person's weight, are denoted by the JML identifiers `requires` and `ensures`, respectively. Before the method execution, the sum of the current weight and the additional kilograms must be non-negative – otherwise the class invariant could be broken. Also, after its execution, the method must yield an object state in which the new weight is defined from the sum of the previous weight value with the parameter.

```
class Person {
  private String name;
  private int weight;
  /*@
    @ invariant !name.equals("") && weight >= 0;
    @*/

  /*@
    @ requires weight + kgs >= 0;
    @ ensures weight == \old(weight) + kgs;
    @*/
  public void addKgs(int kgs){..}
}
```

These assertions can be compiled into expanded runtime checks. JML's runtime assertion checker tool [100] is composed of two components: a translator yielding a program with extra Java code to perform the runtime checks, and an assertion checker, used during the program's execution. Annotation languages may be also subject of static conformance checks, warning the user about possibly erroneous program fragments such as null dereferences and array overflow errors [10, 36].

2.7 Model-driven Development

Model-Driven Development (MDD) consists in the application of models and related technologies to raise the level of abstraction at which developers create and evolve software, for both simplifying and formalizing several development activities, making automation possible [54]. Models may help the mechanization of general software practices; this turns them into an instrument to increase productivity in the following situations [49]:

- Parts of the source code can be automatically generated from models. The degree of completeness of the generated source code is reliant on models' level of abstraction and the specificity of the modeling language;

- Routine tasks, such as generating one artifact from another, may be fully automated. For instance, test cases and scripts can usually be produced from structural and behavioral models;

- Abstract descriptions as models may be used in migrating software to distinct implementation technologies with minimum manual work.

In fact, model-driven approaches [64] go further in promoting models to first-class development artifacts. As such, they explicitly convey intent in a formal way, so tools can interpret this information to provide more advanced forms of automation. The manipulation of models may bring faster and more accurate responses to changes in requirements, as developers deal with abstract descriptions instead of getting into source code details. However, the effectiveness of such process in avoiding source code manipulation, partially or completely, is still limited.

Model-driven Architecture (MDA) [66] is a particular realization of MDD that distinguishes models that include details of the implementation technology from technology-independent models, as described in more detail in the following section. However, MDA represents only one view of MDD, perhaps the most prevalent one; others include Agile MDD [2], Software Factories [49] and Domain-oriented programming [96].

2.7.1 Model-Driven Architecture

The Model-Driven Architecture (MDA) [66] involves manipulation of abstract models in UML-based languages and, backed by specific patterns, generates platform-specific models by automatic transformations. Developing an application with MDA consists in first building platform-independent models (PIMs) at the conceptual level, including specification of business functionality and behavior; then a tool is used to adapt these models to a target platform, resulting in platform-specific models (PSMs). Finally, source code is generated for the target platform from PSMs. Interface definitions are included for specifying how a base model is implemented on a different middleware platform.

The key idea behind MDA is the separation of the application logic from the platform it is intended to execute, in order to minimize the impact of technology evolution on development. The set of technologies for MDA are positioned around UML-based languages, which is considered impractical by some authors [49, 64], due to UML's lack of formal semantics and inappropriate language definition.

Further, MDA encourages the use of UML as a development language with executable semantics [7], using action semantics [1] or statecharts. This "executable code" is realized either in a high-level programming language or by direct compilation to executable representations like assembly languages. Therefore, the accepted idea of abstraction in MDA is linked to implementation platforms, not programming logic. The latter is included as part of the models itself.

2.8 Co-Evolution Tools

As the importance of software modeling advances, issues with software evolution with models become more apparent. As models are subject to evolutionary changes, their supported implementation should be able to migrate jointly towards a desired target state, for documenting the design decisions appropriately and increasing productivity. Even in MDD approaches, model evolution and its implications to related artifacts have not been explored in depth. Evolving models can bring value to software development, since dealing with abstractions minimizes complexity and makes design explorations cheaper. As the idea of refactoring models adds simplicity to software evolution, automation and semantics preservation are even more complex issues regarding models and their implementations.

In practice, models are hardly kept up-to-date. Because models are not the main artifact, they are not forced to change as software changes, quickly becoming obsolete. Models produced during early stages of analysis and design, for example, are discarded when source code becomes the primary artifact, given that the cost of conformance may outweigh the benefits that models provide – a trade-off between dealing with complexity (models) and dealing with change (evolution) [49].

Methods and tools for partially or even totally removing human interaction in the process are valuable to evolution, thus automatic support to address evolutionary change is highly desirable. This practice enables developers to reason and develop on models, and changes to models can be consistently transferred to source code, being the basis of MDD. In the context of embedded systems, for example, this is a common practice, since the domain is precisely defined, allowing for generation of several systems from the same model, by changing model parameters or state transition diagrams [29]. In this context, developers must guarantee that programs conform to models. Several techniques are currently used, such as round-trip engineering and MDD tools, which are described in the following sections.

Round-Trip Engineering

One of the most popular techniques for model-program co-evolution is round-trip engineering (RTE), which comprises automatic consistency of changing software artifacts within a CASE tool. Changes to analysis and design models are propagated to programs through *forward engineering*, whereas changes to the implementation are propagated to models through *reverse engineering*. Although the relationship between structural models (e.g. UML class diagrams) and source code is widely explored by modeling tools, behavioral models are also becoming subject to model and code generation (especially method bodies) [79].

In RTE, forward and reverse engineering are used for ensuring that software artifacts become synchronized at specific points in time. RTE usually presents issues not seen in forward and reverse engineering performed in isolation. As stated by Sendall and Küster [90], those tasks in isolation are typically single transformations, where any updated information in the target artifact is not considered, possibly replacing the previous version. Differently, RTE requires that information in the target artifact is preserved, reconciled with the newly-performed changes. For example, in round-tripping UML class diagrams and Java programs, it is often undesirable that names and associations changed in the class diagram are not reflected in the Java program. In this case, traceability information must be persisted through transformation steps, as a way to recognize generated versus user-defined code; placing markers in the artifacts is a possible approach. Another approach, as described by Harrison et al [55], achieves a similar effect by using programming mechanisms (class hierarchies) to isolate edited source code.

Despite its benefits, limitations of RTE restrain model-program co-evolution. When development activities focus on program changes, RTE helps with generation of an initial program structure from models and consequent evolution of models by reverse engineering. This task generates concrete models from source code, promoting a graphical visualization of a program's structure and behavior. However, code visualization using general-purpose languages like UML can actually be less appealing than working directly with source code, as it is neither abstract enough to capture high-level design concerns nor sufficiently concrete to capture implementation details.

Advanced modeling tools are capable of maintaining traceability information, in order to simplify the generation of consistent models, although abstract business rules (e.g. multiplicity in relationships) may not be possibly recreated from source code. Usually inferring higher-level semantics from lower-level constructs is much harder than the inverse [54]. For this purpose, *program analysis tools* detect abstract information from programs. As an example, Daikon [31] is an implementation of dynamic detection of likely invariants, as it reports likely program invariants. An invariant is a property that holds at a certain point or points in a program, for analysis feasibility; these are often seen in assert statements, documentation, and formal specifications. The tool is based on dynamic analysis, in which program executions are analyzed, in order to detect what kinds of properties are maintained during a set of executions.

Conversely, when models are changed, only concrete models may be able to yield nearly-complete programs, when general-purpose modeling languages are used. In addition, it is unclear how specific program logic can be maintained after model changes, as programs regenerated from model changes may require manual updates for consistency. A usual technique marks previously edited code fragments as immutable, in order to avoid problems that might be caused by code regeneration. Even so, this marked source code may depend on design structures that were modified in the model, resulting in overwriting changes to the generated artifacts when regeneration occurs. In this scenario, models and programs end up outdated, as manual updates become more expensive.

MDA Tools

MDA tools represent an alternative to conventional RTE. As developers deal with models, a MDA-compliant tool is assumed to be able to generate PSMs and source code. The transformation process is driven by a number of patterns and source code idioms, allowing advanced program generation. We present below a brief description of the most used MDA tools, along with a summarized analysis on their support to evolution. Code generation and modeling tools developed prior to the specification of MDA are not considered MDA tools, although they may be applicable to an MDA process.

OptimalJ. Built on Eclipse [30], the tool aims to generate a J2EE [93] application from business rules specified as PIMs [25]. The code editor limits editing on free blocks of generated code; if a portion of the application needs remodeling, models are edited and code is regenerated. As long as edits were limited to these free blocks, which often is not the case, changes are preserved.

ArcStyler. The tool also concentrates on the generation of Java applications, with option also for Microsoft's .NET platform. It establishes a cartridge architecture, in which designers define specific transformations for generating PSMs for custom platforms [80]. It is, to our knowledge, the only MDA tool that support invariants in OCL.

Rational Software Architect. Conventional model editing and RTE is observed in this tool, with subtle differences. Most importantly, after code generation, the tool renders the existing code into UML form – more a code visualization than a PIM – and when changes happen that affect the design, the UML views are instantly updated [91]. IBM argues that this is, in fact, how many of its customers work in practice today with its legacy modeling tools [20].

AndroMDA. It is an open source code generation framework that follows the MDA paradigm [3], which also adopts a cartridge-based architecture. It takes any number of models (usually UML models stored in XMI produced from case-tools) combined with any number plug-ins (cartridge and translation-libraries) and produces custom components. AndroMDA is mostly used by developers working with J2EE technologies [3].

In conclusion, MDA tools seldom support OCL invariants, which cannot be enforced in the underlying source code, limiting model expressiveness. Furthermore, although these tools provide models as a clearer abstraction from programs, they are not sufficiently mature to avoid changes made directly to source code. This scenario is due to the fact that general-purpose modeling languages are not able to generate the complete executable code. In this scenario, RTE problems recur, such as model evolution that may affect previously edited source code, leading to incorrect programs. As an option to these approaches, executable models require fluency in UML for the full benefits, with a low level of abstraction.

Chapter 3

Alloy and Object Model Refactoring

This chapter describes the modeling language considered to establish our theoretical approach. Alloy is the formal object-oriented language used to specify object models for model-driven refactoring. Besides presenting the language, we give special attention to its catalog of algebraic laws, which build the basis for refactoring in terms of semantics-preserving transformations.

3.1 Language

Alloy is a formal object-oriented modeling language, based on first-order logic and a notation called *relational calculus*, that gives a mathematical notation for specifying objects and their relationships [58]. Alloy models are similar to UML class diagrams combined to Object Constraint Language (OCL) [98], but Alloy has a simpler syntax, type system and semantics, being designed for automatic analysis, which motivated us to choose the language for this research work. The choice of this language is justified by our focus on a formal investigation. Nevertheless, we believe that most contributions can be extended almost directly to other popular languages, like UML and OCL [**?**]. In fact, additional work by our research group prepares the ground for transferring results to UML [72, 73].

The language assumes a universe of elements partitioned into subsets, each of which associated with a defining type. An Alloy model contains a sequence of *paragraphs*; one kind of paragraph is called a *signature*, which is used for defining a new type. These instances can be related by *relations* declared in the signatures. A signature paragraph introduces a basic type and a collection of relations, along with their types and other constraints on the values that they relate.

For the file system object model showed in Figure 2.1, the following Alloy fragment defines signatures for `FSObject` and `Name`. The keyword **sig** declares a signature with a name. Signature `Name` is an empty signature, while `FSObject` declares two relations. For example, every instance of `FSObject` is related to exactly one instance of signature `Name` by the relation `name` – the keyword **one** denotes a total function. Also, every file system object may have `contents`; it is optional, and maybe contains more than one instance, since it is annotated with the **set** keyword, which establishes no constraints on the relation.

```
sig Name {}
sig FSObject {
  name: one Name,
  contents: set FSObject
}
```

In Alloy, one signature can extend another – with the **extends** keyword –, establishing that the extended signature is a subset of its supersignature. Signature extension introduces a subtype, establishing that each subsignature is disjoint. The following fragment shows signatures called `File` and `Dir`, that are disjoint (`FSObject` instances may be exclusively files or directories). In addition, `Root` is a subtype of `Directory`.

```
sig File extends FSObject {}
sig Dir extends FSObject {}
sig Root extends Dir {}
```

This object model can be further constrained with invariants, for complex domain rules concerning the declared signatures and relations. For this purpose, an Alloy model can be enriched with formula paragraphs called *fact*, which is used to package formulae that always hold for the model. The following fragment introduces a fact establishing general properties about the file system. The first formula states that there must be exactly one `Root` instance in every file system, by the **one** keyword. Next, the second formula defines that from all file system objects, only directories may present contents. In the expression `(FSObject-Dir).contents`, the join operator (.) represents relational dereference (in this case, yielding contents from the set of instances resulting from `FSObject` instances that are not directories, with symbol `-` as set difference). The `no` keyword establishes that the expression that follows results in an empty set, which gives the constraint the following meaning: only directories have contents in the file system.

```
fact {
  one Root
  no (FSObject-Dir).contents
}
```

Other formula paragraphs can be used in order to build parametric constraints, namely *predicates and functions*. The following predicate defines an optional constraint to the file system, stating that file system objects are files or directories only; `+` is the set union operator. In other words, `FSObject` could be an abstract signature, given the following predicate is used in a fact in the model.

```
pred abstract() {
  FSObject = File + Dir
}
```

Alloy was simultaneously designed with a fully automatic tool that can simulate models and check properties about them (the Alloy Analyzer [59]). The tool translates the model to be analyzed into a boolean formula, and this formula is solved using SAT solvers. The analysis consists in binding instances to signatures and relations, searching for a combination of values that make the translated boolean formula true.

One of the analysis, namely *simulation*, generates structures without requiring the user to provide sample inputs or test cases. If the tool finds a configuration of instances making the formula true, a valid *interpretation* for the model is determined. In our example, we can use the `abstract` predicate in order to simulate the model in the tool, with the `run` command. The complement `for 3` constrains the simulation to work on a scope of at most three instances for each signature; the analysis is limited to a number of instances, being sound and complete up to that specific number [59].

```
run abstract for 3
```

In this example the model is consistent, as at least one interpretation is found, as showed in Figure 3.1, excerpted from the actual tool output. Each box represents an instance with the name of the corresponding signature; for more than one instance, numbers are concatenated to the signature name. A major benefit of this analysis is that modeling errors, such as missing constraints, can be easier to find by visualizing possible interpretations. In this case, two design decisions commonly seen in file systems are unspecified: a directory having itself as contents (`Root`) and orphan files lacking a parent directory (`File1`).

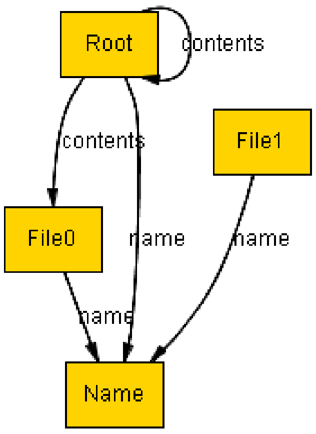


Figure 3.1: An interpretation for the file system model

These two problems can be tackled by adding two more invariants as a fact to the model. The first states that no directory can be related to itself by `contents`, *even indirectly*; the symbol ^ yields the *transitive closure* of the relation. Additionally, the second invariant defines that every file is contained within a directory; keywords **some** and **in** represent existential quantification and element inclusion in a set, respectively. While **all** represents the universal quantifier, **no** represents its negation.

```
fact {
  no d:Dir | d in d.^contents
  all f:File | some d:Dir | f in d.contents
}
```

If another execution is performed in the model, the yielded result is depicted in Figure 3.2, confirming the effects of the introduced invariants.

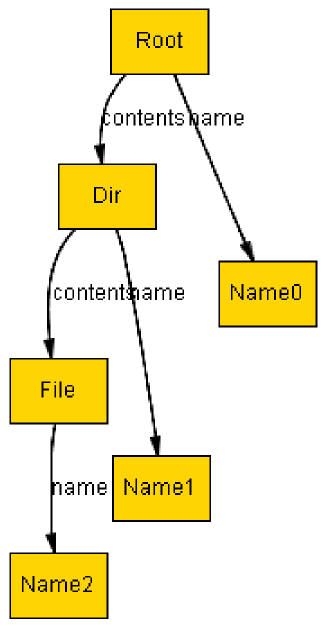


Figure 3.2: Interpretation for the modified model

3.2 Equivalence notion

Proposing primitive laws for Alloy demands a way to compare whether two models have the same meaning. An equivalence notion was defined for Alloy [43], supporting abstraction from names and elements when comparing models. This equivalence notion is applicable to any language used to represent object models.

Consider the pair of Alloy models depicted in Figure 3.3 (drawn as UML class diagrams), presenting a small variation from the file system object model. Figure 3.3(a) shows a model stating that each directory is related directly to a set of file system objects, while Figure 3.3(b) establishes that each directory is related to a collection, which is then related to a set of file system objects.

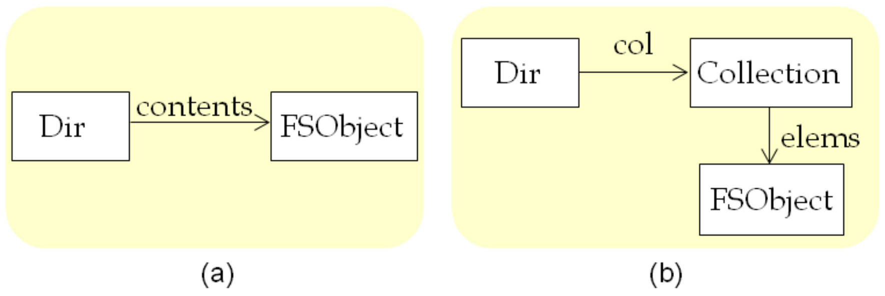


Figure 3.3: Two alternative object models for the file system

The traditional equivalence notion, which compares whether two models have the same semantics, is useful, but not flexible enough to compare equivalent models with auxiliary elements such as `Collection`, or with different forms of representing the same concept, such as `contents` in Figure 3.3(a). Here we are interested in verifying whether they have the same semantics and are, therefore, equivalent alternative designs. The models are intuitively equivalent, taking into consideration the relationship between directories and file system objects, which is maintained whether there is an intermediate collection or not.

In order to compare models in such scenario, a flexible equivalence notion compares the semantics of two object models only for a number of relevant model elements (signatures and relations), abstracting the values assigned to the others. The set of relevant elements is called alphabet (Σ). The names that are not in the alphabet are considered auxiliary, or not relevant for the comparison. Assume that Σ contains only the `Dir` and `FSObject` names. If both models have the same instances in valid interpretations for those names, they are considered to be equivalent under this equivalence notion. Other names, such as `col`, `Collection` and `elems`, are regarded as auxiliary, thus not considered when searching for an equivalent interpretation in the corresponding model. Accordingly, we may now compare the models depicted in Figure 3.3.

However, there might be model elements that, although relevant, cannot be compared, since they are not present in both models. For instance, suppose that we include `contents` to Σ. In this case, we cannot compare the models, since `contents` is not part of the model in Figure 3.3(b). However, `contents` can actually be expressed as a composition between `col` and `elems`. In those cases, a mapping is considered, called view (v), establishing how an element of one model can be interpreted using elements of another model. Views consist of a set of items such as $n \rightarrow exp$, where n is an element's name and exp is an expression, specifying how the concept n can be expressed in terms of other concepts. Notice that although the values of auxiliary names are not compared, they can be used to yield an alternative meaning to relevant names. In the example, we may choose a view containing the following item: $contents \rightarrow col.elems$. Now we can infer that both models are equivalent.

Two models can be equivalent with respect to an alphabet and a view (σ, v), and the choice of a different alphabet or view yields a different equivalence. This notion is more flexible, it supports abstraction from names and elements when comparing models. By choosing specific alphabets and views, as desired, the developer can choose the appropriate abstraction level for a given situation. In fact, the usual equivalence notion is a particular instantiation of our notion when we simply take an empty view and an alphabet containing all names in the model. More details about the formalization of this equivalence notion, its properties and proofs can be found elsewhere [42, 43].

3.3 Laws and Refactoring

Using the previous equivalence notion, a catalog of primitive laws for Alloy is presented. These laws state properties about several constructs of the language, including signatures, relations and facts. Here we only show a subset of the laws proposed for Alloy; a comprehensive catalog has been proposed [42], providing powerful guidance on the derivation of complex transformations, such as model refactorings and optimizations, in addition to being the basis for our model-driven approach to refactoring.

Each law defines two semantics-preserving transformations. As an example of a law, we can introduce a new relation along with its definition, which is a formula of the form $r = exp$, establishing a value for the relation, as formalized in Law 1. We can also remove a relation that is not being used. *rs* represents relations, while *forms* represents a set of formulas. Below the law, provisos are specified, which must be valid for the law to be correctly applied. ($\leftarrow$) defines provisos for application from Left-to-Right (L-R), while ($\rightarrow$) defines provisos for applying the law from Right-to-Left (R-L). $\leftrightarrow$ defines provisos for both directions. *ps* denotes a set of signature and fact paragraphs that are not showed in the law template.

Law 1. ⟨*introduce relation and its definition*⟩

```
ps
sig S {
  rs
}
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
  rs,
  r : set T
}
fact F {
  forms
  r=exp
}
```

provided
($\leftrightarrow$) if r belongs to Σ, r does not appear in *exp* and v contains the $r \rightarrow exp$ item;
($\rightarrow$) (1) the family of S does not declare any relation named r; (2) T is either S or declared in *ps*; (3) r does not appear in *exp*, or *exp* is r; (4) *exp* is a subtype of r in *ps* and *forms*; (5) for all names in Σ that are not on the right-hand side model, v must have exactly one valid item for it;
($\leftarrow$) (1) r is not mentioned in any constraints within *ps*; (2) for all names in Σ that are not on the left-hand side model, v must have exactly one valid item for it.

The family of a signature is the set of all signatures that extend or are extended by it direct or indirectly. The *exp* expression can be either r or an expression having the same type of r and not containing r. This law can be used to simply introduce a relation, without any definition. It is only required to assign *exp* the r relation itself, introducing a tautology that is later removed. Moreover, constraints involving Σ and v must be considered. When introducing or removing a relation in Σ, the $r \rightarrow exp$ item belongs to v and r does not appear in *exp* in order to avoid a recursive definition in v.

Law 2 introduces transformations for adding or removing a subsignature to an existing hierarchy. We can add an empty subsignature if declared with a fresh name. After this transformation, the supersignature becomes abstract (defining no direct instances), as denoted by the resulting invariant (X=U−S−T). Similarly, the subsignature can be removed if it is not being used elsewhere and, in order to avoid type errors, there is no invariant using its type ($exp \leq U, exp \nleq S$ and $exp \nleq T$), where $\leq$ denotes subtype.

Law 2. $\langle$*introduce subsignature*$\rangle$

$$\boxed{\begin{array}{l} ps \\ \textbf{sig}\ \ U\ \ \{\ \ rsU\ \ \} \\ \textbf{sig}\ \ S\ \ \textbf{extends}\ \ U\{rsS\} \\ \textbf{sig}\ \ T\ \ \textbf{extends}\ \ U\{rsT\} \\ \textbf{fact}\ \ F\ \ \{forms\} \\ \\ \\ \\ \end{array}} =_{\Sigma,v} \boxed{\begin{array}{l} ps \\ \textbf{sig}\ \ U\ \ \{rsU\} \\ \textbf{sig}\ \ S\ \ \textbf{extends}\ \ U\{rsS\} \\ \textbf{sig}\ \ T\ \ \textbf{extends}\ \ U\{rsT\} \\ \textbf{sig}\ \ X\ \ \textbf{extends}\ \ U\{\} \\ \textbf{fact}\ \ F\ \ \{ \\ \quad forms \\ \quad X = U - S - T \\ \} \end{array}}$$

provided
($\leftrightarrow$) if X belongs to Σ, v contains the $X = (U - S - T)$ item;
($\rightarrow$) (1) ps does not declare any paragraph named X; (2) there is no signature in ps that extends U (3) for all names in Σ that are not on the right side model, v must have exactly one valid item for it;
($\leftarrow$) X does not appear in ps, rsU, rsS, rsT and *forms*; (2) there is no expression exp, where $exp \leq U$ and $exp \leq S$ and $exp \leq T$, in ps or *forms* or any valid item in v; (3) for all names in Σ that are not on the left side model, v must have exactly one valid item for it.

In addition to the presented laws, a catalog of laws of modeling for Alloy was proposed [42], as listed in Table 3.1. This catalog has been proved to be semantics preserving with the help of a theorem prover; also, a proof of compositionality is also provided [42].

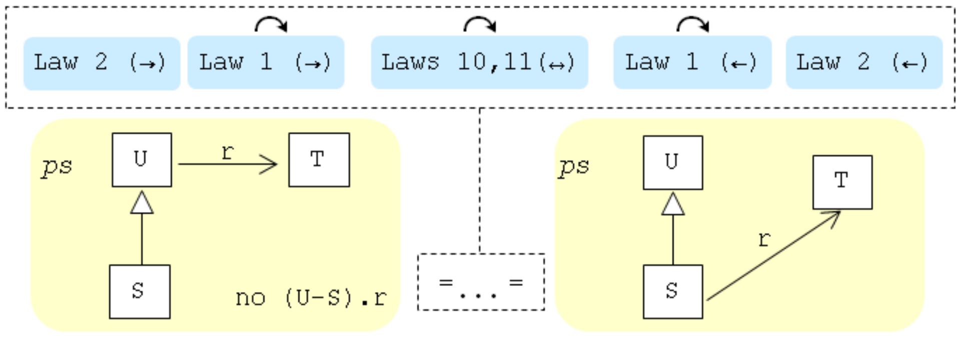
1. introduce relation and its definition	12. separate relation declarations
2. introduce subsignature	13. replace relation expression
7. introduce signature	14. introduce formula
8. introduce generalization	15. introduce empty fact
9. remove abstract qualifier	16. split relation
10. remove signature cardinality qualifier	17. remove one relation
11. separate signature declarations	18. remove lone relation

Table 3.1: Alloy laws

The full catalog is presented in Appendix A. Furthermore, these laws can be used as basis for several applications that require semantics-preserving transformations, for instance *model refactorings*. Since laws are simpler – dealing with a few language constructs – they can be more easily proved to be semantics-preserving. By construction, a composition of laws is also correct, providing safe refactorings for object models.

Refactoring 1 depicts, in UML, a refactoring rule for pushing down a relation down the hierarchy. As defined in previous work, a relation may be pushed down if the model presents an invariant stating that the relation only relates objects in the subclass s. A relation may be pulled up by adding this constraint to the model.

Refactoring 1. ⟨*Push Down Relation*⟩



provided
(→) $exp.r$, where $exp \leq U$ and $exp \not\leq S$, does not appear in ps;
(←) S's family does not declare any relation named r.

This refactoring is derived by successive applications of Laws 1 and 2 mainly. It must be clear that this derivation is carried out only once by a refactoring designer; once done, the refactoring is ready to be applied as enunciated in Refactoring 1. In the file system example, the developer would be able to apply this refactoring for pushing down the `contents` relation to `Dir`, since the required invariant is explicitly defined in Figure 2.1 (`no (FSObject-Dir).contents`). The application of Refactoring 1 entails the following law transformations (we take $\{FSObject, Dir, File, contents\}$ as the alphabet, and a view $v = \{contents \mapsto contentsDir\}$):

1. Apply Law 2 (L-R) to introduce an auxiliary `X` subclass of `FSObject`, with which `X= FSObject-Dir-File`;

2. Introduce three relations – Law 1(→) – from each subclass to `FSObject`, namely `contentsDir`, `contentsFile` and `contentsX`;

3. Deduce and add invariant `contents`=`contentsDir` + `contentsFile` + `contentsX` by using Law 14 from Table 3.1;

4. Remove `contents` based on the deduced formula, with Law 1(←);

5. Deduce that `contentsFile` and `contentsX` are always empty, based on the original invariant `no (FSObject-Dir).contents`;

6. Remove `X` with Law 2(←);

7. Rename `contentsDir` to `contents`.

Chapter 4

Programming Language and Refactoring

In this work, we use a small object-oriented language inspired by the language presented in Banerjee and Naumann's work [8], which we call *BN* (from the authors' last name initials). For establishing a formal basis for program refactorings in our model-driven refactoring approach, we use *laws of programming* [56] as a basis for formal program transformations. An extensive catalog of behavior-preserving laws of object-oriented programming has been defined for ROOL [16], another subset of Java with a formal semantics [21]. However, the language presents a copy semantics, restraining reasoning for modern programming languages such as Java and C#, which are based on references. On the other hand, BN follows Java's reference semantics. Also, we apply confinement as a required program property, since some of the laws are incorrect with references without confinement, as shown in this section.

4.1 Language

A program is a set of classes, namely a *class table* CT, which always include a class named `Main`, with a method `main` representing the execution starting point. A generic class declaration is defined as follows

$$\textbf{class } C \textbf{ extends } D \ \{ \ \bar{T}\ \bar{f};\ \bar{M} \ \}$$

where barred identifiers like $\bar{T}$ indicate finite lists. $\bar{T}\ \bar{f}$ stands for typed fields in the class, while $\bar{M}$ represents a list of methods.

The grammar, showed in Table 4.1, is based on given sets of class names (with typical element `C` and including at least `Object`), field names (codef), method names (`m`), and names (`x`) for parameters and local variables. In most respects **self** and **result** are like any other variables, but **self** cannot be the target of assignment.

The following code fragment declares the `FSObject` class from the file system example. Fields are private and methods are public by default, although public fields can be declared with the **pub** modifier. Also, a constructor can be declared with the **constr**

Table 4.1: RN Grammar [8]

$T ::= \textbf{bool} \mid \textbf{unit} \mid C$	data type
$CL ::= \textbf{class}\ C\ \textbf{extends}\ C\{\bar{T}\ \bar{f};\ \bar{M}\}$	class declaration
$M ::= T\ m(\bar{T}\ \bar{x})\{S\}$	method declaration
$S ::= x := e \mid e.f := e$	assign to variable, to field
$\mid x := \textbf{new}\ C$	object construction
$\mid x := e.m(\bar{e})$	method call
$\mid T\ x := e\ \textbf{in}\ S$	local variable block
$\mid \textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{fi} \mid S;\ S$	conditional, sequence
$e ::= x \mid \textbf{null} \mid \textbf{true} \mid \textbf{false} \mid \textbf{it}$	variable, constant
$\mid e.f \mid e = e$	field access, equality test
$\mid e\ \textbf{is}\ C \mid (C)e$	type test, cast

keyword. The **set** modifier establishes a collection variable (set of objects). Keywords **result** and **self** denote a method's return variable and the current object, respectively.

```
class FSObject{
   Name name;
   pub set FSObject contents;
   constr { self.name:= new Name }
   Name getName() { result:= self.name }
   ...
}
```

As in Java, there is also a class **object** with no fields or methods. Only **bool** and **unit** (the empty type) are predefined as primitive types in the language; other primitives can be similarly defined by the language designer, such as integers. Furthermore, expressions do not have side effects. Object construction occurs only as a command – `x:=` **new** `C`. This assigns the constructed object to a local variable. Similarly, method calls occur in special assignments `x:= e.m(`$\bar{e}$`)` defining both side effect and a return value. Methods can be defined recursively, so loops are omitted.

For the direct superclass of C, we define $superC = D$. Let M be in the list $\bar{M}$ of method declarations, with

$$M = T\ m(\bar{T}_2\ \bar{x})\{S_2\}.$$

Typing information is recorded by defining $mtype(m, C) = \bar{T}_2 \to T$ ($\bar{T}_2 \to T$ is not a data type in the language). For the parameter names, $pars(m, C) = \bar{x}$. If m has no declaration in C but $mtype(m, D)$ is defined, then m is an inherited method, for which we define $mtype(m, C) = mtype(m, D)$ and $pars(m, C) = pars(m, D)$. For any variable, $type(x)$ denotes its declaration type.

A typing context Γ is a finite mapping from variable and parameter names to data types, such that $\textbf{self} \in dom\Gamma$. Typing of commands for methods declared in class C is expressed using judgements $\Gamma \vdash S$, where $\Gamma\textbf{self} = C$. Further, if $mtype(m, C) = \bar{T} \to T$ and $pars(m, C) = \bar{x}$ then $\Gamma\bar{x} = \bar{T}$ and $\Gamma\textbf{result} = T$. The judgement $\Gamma \vdash e : T$ states that expression e has type T.

Table 4.2: RN Semantics of Expressions [8]

$$
\begin{array}{lcl}
[\![\, \Gamma \vdash x : T \,]\!]\,(h,\eta) & = & \eta x \\
[\![\, \Gamma \vdash \textbf{null} : B \,]\!]\,(h,\eta) & = & \mathit{nil} \\
[\![\, \Gamma \vdash \textbf{it} : \textbf{unit} \,]\!]\,(h,\eta) & = & \mathit{it} \\
[\![\, \Gamma \vdash \textbf{true} : \textbf{bool} \,]\!]\,(h,\eta) & = & \mathit{true} \\
[\![\, \Gamma \vdash \textbf{false} : \textbf{bool} \,]\!]\,(h,\eta) & = & \mathit{false} \\
[\![\, \Gamma \vdash e_1 = e_2 : T \,]\!]\,(h,\eta) & = & \text{let } d_1 = [\![\, \Gamma \vdash e_1 : T_1 \,]\!]\,(h,\eta) \text{ in} \\
 & & \text{let } d_2 = [\![\, \Gamma \vdash e_2 : T_2 \,]\!]\,(h,\eta) \text{ in} \\
 & & \text{if } d_1 = d_2 \text{ then } \mathit{true} \text{ else } \mathit{false} \\
[\![\, \Gamma \vdash e.f : T \,]\!]\,(h,\eta) & = & \text{let } \ell = [\![\, \Gamma \vdash e : (\Gamma\textbf{self}) \,]\!]\,(h,\eta) \text{ in} \\
 & & \text{if } \ell = \mathit{nil} \text{ then } \bot \text{ else } h\ell f \\
[\![\, \Gamma \vdash (B)e : B \,]\!]\,(h,\eta) & = & \text{let } \ell = [\![\, \Gamma \vdash e : D \,]\!]\,(h,\eta) \text{ in} \\
 & & \text{if } \ell = \mathit{nil} \vee \mathit{loctype}\ \ell \leq B \text{ then } \ell \text{ else } \bot \\
[\![\, \Gamma \vdash e \textbf{ is } B : \textbf{bool} \,]\!]\,(h,\eta) & = & \text{let } \ell = [\![\, \Gamma \vdash e : D \,]\!]\,(h,\eta) \text{ in} \\
 & & \text{if } \ell = \mathit{nil} \wedge \mathit{loctype}\ \ell \leq B \text{ then } \mathit{true} \text{ else } \mathit{false}
\end{array}
$$

Finally, a denotational semantics is defined for RN. Here we present a simplified version of the semantics described in detail in Banerjee and Naumann's work [8]. Our focus is on the semantics of expressions and commands, and the semantics of a class table. The state of a method in execution includes a heap h, which is a finite partial function from locations to object states, and a store η, which assigns locations and primitive values to the local variables and parameters given by a typing context Γ. An object state is a mapping from field names to values. Function application associates to the left, so $h\ell f$ is the value of field f of the object $h\ell$ at location ℓ.

A command denotes a function mapping each initial state (h_0, η_0) either to a final state (h, η) or to the distinguished value $\bot$. The improper value $\bot$ represents nontermination as well as runtime errors: attempts to dereference nil or cast a location to a type that it does not have. For locations, we assume that a countable set *Loc* is given, along with a distinguished value *nil* not in *Loc*. To track the class of an object we assume given a function $\mathit{loctype} : \mathit{Loc} \rightarrow \mathit{ClassNames}$ such that for each C there are infinitely many locations ℓ with $\mathit{loctype}\ell = C$. We write $\mathit{locs}C$ for $\{\ell \in \mathit{Loc} \mid \mathit{loctype}\ell = C\}$, and $\mathit{locs}(C\downarrow)$ for $\{\ell \mid \mathit{loctype}\ell \leq C\}$ ($\leq$ denotes subtyping). In addition, an allocator is a location-valued function *fresh* such that $\mathit{loctype}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}h$, for all C, h.

For expressions and commands, the semantics is defined by induction on typing derivations. We let $(h, \eta) \in [\![\, \mathit{Heap} \otimes \Gamma \,]\!]$ in the definitions in Table 4.1. For semantic values we use the identifier d, but sometimes ℓ for elements of the sets $[\![\, C \,]\!]$. For expressions the semantics is straightforward; the Java semantics is used for casts and tests.

Next, Table 4.1 shows the semantic definitions for commands in RN. In the semantics of commands, $[\mathit{fields} \mapsto \mathit{defaults}]$ is the abbreviation for the function sending each $f \in \mathit{dom}(\mathit{fields}\ B)$ to the default value for $\mathit{type}(f, B)$. The defaults are **false** for **bool**, **it** for **unit**, and *nil* for classes. Function update or extension is written like $[\eta \mid x \mapsto d]$.

Table 4.3: RN Semantics of Commands [8]

$$
\begin{array}{lcl}
\llbracket\, \Gamma \vdash x := e \,\rrbracket\, \mu(h,\eta) & = & \text{let } \ell = \llbracket\, \Gamma \vdash e : T \,\rrbracket\, (h,\eta) \text{ in } (h, [\eta \mid x \mapsto d] \\
\llbracket\, \Gamma \vdash e_1.f := e_2 \,\rrbracket\, \mu(h,\eta) & = & \text{let } \ell = \llbracket\, \Gamma \vdash e_1 : (\Gamma \mathbf{self}) \,\rrbracket\, (h,\eta) \text{ in} \\
 & & \text{if } \ell = \mathit{nil} \text{ then } \bot \text{ else} \\
 & & \text{let } d = \llbracket\, \Gamma \vdash e_2 : U \,\rrbracket\, (h,\eta) \text{ in} \\
 & & ([h \mid \ell \mapsto [h\ell \mid f \mapsto d]], \eta \\
\llbracket\, \Gamma \vdash x := \mathbf{new}\ B \,\rrbracket\, \mu(h,\eta) & = & \text{let } \ell = \mathit{fresh}(B, h_0) \text{ in} \\
 & & \text{let } h = [h_0 \mid \ell \mapsto [\mathit{fieldsB} \mapsto \mathit{default}]] \text{ in} \\
 & & (h, \eta_0[x \mapsto \ell]) \\
\llbracket\, \Gamma \vdash x := e.m(\bar{e}) \,\rrbracket\, \mu(h,\eta) & = & \text{let } \ell = \llbracket\, \Gamma \vdash e : B \,\rrbracket\, (h_0,\eta_0) \text{ in} \\
 & & \text{if } \ell = \mathit{nil} \text{ then } \bot \text{ else} \\
 & & \text{let } \bar{x} = \mathit{pars}(m, B) \text{ in} \\
 & & \text{let } \bar{d} = \llbracket\, \Gamma \vdash \bar{e} : \bar{U} \,\rrbracket\, (h_0,\eta_0) \text{ in} \\
 & & \text{let } \eta = [\bar{x} \mapsto \bar{d}, \mathbf{self} \mapsto \ell] \text{ in} \\
 & & \text{let } (h, d_1) = \mu(B)m(h_0, \eta) \text{ in} \\
 & & ([h, [\eta \mid x \mapsto d_1]) \\
\llbracket\, \Gamma \vdash S_1;\ S_2 \,\rrbracket\, \mu(h,\eta) & = & \text{let } (h_1,\eta_1) = \llbracket\, \Gamma \vdash S_1 \,\rrbracket\, \mu(h,\eta) \text{ in} \\
 & & \llbracket\, \Gamma \vdash S_2 \,\rrbracket\, \mu(h_1,\eta_1) \\
\llbracket\, \Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \,\rrbracket\, \mu(h,\eta) & = & \text{let } b = \llbracket\, \Gamma \vdash e : \mathit{bool} \,\rrbracket\, (h_0,\eta_0) \text{ in} \\
 & & \text{if (b) then } \llbracket\, \Gamma \vdash S_1 \,\rrbracket\, \mu(h_0,\eta_0) \\
 & & \text{else } \llbracket\, \Gamma \vdash S_2 \,\rrbracket\, \mu(h_0,\eta_0) \\
\llbracket\, \Gamma \vdash T\ x := e\ \mathbf{in}\ S \,\rrbracket\, \mu(h,\eta) & = & \text{let } d = \llbracket\, \Gamma \vdash e : U \,\rrbracket\, (h,\eta) \text{ in} \\
 & & \text{let } \eta_1 = [\eta \mid x \mapsto d] \text{ in} \\
 & & \text{let } (h_1,\eta_2) = \llbracket\, (\Gamma, x : T) \vdash S \,\rrbracket\, \mu(h,\eta_1) \text{ in} \\
 & & (h_1, \eta_2 \downharpoonleft x)
\end{array}
$$

The $\downharpoonleft$ symbol denotes domain restriction: if x is in the domain of η then $\eta \downharpoonleft x$ is the function like η but with x removed from its domain. Symbol μ represents a *method environment*, which keeps the semantics of all methods from the class table, being looked up in method calls.

4.2 Laws of Programming

One classical approach for defining semantics of a programming language is the algebraic approach, which consists in postulating general properties of the language constructs – *laws of programming* [56]. Program transformations are defined by laws, taking a program to a (possibly) alternative program that presents equivalent behavior. Such laws can be valuable not only for reasoning about programs, but also for designing correct compilers [89] and supporting informal programming practices, such as refactoring [26].

In this context, a comprehensive set of laws was developed for an object-oriented programming language known as the Refinement Object-Oriented Language (ROOL) [16].

Similarly to BN, it includes most OO language constructs, added to *specification statements* of Morgan's refinement calculus [77]. These laws, proved to be behavior-preserving according to a formal semantics [21], provide a formal basis for defining program refactorings. The following law, for instance, allows introducing and eliminating class declarations, which may change the context of development. Similar to Alloy laws, provisos ensure that the transformations denoted by the law preserve semantics. In order to improve reading, BN language's syntax is used for depicting the laws (cd_1 is the class declaration to be removed).

Law 3. ⟨*class elimination*⟩

$$CT\ cd_1\ =\ CT$$

provided

($\leftrightarrow$) $cd_1 \neq$ `Main`;

($\rightarrow$) $name(cd_1)$ is not used in CT.

($\leftarrow$) (1) cd_1 is a distinct name; (2) Field, method and superclass types in cd_1 are declared in CT.

In addition to equivalence laws, reasoning about classes usually requires a notion of class refinement, for internal representation changes such as addition and removal of private fields. Class refinement is directly related to traditional data refinement [77]. The following law changes private fields in a class, relating them to new fields. The application of this law changes the bodies of the methods in the class. This law is not numbered, as it is proposed for the ROOL language, which presents subtle differences in syntax [16]: c is analogous to the main method, and $\preceq$ symbolizes refinement between the two versions of class A.

Law ⟨*private field-coupling invariant*⟩

$$\boxed{\begin{array}{l}\textbf{class}\ A\ \textbf{extends}\ C\\ \quad \textbf{pri}\ x : T;\ ads\\ \quad mts\\ \textbf{end}\end{array}} \ \preceq_{cds,c,CI}\ \boxed{\begin{array}{l}\textbf{class}\ A\ \textbf{extends}\ C\\ \quad \textbf{pri}\ y : T';\quad ads\\ \quad CI(mts)\\ \textbf{end}\end{array}}$$

The coupling invariant CI relates abstract and concrete attributes. The notation $CI(mts)$ indicates the application of CI to each of the methods in mts, as stated by traditional laws of data refinement [77]: every guard may assume the coupling invariant and every command is extended by modifications to the concrete variables so that the coupling invariant is maintained. Simulation is then established, in which the constructor makes the coupling invariant true, and every method begins executing on a valid state and results in another valid state.

Additionally, the set of laws for ROOL is complete in the sense that it is sufficient to reduce an arbitrary program to a normal form substantially close to an imperative program. Another application of these laws is the formal derivation of refactorings; the laws offer a basis for proving that the transformations preserve behavior and, therefore, are indeed refactorings [16], similarly to object model refactorings with Alloy laws.

These laws have been proved, however, for a *copy semantics*, in which objects are *values*, not referenced by pointers. This decision simplified the semantics to the point in which certain laws – in special laws based on data refinement – require simpler proofs, as *aliasing* can be ruled out, allowing direct modular reasoning. For instance, the ROOL catalog includes the following assignment law, in which $e_2[e_1/le]$ denotes the substitution of le by the free occurrences of e_1 in e_2.

$$(le := e_1;\ le := e_2) = (le := e_2[e_1/le])$$

This law holds only with copy semantics; in general, it is unsound when le is a shared object reference, in particular if le takes the form of $le.a$. Using the file system as example, we assume that two variables – **self** and `anotherFile` refer to the same `File` instance. Also, we include an integer field named `size` to `File`. Then the following sequential statement cannot be subject to the previous law:

```
self.size:=0;
self.size:= anotherFile.getSize() + 2048;
self.size:= self.size + anotherFile.getSize()
```

$\neq$

```
self.size:=0;
self.size:= (anotherFile.getSize() + 2048) + anotherFile.getSize()
```

The outcome for **self**`.size` in the first sequence of statements is 4096, whereas 2048 is the outcome in the transformed statement. This difference takes place due to *aliasing*, in which the first assignment to **self**`.size` produces a side effect to `anotherFile.size`.

Although most object-oriented programming laws considered here do not rely on copy semantics – except the simulation law for applying data refinement – most practical object-oriented programming languages have a reference semantics. Therefore, due to the inconvenience of defining refactoring on a language that is disconnected from the practice of object-oriented programming, we transferred the law catalog to a reference-based language (BN), inspired by the work of Banerjee and Naumann [8], which guarantees modular reasoning by using *confinement* as a requirement for programs.

4.3 Confinement

Modular reasoning has been a core issue during software development and evolution. The ability to develop and understand modules independently of the others is critical to extensibility and reusability. This data abstraction requires from programming languages powerful encapsulation mechanisms, which have been tackled by advances in language design. Nevertheless, encapsulation provided by language constructs often presents a few concerns; for object-oriented languages, they usually fail in the presence of aliasing, as showed in the previous section.

Several disciplines have been proposed for controlling aliasing. The most relevant ones define some form of *ownership confinement* [23], restricting access to designated *representation objects* (reps), except through their *owners*, to avoid representation exposure [8]. An owner is a class maintains representation objects stored in the fields of its objects. We show next a program fragment depicting confinement and representation independence, using as example a small version of classes `FSObject` (owner) and `Name` (rep) from the file system example. `FSObject` declares accessor methods (`setName,getName`), for basic operations on the `name` field. The constructor assigns a new name to the file system object that is being constructed. In the `Name` class, fields represent the two parts of a name (`String` defines a type as in Java).

```
class FSObject{
  Name name;
  unit setName(Name n) { self.name:= n }
  Name getName() { result:= self.name }
  constr { self.name:= new Name; }
}
class Name{
  String label;
  String extension;
  unit setExtension(String ext) { self.extension:= ext }
}
```

Usually, directory names have no extension, differently from files. We define an *object invariant* for `FSObject`, establishing the valid states for an instance of this class ($\mathbf{self}\ \mathbf{is}\ Dir \Rightarrow \mathbf{self}.name.extension = \mathbf{null}$) – an instance of `FSObject` is in a valid state only if its name has no extension, in case it is a directory. Regular information hiding enforces a simple encapsulation rule – `name` is a private attribute. However, the `getName` method leaks the reference to `name`, so the invariant can be broken from a client code, as follows.

```
  FSObject dir:= new Dir, Name n in
    ..
    n:= dir.getName();
    n.setExtension("txt");
```

In this case, a directory is invalid, as the reference to the rep object is shared by its owner and a client program. This issue raises the need for advanced encapsulation mechanisms; ownership confinement institutes, for the owner `FSObject`, the confined access to its rep objects, in this case `Name` instances. The `FSObject` can be simply rewritten for ensuring confinement, as showed in the next program fragment. No direct access is given for the rep object; instead, `FSObject` defines methods for changing the members of a name indirectly, testing whether the instance is a directory.

```
class FSObject{
  Name name;
  unit setName(String n) { self.name.setName(n) }
  unit setExtension(String ext) {
    if (self is Dir) then
      self.name.setExtension(ext)
  }
}
```

Despite this simple example, confinement may be hard to ensure for any given program. Techniques for statically checking confinement in source code have evolved, allowing less restrictive verifications. Banerjee and Naumann [8] present a number of static analysis rules for ensuring a property called by them *safety*, which is shown in their work to imply confinement. The input is a class table and its division into three sets of classes: *Own* and *Rep*, defining the possibly non-disjoint sets of owner and representation classes, respectively, and *Client* with all other classes. The restrictions are weak enough to admit interesting programs. Also, the analysis is modular, as only *Own* and *Rep* code is constrained (with one exception, for **new** commands). They present the following rules, as adopted in this thesis:

- Public methods declared in *Own* or subclasses cannot return *Rep* types; otherwise, references to internal objects might leak to clients.
- Methods inherited or declared by *Own* cannot have parameters of *Rep* types; otherwise, non-owner subclasses might have access to *Rep* instances;
- *Rep* classes cannot inherit any methods from non-*Rep* superclasses; for instance, a method could return **self** to a client, which is highly undesirable;
- For any field access $e.f$, if e is of type *Own*, it cannot access fields of type *Rep*, unless e is **self**; this rule must be checked only for public fields, as it is guaranteed by type safety for private fields;
- For assignments `x:=` **new** `B` in *Client*, B cannot be *Rep* or any of its subclasses; otherwise, these clients would have direct access to *Rep* instances.
- For method calls `x:= e.m(`$\bar{e}$`)`, (1) if e is a *Client* object, and the call is within *Own* or *Rep* (or subclasses), m cannot have *Rep* parameters (otherwise *Rep* instances could leak); also, (2) if the call is within *Own*, `m` is declared in *Own*, and e is **self**, parameters and return may be *Rep* type. The second case in fact weakens the confinement constraints with a condition that can be detected by static analysis.

If public methods in *Own* return *Rep* objects, confinement can be easily broken by any client with access to these objects. Regarding inheritance, parameters in methods that can be inherited by subclasses of *Own* cannot be *Rep*, for the mentioned reason. Clients cannot either instantiate *Rep* objects, and method calls must be controlled for not passing *Rep* parameters to Client objects.

4.4 Proofs for laws

Although the catalog of laws is proved for a language with a copy semantics, most laws are not affected by a change to reference semantics, except laws for change of data representation [16]. Based on this property, we use the same catalog as a basis for this thesis. Nevertheless, we chose five (5) representative command and ROOL catalog laws, and developed a proof for their correctness on the denotational semantics of the BN language described in Section 4.1. In this section, we present the proof for Law *new superclass* [26]; other laws are *class elimination, assumption guard, change attribute type*, and *move redefined method to superclass*. The proofs for these other four laws are presented in Appendix C.

Law 4 establishes that we can replace instantiations of a class B by instantiations of its superclass A, as long as B is an empty class. This change can occur either in the body of A's methods or any method in the class table CT ($CT[exp'/exp]$ is the notation for substitution of free occurrences of exp by exp'). Instantiations of the superclass can be used if expressions of type A are not cast with B and B instances are assigned only to A-typed variables – type errors would be generated otherwise. The opposite application is constrained by a proviso: tests involving A-typed variables with B may not work if A's instances become B instances.

Law 4. ⟨*new superclass*⟩

$$\boxed{\begin{array}{l} \textbf{class}\ A\ \textbf{extends}\ C\ \{ \\ \quad ads \\ \quad mts \\ \} \\ \textbf{class}\ B\ \textbf{extends}\ A\ \{\ \} \\ CT \end{array}} =_{CT} \boxed{\begin{array}{l} \textbf{class}\ A\ \textbf{extends}\ C\ \{ \\ \quad ads \\ \quad mts' \\ \} \\ \textbf{class}\ B\ \textbf{extends}\ A\ \{\ \} \\ CT' \end{array}}$$

where

$CT' = CT[\textbf{new}\ A/\textbf{new}\ B]$

$mts' = mts[\textbf{new}\ A/\textbf{new}\ B]$

provided

($\rightarrow$) (1) B is not used in type casts or tests in CT or ops for expressions of type A; (2) $x := \textbf{new}\ B$ appears only if $type(x) \leq A$.

($\leftarrow$) Variables of type $T \leq A$ are not involved in tests with type B.

In order to prove this law, we first define the equivalence notion for class tables [8]. Class tables can only be equivalent if they are *comparable*. Comparable class tables differ only in the implementation of a set of classes in the *Own* set – the owners in the confinement notion. For this law, the methods' signature is maintained in both versions of *Own*. In the following definition, class tables are compared in terms of the classes that do not change. For simplification, we consider well-formed class tables.

Definition 4.1. *Suppose that Own, Own' are subsets of class tables CT and CT', respectively. CT and CT' are comparable iff:*

1. $CT - Own = CT' - Own'$;
2. $dom\ \ Own = dom\ \ Own'$;
3. *For any method in Own, either mtype(m,Own) and mtype(m,Own') are both undefined or defined and equal.*

A program consists in a class table CT together with a command $\Gamma \vdash S$, which may be for instance the sequence of commands within the `main` method; Γ defines the typing context, a finite mapping from variable and parameter names to data types (**self** $\in$ dom Γ). Instances that are reachable from variables of Γ are considered the *inputs and outputs* of the program. We only consider programs following the set of confinement rules defined in Section 4.3. In BN's denotational semantics, commands denote functions on states. Thus an obvious notion of equivalence is that $[\![\, \Gamma \vdash S \,]\!]$ and $[\![\, \Gamma \vdash S' \,]\!]$ are equal as functions; however, semantic domains differ for owner object states which may have different private fields. Therefore, we use the concept of *Own*-free program heaps (program states that do not contain instances of owner classes, as explained in detail in Chapter 5), and compare command meanings on the *Own*-free heaps only.

In order to ensure *Own*-free heaps after command execution, the *collect* function, as defined next, yields the input heap from which objects from classes in *Own* are deleted, if unreachable in the heap – this function can be compared to the operation of a garbage collector.

Definition 4.2. *For a list $\bar{d}$ of values, the heap $gc(\bar{d}, h)$ is the restriction of h to values reachable from $\bar{d}$. For $(h, \eta) \in [\![\, Heap \otimes \Gamma \,]\!]$, $collect(h, \eta) = (gc(rng\ \eta, h), \eta)$, where $rng\ \eta$ is the set of values mapped in the store η.*

The assumption is that the initial store η_0, mapping local variables to their current values, never defines a variable that will reach an object from *Own*. Next, we establish the theorem for proving Law 4. In this theorem, both sides of the law define equivalent programs, as they result in the same *Own*-free heaps after the execution of the S command in the `main` method. Taking two heaps, one from each compared program, they only differ in the reference (ℓ) types – from B to A.

Theorem 4.1. *Let CT and CT' be two class tables, representing, respectively, the left-hand and right-hand sides of Law 4, with $Own = \{A, B\}$. Then, for any Own-free heap h_0 and store η_0 in CT, and any command S in the main method:*

$$\begin{aligned}&\forall \mu : [\![\, CT \,]\!], \mu' : [\![\, CT' \,]\!] \bullet \\ &collect([\![\, \Gamma \vdash S \,]\!]\, \mu(h_0, \eta_0)) = collect([\![\, \Gamma \vdash S' \,]\!]\, \mu(h_0', \eta_0'))\end{aligned}$$

where

$$S' = S[\mathbf{new}\ A/\mathbf{new}\ B]$$

$$h_0' = h_0[\ell \in locsA/\ell \in locsB]$$

$$\eta_0' = \eta_0[\ell \in locsA/\ell \in locsB]$$

The proof for this theorem expands semantic definitions from the language's denotational semantics described in Section 4.1.

Proof 4.1. The proof is developed by *induction* over the language commands. Here, we consider two cases: **new** B commands and method calls on B objects. The proof method starts from the program's definition from the left-hand side of the law, and the proof is concluded when the program's definition from the right-hand side of the law is deduced.

Case $x :=$ **new** B.

$collect([\![\, \Gamma \vdash x := \textbf{new } B \,]\!] \, \mu(h_0, \eta_0))$

$=$ [by semantics of **new**]
$$\begin{array}{l} \textit{let } \ell = \textit{fresh}(B, h_0) \textit{ in} \\ \quad \textit{let } h = [h_0 \mid \ell \mapsto [\textit{fieldsB} \mapsto \textit{default}]] \textit{ in} \\ \quad (h, \eta_0[x \mapsto \ell]) \end{array}$$

As in the law B is an empty class, A and B share the same fields (*fieldsB* denotes all fields from class B, either declared or inherited).

$=$ [by hypothesis, $\textit{fieldsB} = \textit{fieldsA}$]
$$\begin{array}{l} \textit{let } \ell = \textit{fresh}(B, h_0) \textit{ in} \\ \quad \textit{let } h = [h_0 \mid \ell \mapsto [\textit{fieldsA} \mapsto \textit{default}]] \textit{ in} \\ \quad (h, \eta_0[x \mapsto \ell]) \end{array}$$

The heaps before and after the command only differ in locations of type B. Since $B \in \mathit{Own}$, the heaps and stores can be considered as equal.

$=$ [h_0 is Own-free, so $h_0 = h_0'$]
$$\begin{array}{l} \textit{let } \ell = \textit{fresh}(B, h_0') \textit{ in} \\ \quad \textit{let } h = [h_0' \mid \ell \mapsto [\textit{fieldsA} \mapsto \textit{default}]] \textit{ in} \\ \quad (h, \eta_0'[x \mapsto \ell]) \end{array}$$

Because instances of A and B are structurally equivalent, and these objects get collected, we can replace $\textit{fresh}(B, h_0')$ by $\textit{fresh}(A, h_0')$.

$=$ [$\textit{fresh}(B, h_0') = \textit{fresh}(A, h_0')$]
$$\begin{array}{l} \textit{let } \ell = \textit{fresh}(A, h_0') \textit{ in} \\ \quad \textit{let } h = [h_0' \mid \ell \mapsto [\textit{fieldsA} \mapsto \textit{default}]] \textit{ in} \\ \quad (h, \eta_0'[x \mapsto \ell]) \end{array}$$

$=$ [by semantics of **new**]
$collect([\![\, \Gamma \vdash x := \textbf{new } A \,]\!] \, \mu'(h_0', \eta_0'))$

Case $x := e.m(\bar{e}), e$ **is** B.

$collect([\![\, \Gamma \vdash x := e.m(\bar{e}) \,]\!] \, \mu(h_0, \eta_0))$

The semantic definition of method call first considers the case in which the target expression e is **null**; the program must abort, which is represented by $\bot$. Else, a binding for the parameter names ($\textit{pars}(m, B)$) to the values of expressions $\bar{e}$ is added to store η. The resulting heap and store (h, d_1), given by the semantics of the method call $\mu(B)m(h_0, \eta)$ are then added to the binding of the **result** variable.

= [by semantics of method call]

```
  let $\ell = [\![ \Gamma \vdash e : B ]\!] (h_0, \eta_0)$ in
    if $\ell = nil$ then $\perp$ else
    let $\bar{x} = pars(m, B)$ in
    let $\bar{d} = [\![ \Gamma \vdash \bar{e} : \bar{U} ]\!] (h_0, \eta_0)$ in
    let $\eta = [\bar{x} \mapsto \bar{d}, self \mapsto \ell]$ in
    let $(h, d_1) = \mu(B)m(h_0, \eta)$ in
    $([h, [\eta \mid x \mapsto d_1])$
```

By the law template, B does not declare methods, so the semantics of any method in both classes is the same.

= $[\mu(B)m = \mu'(A)m]$

```
  let $\ell = [\![ \Gamma \vdash e : B ]\!] (h_0, \eta_0)$ in
    if $\ell = nil$ then $\perp$ else
    let $\bar{x} = pars(m, A)$ in
    let $\bar{d} = [\![ \Gamma \vdash \bar{e} : \bar{U} ]\!] (h_0, \eta_0)$ in
    let $\eta = [\bar{x} \mapsto \bar{d}, self \mapsto \ell]$ in
    let $(h, d_1) = \mu'(A)m(h_0, \eta)$ in
    $([h, [\eta \mid x \mapsto d_1])$
```

Also from the law template, both classes have the same fields (*fieldsA* = *fieldsB*), so objects in both heaps map to the same values.

= $[[\![\Gamma \vdash e : B]\!] (h_0, \eta_0) = [\![\Gamma \vdash e : A]\!] (h_0', \eta_0')]$

```
  let $\ell = [\![ \Gamma \vdash e : A ]\!] (h_0', \eta_0')$ in
    if $\ell = nil$ then $\perp$ else
    let $\bar{x} = pars(m, A)$ in
    let $\bar{d} = [\![ \Gamma \vdash \bar{e} : \bar{U} ]\!] (h_0', \eta_0')$ in
    let $\eta = [\bar{x} \mapsto \bar{d}, self \mapsto \ell]$ in
    let $(h, d_1) = \mu'(A)m(h_0', \eta)$ in
    $([h, [\eta \mid x \mapsto d_1])$
```

= [by semantics of method call]

$collect([\![\Gamma \vdash x := e.m(\bar{e})]\!] \mu'(h_0', \eta_0'))$

□

Regarding ROOL laws that do not work with reference semantics, we extensively use *class refinement* in this thesis. In this law, the internal representation of a class is modified, for instance with the addition or removal of private attributes. Although method bodies change, its signature must be maintained, as these changes must be transparent to the client of this class. In order to ensure the correctness of the refinement, a coupling invariant [77] between the previous and new representation must be established and ensured by the methods of the class. As pointed out in Section 4.2, this invariant might not be guaranteed in programs with reference sharing.

Nevertheless, work on a theory for refinement for programs with confinement [8] provides a proof for the following proposition: if two class tables CT and CT' are confined and the coupling relation establishes a *simulation*, then the refinement is correct ($CT \sqsubseteq CT'$). The main corollary of this theorem is that laws related to class refinement can be proved by simulation, and we use this corollary in our thesis for class refinement in a language with references. The simulation property defines that a coupling invariant between two classes:

- holds initially (execution of constructors);
- holds in the outcome of two versions of the same method, if these versions are executed for initial states related by the coupling invariant.

We apply this theorem in proving correctness of transformations that include class refinement, as showed in Chapter 7.

4.5 Laws and refactorings enunciated for this work

The BN laws and refactoring rules presented in this section were stated during our work; as laws of programming form the basis for our model-driven approach to program refactoring, a number of program equivalences are used, hereafter identified as laws. Refactorings can be derived from other laws; in these cases we show the derivation steps. Nevertheless, Law 5 for distributing type tests, presented in this section, is likely to be not derived (a primitive law). For that, we developed a proof in the semantics of the BN programming language.

The following refactoring rule defines a way to eliminate references to a specific abstract class within cast expressions. A command with a cast expression for class B can be replaced by a sequence of **if** statements (denoted by $\mathbf{if}_i, i = 1..n$), whose conditions correspond to each of B's subclasses. The command for each **if** is rewritten with the subclass in the corresponding condition. During the proof, a special command, *assumption*, is used. The command is given by {`boolexp`} `S`, being a syntactic sugar for **if** `(boolexp)` **then** `S` **else** $\perp$.

Refactoring 2. ⟨*distribute casts*⟩
Considering the following class declarations, where $C_i, i = 1..n$ encompasses all subclasses of B in CT, and that no object values reachable from **self**`.inout` have exact type B:

class B **extends** A {
  fds;
  mts
}
class C_i **extends** B {
  fds_i;
  mts_i
}

then, for any command *cmd* in CT:
$cmd[(B)exp] = \textbf{if}_i\ (exp\ \textbf{is}\ C_i)\ \textbf{then}\ cmd[(C_i)exp]$

provided

$(\leftrightarrow)$ $x := \textbf{new}\ B$ does not appear in CT, *mts* or mts_i

Proof 4.2. This refactoring can be derived from other laws of programming and predicate logic inference rules, as detailed in the following steps. The only command with a cast expression considered in the derivation is assignment ($x := (B)exp$), since the casts in other commands can be reduced to the assignment, by introducing a local variable block.

$x := (B)exp$

= [Law 39(L-R) *eliminate cast of expressions*]
  $\{exp\ \textbf{is}\ B\}x := exp$

After replacing the cast expression with an assumption, the assumption formula is replaced with a predicate stating one of the properties in inheritance when B is abstract (if an object is an instance, it is also an instance of any of the subclasses).

= [Since B is abstract, property of inheritance ($exp\ \textbf{is}\ B \Rightarrow \bigvee_i(exp\ \textbf{is}\ C_i)$)]
  $\{\bigvee_i(exp\ \textbf{is}\ C_i)\}x := exp$

With a law application, we can introduce **if** statements for each of the conditions in the disjunction within the assumption. In the outcome, the assumption can be absorbed by all the conditions in those statements, as the disjunction is valid, then trivially eliminated by predicate logic.

= [Law 25(R-L) *if identical guarded commands*]
  $\{\bigvee_i(exp\ \textbf{is}\ C_i)\}\textbf{if}_i(exp\ \textbf{is}\ C_i)\ \textbf{then}\ x := exp$

= [Law 21(L-R) *absorb assumption*]
$\textbf{if}_i((exp \textbf{ is } C_i) \wedge (exp \textbf{ is } C_i)) \textbf{ then } x := exp$

= [by predicate logic]
$\textbf{if}_i(exp \textbf{ is } C_i) \textbf{ then } x := exp$

Assumptions now are introduced into each **then** clause with a primitive law. The newly-introduce assumptions can be replaced by casts for each of the subclasses, which finishes the derivation. Since all the steps are equalities, they can be generalized for the refactoring's reverse application.

= [Law 20(L-R) *assumption guard*]
$\textbf{if}_i(exp \textbf{ is } C_i) \textbf{ then } \{exp \textbf{ is } C_i\}x := exp$

= [Law 39(R-L) *eliminate cast of expressions*]
$\textbf{if}_i(exp \textbf{ is } C_i) \textbf{ then } x := (C_i)exp$

□

The next law is similar, but for replacing type tests. Each type test with B is replaced by an alternation of tests, one for each subclass. Again the only proviso states that B must be an abstract class.

Law 5. ⟨*distribute type tests*⟩
Considering the following class declarations, where $C_i, i = 1..n$ encompasses all subclasses of B in CT, and that no object values reachable from **self**.`inout` have exact type B:

class B **extends** A {
  fds;
  mts
}
class C_i **extends** B {
  fds_i;
  mts_i
}

then, for any command *cmd* in CT:
$cmd[exp \textbf{ is } B] = cmd[\bigvee_i(exp \textbf{ is } C_i)]$

provided

($\leftrightarrow$) $x := \textbf{new } B$ does not appear in CT, *mts* or mts_i

We first establish the theorem for this specific law, over the type test expression.

Theorem 4.2. *For any heap h_0 and store η_0:*

$$\begin{array}{l} \forall\, \mu : [\![\, CT \,]\!] \bullet \\ [\![\, \Gamma \vdash \mathit{exp}\ \textbf{is}\ B : \textbf{bool} \,]\!]\ (h_0, \eta_0) = \\ [\![\, \Gamma \vdash \bigvee_i (\mathit{exp}\ \textbf{is}\ C_i) : \textbf{bool} \,]\!]\ (h_0, \eta_0) \end{array}$$

Proof 4.3. Direct proof. The expression *loctype* ℓ yields the exact type of object reference ℓ.

$[\![\, \Gamma \vdash \mathit{exp}\ \textbf{is}\ B : \textbf{bool} \,]\!]\ \mu(h_0, \eta_0)$

$=$ [by semantics of type test]
$$\begin{array}{l} \mathit{let}\ \ell = [\![\, \Gamma \vdash \mathit{exp} : D \,]\!]\ (h_0, \eta_0)\ \mathit{in} \\ \quad \mathit{if}\ \ell \neq\ \mathit{nil} \wedge \mathit{loctype} \ell \leq B\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{false} \end{array}$$

$=$ [by property of inheritance, and B is abstract, $\mathit{loctype}\ \ell \leq B \Rightarrow \bigvee_i (\mathit{loctype}\ \ell \leq C_i)$]
$$\begin{array}{l} \mathit{let}\ \ell = [\![\, \Gamma \vdash \mathit{exp} : D \,]\!]\ (h_0, \eta_0)\ \mathit{in} \\ \quad \mathit{if}\ \ell \neq\ \mathit{nil} \wedge \bigvee_i (\mathit{loctype}\ \ell \leq C_i)\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \mathit{false} \end{array}$$

$=$ [by semantics of type test]
$$[\![\, \Gamma \vdash \bigvee_i (\mathit{exp}\ \textbf{is}\ C_i) : \textbf{bool} \,]\!]\ \mu(h_0, \eta_0)$$

$\square$

The next law states that an **if** command with the same body in both **then** and **else** parts may be trivially eliminated. This law is only an adaptation for the BN syntax of a similar law in ROOL for eliminating guards [26].

Law 6. ⟨*eliminate redundant if*⟩
if ψ **then** *cmd* **else** *cmd* $=$ *cmd*

Finally, we define a refactoring by specializing Law 34 *method elimination* with a specific constraint: the superclass subject to change is abstract. In this law, a method can be eliminated even if it is being called throughout the program. In this case, the class must be abstract and all subclasses must redefine the method, thus calls to this method surely refer to the redefined version.

Refactoring 3. ⟨*method elimination-abstract class*⟩

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad fds\\ \quad T\ m(\bar{T}\ \bar{x})\ \{\ cmd\ \}\\ \quad mts\\ \}\end{array}} =_{CT} \boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad fds\\ \quad mts\\ \}\\ \end{array}}$$

provided

(↔) (1) $x := \textbf{new}\ C$ does not appear in CT, mts, mts' or cmd; (2) m is redefined in all subclasses of C in CT;

4.6 Chapter summary

In contrast to the modeling language, the chosen programming language demanded contributions from this thesis, as presented in this chapter. Mainly, this thesis' approach was applied to a reference-based programming language. In order to use laws of programming as the basis for program refactoring, we reused part of the law catalog for the ROOL language [16]; as this catalog was defined for copy semantics, we used ownership confinement as a guarantee that laws are correctly applied. We developed proofs for laws in the semantics of the reference-based language (BN), and class refinement is dealt with by directly applying a theorem from Banerjee and Naumann's work [8]: if a simulation is proved for a given confined owner class, the class refinement is entailed. In addition, several laws and refactorings have been enunciated during their application in this thesis.

Chapter 5

Object Model and Program Conformance

A desirable property of object models is abstraction; ideally, they can be implemented by several structurally-different programs as long as the invariants hold during their executions. These programs are then in *conformance* with the object model. In order to precisely define the level of abstraction for models that are subject to program conformance, we must establish a specific *conformance relationship*. Conformance, whose motivation is described in Section 5.1, is given both in syntactic and semantic terms, as shown in Sections 5.2 and 5.3, respectively. Also, we define a formal framework (Section 5.4) for defining conformance relationships, which evolved from our specific notion. Conformance relationships can be applied in several contexts. For instance, generation of source code from models, reverse engineering and evolution activities in model-driven methodologies, in which software is completely generated and evolved by manipulating models – the case for this thesis.

5.1 Motivation

For our purposes, conformance consists in design decisions being fulfilled by all executions of a program. Regarding object models, we consider that a program is in conformance with a model when it meets all of the specified invariants throughout every execution. In order to investigate how this conformance can be maintained throughout refactoring tasks, a formal definition of conformance is needed.

Structures in the model must be somehow implemented in the program, offering a basis for evaluating whether the modeled constraints are met. In the file system example, its implementation must offer a number of declarations representing directories and files; the constraints over these concepts may then be analyzed, or even machine-checked. We call this correspondence *syntactic conformance*. Given a syntactic conformance relationship, fulfilment of model invariants by the executions of a given program is regarded as *semantic conformance*. An object model may be implemented by a virtually infinite number of programs, around two axes: distinct statements, and different syntactic conformance relationships, as depicted in Figure 5.1. First, programs following the same modeled structure may have completely diverse method implementations. Second, several programs with different implemented structures may be conforming with a given object model. More on distinct conformance relationships can be seen in Section 5.4.

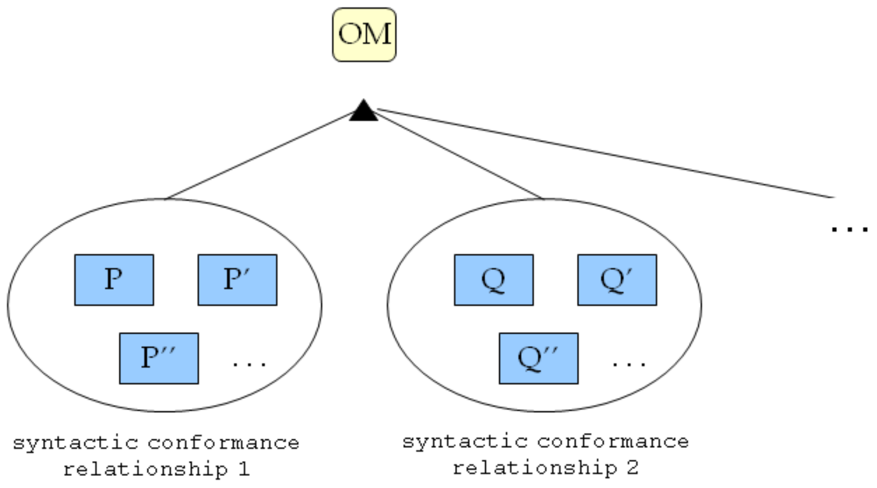

OM
P
P′
P′′
. . .
Q
Q′
Q′′
. . .
. . .
syntactic conformance relationship 1
syntactic conformance relationship 2

Figure 5.1: Possible conforming implementations for an object model

5.2 Syntactic Conformance

In an object model in Alloy, signatures and relations constitute the main elements, along with invariants over these elements. These elements are given concrete representations in the program, in terms of object-oriented programming language constructs (classes, fields, inheritance, etc.). In this section, we detail the formal definitions of syntactic conformance – and later for semantic conformance – using the PVS (Prototype Verification System) specification language [83], on which these definitions were type-checked. In fact, besides mixing some well-known mathematical symbols with PVS keywords and functions, we consider a few extensions to the original PVS language for improving readability.

A formal definition for models and programs is given, using uninterpreted types (`TYPE`), in the following specification fragment – subtypes can be defined with the `FROM` keyword. Names are unique, and for signatures and classes, they are taken from the same set of names (same for relations and fields). Relation multiplicities may be defined as single-valued – `ONE_LONE` – or collection-valued – `SOME_SET`; field multiplicities are analogously defined as `SINGLE` or `SET`.

```
Model, Program: TYPE
Name: TYPE
SigClassName: TYPE FROM Name
RelFieldName: TYPE FROM Name

ONE_LONE, SOME_SET, SINGLE, SET: TYPE
RelMulti: TYPE = { ONE_LONE,SOME_SET }
FieldMulti: TYPE = { SINGLE, SET }
```

The signature structure – defined as a record (`[# #]`) – is composed of a single name and a list of parent signatures, encompassing all signatures from the immediate supersignature to `object`, a special signature also in Alloy – Alloy allows only single inheritance, so the `extends` is always a linear list (the first element of this list is the immediate supersignature). Relations (and fields) are structured as a name, a multiplicity value and two types, left and right – we consider only binary relations, even though Alloy permits n-ary relations – this choice is motivated not only for simplification, but also due to the requirements of our solution, as common object-oriented programs are able to implement only binary relations mostly. Although relations and fields are declared within signatures and classes, they are separately specified. Furthermore, the function declarations `sigs, relations, classes, fields` define how to recover all elements from a model or a program; analogously, functions for gathering the names contained in models and programs are declared (for instance, `sigNames`).

```
Sig: TYPE =
[# name: SigClassName,
   extends: list[SigClassName]
#]
Relation: TYPE =
[# name: RelFieldName,
   multiplicity: RelMulti,
   leftType: SigClassName,
   rightType: SigClassName
#]
Class: TYPE =
[# name: SigClassName,
   extends: list[SigClassName]
#]
Field: TYPE =
[# name: RelFieldName,
   multiplicity: FieldMulti,
   leftType: SigClassName,
   rightType: SigClassName
#]

sigs: Model → ℙ Sig
sigNames: Model → ℙ SigClassName
relations: Model → ℙ Relation
relNames: Model → ℙ RelFieldName
classes: Program → ℙ Class
classNames: Model → ℙ SigClassName
fields: Program → ℙ Field
fieldNames: Model → ℙ RelFieldName
```

Developers intended to reflect model refactoring to source code might also expect some syntactic conformance, otherwise the approach would not be applicable. In model-driven refactoring, we used a specific conformance relationship, as shown in the next specification fragment. The syntactic mapping for signatures defines one direct class for each signature declared in the model (for simplicity, this specification rely on the equality between names, although a mapping between names could be easily established as well). Also, all supersignatures of a signature are included in the list of superclasses of the corresponding class, indicating that more superclasses may be declared in the program, but the hierarchy is maintained. in this case we used PVS function `list2set` for converting lists into sets and comparing set containment.

```
sigMapping(s:Sig,p:Program): boolean =
  ∃ c:classes(p) • c.name=s.name ∧ list2set(s.extends) ⊆ list2set(c.extends)
```

Likewise, every relation is mapped to one field, with one additional constraint: relations with single multiplicity (yielding a scalar value) are mapped to single field, whereas relations with set multiplicity must be mapped to collection-type fields. First, we define predicates `isScalar` for relation and field multiplicities.

```
isScalarR(r:Relation): boolean =
  r.multiplicity = ONE_LONE
isScalarF(f:Field): boolean =
  f.multiplicity = SINGLE

relationMapping(r:Relation,p:Program): boolean =
  ∃ f:fields(p) • r.name=f.name ∧
    r.leftType=f.leftType ∧ r.rightType=f.rightType ∧
    (isScalarR(r) ⇒ isScalarF(f)) ∧
    (¬ isScalarR(r) ⇒ ¬ isScalarF(f))
```

The syntactic conformance then reduces to checking the signature mapping for all signatures and the relation mapping for all relations declared in the Alloy model. An additional constraint establishes that any of the non-modeled intermediate classes in a hierarchy between classes whose corresponding signatures are modeled must be abstract, for which instances are not created (this constraint is due to a specific aspect of our approach, as detailed in Section 6.6. The used `abstract(c:Class,p:Program)` predicate tests whether class `C` is abstract in program `P`, which is true if the class is never instantiated in the program.

```
abstractConstraint(m:Model, p:Program): boolean =
  ∀c:classes(p) •
     c.name ∉ sigNames(m) ∧ ∃c1,c2:classes(p) •
       c1.name ∈ sigNames(m) ∧ c2.name ∈ sigNames(m) ∧
       c ∈ list2set(c1.extends) ∧ c2 ∈ list2set(c.extends) ⇒ abstract(c,p)

syntConformance(m:Model, p:Program): boolean =
  ∀ s:sigs(m) •  sigMapping(s,p) ∧
  ∀ r:relations(m) • relationMapping(r,p) ∧
  abstractConstraint(m,p)
```

5.3 Semantic Conformance

For a conforming program, model invariants must be preserved throughout the program's execution. If this is confirmed, the program is in semantic conformance with the model; this concept is similar to data refinement. Since object models constrain the valid states of a program, the focus is on the set of possible states of a program that may be the result of an execution step – we ignore the transition between those states, which for instance are tackled by behavioral modeling using UML state diagrams. First, a semantic definition for both languages is provided, succeeded by a specific topic in the program semantics, namely heaps of interest. Finally, conformance itself is described based on the previous specifications.

5.3.1 Semantics

An Alloy model defines valid states for a given system. An Alloy state is an *interpretation* [42], which contains mappings of signatures and relation names to sets of *object values*, as declared next. Object values may be single objects for sets and pairs of objects for relations. The semantic definitions for Alloy that is presented in this thesis is a simplification of the semantics defined by Gheyi [42], being sufficient for the problem at hand.

```
Value: TYPE
objValue: TYPE FROM Value
objPairValue: TYPE FROM Value =
[# t1: objValue,
   t2: objValue
#]

Interpretation: TYPE =
[# mapSig: SigClassName → ℙ objValue,
   mapRel: RelFieldName → ℙ objPairValue
#]
```

We consider the semantics of an object model in Alloy as the set of all valid interpretations satisfying all modeled invariants. Each of these interpretations consists in all valid assignments of values to global constants – the signatures and the relation names. A valid assignment satisfies all modeled invariants – implicit or explicit [42], which is indicated in the following definition. Invariants are implicit when they constrain the model but are not declared in facts; we consider only the implicit invariants from **extends**, in which (1) instances of a subsignature must form a subset of the supersignature's set of instances and (2) values for subsignatures are disjoint. For explicit invariants, a valid interpretation must satisfy the formulas contained in the model (given by the function `factInvs(m)`). The `satisfyFormula(f,i)` predicate assigns values from `i` to the formula `f` – we omit the definition here, as it is not used in our proof, being fully presented in Gheyi's thesis [42]. As an example, Figures 3.1 and 3.2 (pages 22) and 23) depict for the file system example two valid interpretations, which are part of the semantics of the corresponding model. For this definition, we only consider well-formed Alloy models.

```
satisfyImpInvs(m:Model,i:Interpretation): boolean =
  ∀ s:sigs(m) •
    i.mapSig(s.name) ⊆ i.mapSig(super(s).name) ∧
  ∀ s1,s2:sigs(m) • super(s1) = super(s2) ⇒
    i.mapSig(s1.name) ∩ i.mapSig(s2.name) = ∅

satisfyExpInvs(m:Model,i:Interpretation): boolean =
  ∀ f:factInvs(m) • satisfyFormula(f,i)

semantics(m:Model): ℙ Interpretation =
  {i: Interpretation | satisfyImpInvs(m,i) ∧ satisfyExpInvs(m,i) }
```

Regarding object-oriented programs, states are formalized as *heaps* of object values, defined in the following specification as a record mapping class names to sets of objects and field names to pairs of object values (references). Since we deal with a programming language with references, the semantic definition is very close to Alloy. If an object in a heap contains a field storing a null value, no pair of values exists with that object as the first member.

```
Heap: TYPE =
  [# mapClass: SigClassName → ℙ objValue,
     mapField: RelFieldName → ℙ objPairValue
  #]
```

The semantics of a program is given by *the set of sequences of heaps* resulting from all possible execution traces – depending on the possible program inputs. Again, the complete definition of the Alloy semantics is presented in Gheyi's thesis [42].

```
semantics(p:Program): ℙ(seq Heap)
```

5.3.2 Heaps of interest

Our aim is to define how each *relevant heap* preserves the model invariant – we regard a relevant heap as a stable program state that is important for conformance checking. Even though the semantics of an object-oriented program encompasses a set of heap sequences resulting from all possible execution traces, for the purpose of verifying semantic conformance with object models, we consider the *set of heaps from those sequences*. It would be straightforward to consider a filter yielding all possible heaps from execution traces; however, this approach does not truly reflect the real intentions of conformance checking in most scenarios, since some of the heaps may be acceptably invalid at some well-defined points of the program.

In order to illustrate the problem, consider the class `Dir` from the file system example, extended with a method for moving its files and directories to another directory. In the following method, the content of a directory is moved to `d` (line 3), before cleaning the current directory (line 4).

```
1 class Dir{ ..
2   unit transferFilesTo(Dir d){
3     d.setContents(self.contents);
4     self.contents:= ∅
5   }
6 }
```

Assuming an invariant from the model stating that two `Dir` instances cannot have overlapping content, it is guaranteed before and after calls to the indicated method; however, this is not true right after executing line 3 and before line 4 is executed. At this moment, they have the same contents, breaking the invariant. Nevertheless, in practice, this program is conforming to the invariant, since it is natural that methods perform encapsulated state changes which are not perceived by clients. For that, we need to restrict on which points of the program code the clients may rely on model invariants, for example before entry and after execution of the `transferFilesTo` method. *Heaps of interest* are then the program heaps that are valid for semantic conformance.

A suitable solution for making those program portions explicit is provided by Barnett et al. [11], who present a specification methodology for enriching the program with constructs that indicate code on which invariants may be invalid. In their approach, every object is added a special public field, named `st` (for "state"), of type `{Invalid,Valid}`; if `obj.st=Valid`, the object `obj` is considered valid, which means that the invariants over its state should hold. Otherwise, this is not guaranteed. As a result, conformance checking is performed only when all objects are valid.

In source code, the value of `st` can only be modified through the use of two special statements, **unpack** and **pack** [11]. The command **unpack** `obj` changes `obj.st` to `Invalid`, opening a portion of code that is not considered for conformance – this portion is finalized with the **pack** `obj` command. These commands are exemplified in a new version of the `transferFilesTo` method, in the following program fragment.

```
class Dir{ ..
  unit transferFilesTo(Dir d){
    unpack self;
      d.setContents(self.contents);
      self.contents:= ∅;
    pack self;
  }
}
```

These two commands can be seen as object transaction delimiters. Object transactions include copy or removal of references, value changes and other operations for consolidating major state changes. The invariants are known to hold before or after those transactions. Also, their application may be straightforward, since they can be subject to automatic generation.

We now extend our formal definitions in the light of the presented solution, for defining a filter for heaps of interest. We first extend the definition of heaps including the `invalid` field, which indicates the truth for class names whose any object presents an invalid status, according to the `st` field.

```
Heap: TYPE =
  [# mapClass: SigClassName → ℙ objValue,
     mapField: RelFieldName → ℙ objPairValue,
     invalid: SigClassName → boolean
  #]
```

Now we can define a predicate that indicates invalid heaps for a set of class names. A heap is invalid for a set of class names if, and only if, it is invalid for any of the names in this set. The predicate is then used for determining heaps of interest from the semantics of the program, as denoted by the following PVS fragment by the `filter` function. In addition, function `toSet` takes a set of sequences of heaps and a set of class names, resulting in the set of valid heaps from those sequences, ignoring equivalent heaps (using the PVS function `seq2set` [83]). The `filter` function may then be used for gathering the set of heaps of interest for a program, as given by the `heaps` function.

```
invalidHeap(h:Heap, cNames: ℙ SigClassName): boolean =
  ∃ n:cNames • h.invalid(n)

filter(h:Heap, p:Program): boolean =
  ¬invalidHeap(h,classNames(p))

toSet(ss:ℙ(seq Heap)): ℙ Heap =
  { y:Heap | ∃ t:ss • y ∈ seq2set(t) }

heaps(p:Program): ℙ Heap =
  { h: Heap | h ∈ toSet(semantics(p)) ∧ filter(h,p)}
```

We can now establish the conditions on which programs are in semantic conformance with an object model.

5.3.3 Conformance between interpretations and heaps

A program is in semantic conformance with a model if, and only if, it is in syntactic conformance, and, for every filtered heap from its execution, there is a corresponding interpretation from the semantics of the model.

```
semanticConformance(m:Model, p:Program): boolean =
  syntConformance(m,p) ∧ heaps(p)≠ ∅ ∧
  ∀ h: heaps(p) • ∃ i: semantics(m) •
    ∀ s: sigs(m) • i.mapSig(s.name) = h.mapClass(s.name) ∧
    ∀ r: relations(m) • i.mapRel(r.name) = h.mapField(r.name)
```

If semantic conformance is fulfilled, we say that the Alloy invariants hold during executions of the program. The established relationship between heaps and interpretations in semantic conformance is depicted in Figure 5.2. From the sequences of heaps in the program's semantics, the filter yields a set of heaps of interest. The semantic conformance is defined for these heaps having a counterpart in the model's semantics, only considering the modeled names. If a program presents an empty `heaps(p)` (`main` method could be empty, for instance), it is not considered for semantic conformance.

5.4 General Framework for Conformance Relationships

The conformance relationship presented in this chapter enforces a few constraints over the syntax of the program, which is required for establishing the semantic conformance predicate in the specification. In this case, the syntactic constraints are defined for a

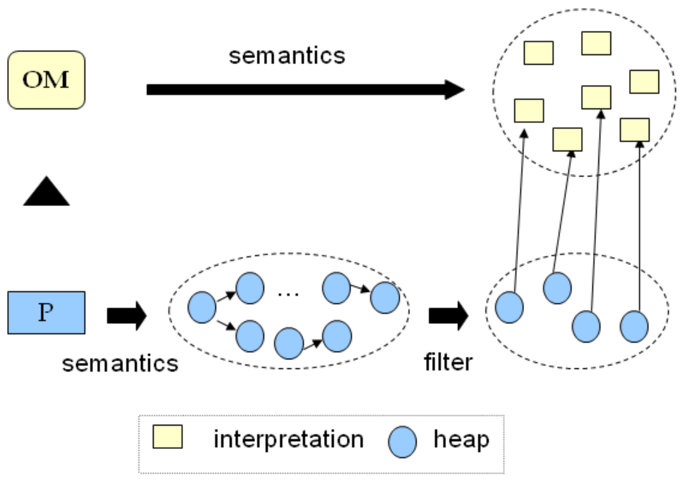


Figure 5.2: Semantic conformance

specific application – model-driven refactoring. However, object models may be implemented in different ways, due to abstraction. In this context, an additional contribution of this thesis is a formal framework for defining conformance relationships between object models and object-oriented programs, allowing reasoning with more flexible rules of syntactic conformance. It supports independence of model and program semantics, by relying upon intermediate representations of interpretations and heaps, which are related by a *mapping formula* that involves model and program syntactic constructs, defining how a model element is implemented in the program.

This syntactic mapping is the framework's *hot spot*, as provided by the user, offering means for semantic conformance between models and programs, independently from object modeling or programming languages. The chosen representation for model interpretations is language independent, indicating how the semantics of the object model is defined; mapping from names to values can describe interpretations of object models written in Alloy or even UML class diagrams. In fact, any modeling language whose semantics can be defined in terms of interpretations is applicable.

Usually, a *traditional syntactic conformance* consists in modeled sets or relations being mapped to a single class or field, respectively. However, several possible conformance relationships may be useful in practice. Using BN fragments, we show several implementations of the object model for the file system example using different syntactic conformance relationships.

Syntactic conformance 1. Using the traditional syntactic conformance, signatures are directly implemented as single classes. The fields implement the modeled relations – `contents` as a collection of file system objects, and `name` as a single variable. `Dir` is implemented as a subclass of `FSObject`. The program declares several methods as needed by the implementation.

```
class FSObject {
  set FSObject contents;
  Name name;
  ..
}
```

```
class Dir extends FSObject {..}
```

The traditional conformance relationship favors the manipulation of those constructs by CASE tools, especially for code generation and reverse engineering. For instance, a skeleton declaration for the `FSObject` can be easily generated from the object model, including the `contents` field; on the other hand, the object model sets and relations can also be generated from the source code with minimal user intervention. Similarly, semantic conformance with direct correspondence is usually easier to check. Although useful and often applied, the traditional notion is too restrictive. Not so direct implementations are also seen in practice, as shown below.

Syntactic conformance 2. In another conformance relationship, the fields are implemented as object *collections*, exemplified by the `List` of file system objects as showed in the next program fragment. The method `add` includes new elements in the list. The program declares several methods as needed by the implementation.

```
class FSObject {
  List contents;
  Name name;
  constr { self.contents:= new List }
  unit addContent (FSObject obj) { self.contents.add(obj) }
  ..
}
```

Another example, specific for signature conformance, is related to the subset relationship from the model, which may not be implemented with inheritance in the program.

Syntactic conformance 3. Due to abstraction, another conformance relationship allows implementations to be even more distinct from the object model. Here the supertype relationship from the object model is implemented as a field in `FSObject`, defining the type of each object (file or directory).

```
class FSObject {
  int type;
  ..
  unit setType(int t) { self.type:= t }
  ..
  /* added only when type = 1 (directory) */
  unit addContent (FSObject obj) { if (self.type = 1) then .. else .. }
}
```

Syntactic conformance relationships 2 and 3 limit the application of CASE tools, since they require more intricate conformance relationships that most tools do not support. For instance, a complex algorithm is required to detect inheritance in the implementation of `FSObject`; similar conclusions are drawn in other approaches dealing with relations [53, 55]. Alternatively, a tool could allow users to define custom mappings for syntactic conformance. Well-known tools, such as Rational Software Architect [91] and Poseidon [40] still offer the traditional notion for code generation and reverse engineering. This scenario usually results in rather concrete reverse-engineered models, cluttered with implementation details.

Regarding semantic conformance, it is much harder to verify that model invariants hold in program heaps, when the program presents such disparate constructs for implemented model elements. The user-defined syntactic mapping between model and program must be applied in this verification, in order to correctly relate values from both worlds and check conformance.

5.4.1 Syntactic Mapping

In order to establish conformance, a correspondence between model and program declarations is established. For instance, the traditional syntactic conformance takes classes for representing signatures and fields for relations. If we define these rules in terms of formulas, exemplars could be as follows, where m and p decorations indicate names from the model or program, respectively.

$$FSObject_m = FSObject_p \wedge contents_m = contents_p$$

Likewise, more complex formulas can be provided, adding required flexibility for different categories of conformance. In our framework, the set of these definition formulas for a given model and a program is defined as the `mapping` function, as shown in the following specification:

```
mapping(m:Model, p:Program): Formula
```

The `mapping` formula is the hot spot of this framework, provided for a specific correspondence between object model and program declarations. This formula can be specified in a language based on first-order logic with transitive closure. For instance, $FSObject_m = FSObject_p$ is an example of equality formula between two expressions relating signature and class names. Although most formulas in conformance relationships will use equalities, additional formulas include subset, negation, conjunction and universal quantification, allowing developers to specify more complex relationships between model and program elements. The other kinds of formulas, such as existential quantification and disjunction, can be appropriately derived from the core constructs. Moreover, we consider some binary (union, intersection, difference, join and product) and unary (transpose and transitive closure) expressions. We show the language's formulas and expressions as follows.

```
formula ::= expr ∈ expr | expr ⊆ expr | expr = expr |
            ¬formula | formula ∧ formula |
            (∀ var: sigName | formula)
expr ::= setName | relName | className | fieldName | var |
         expr binop expr | unop expr
binop ::= ∪ | ∩ | - | . | →
unop ::= ~ | ^
```

In our core language for mapping formulas, we consider twelve kinds of expressions, which are specified next, using PVS datatypes [83]. There are expressions for signature, class, relation, field and variable names. Moreover, there are five kinds of binary expressions representing the union, intersection, difference, join and product expressions. Finally, there are the transpose and closure unary expressions.

```
Expression: DATATYPE BEGIN
  IMPORTING Names
    VARIABLE(n: VarName): VARIABLE?: Expression
    SIGNAME(n: SetName): SIGNAME?: Expression
    RELNAME(n: RelName): RELNAME?: Expression
    CLASSNAME(n: ClassName): SIGNAME?: Expression
    FIELDNAME(n: AtribName): RELNAME?: Expression
    UNION(l,r: Expression): UNION?: Expression
    INTERSECTION(l,r: Expression): INTERSECTION?: Expression
    DIFFERENCE(l,r: Expression): DIFFERENCE?: Expression
    JOIN(l,r: Expression): JOIN?: Expression
    PRODUCT(l,r: Expression): PRODUCT?: Expression
    TRANSPOSE(exp: Expression): TRANSPOSE?: Expression
    CLOSURE(exp: Expression): CLOSURE?: Expression
END Expression
```

A PVS datatype is specified by providing a set of *constructors*, *recognizers* and *accessors* [83]. The previous datatype presented constructors, such as `SIGNAME` and `UNION`, which allow the expressions to be constructed. For instance, the expression `SIGNAME(n)` is an element of this datatype if `n` is a signature name. The `UNION?` and `CLOSURE?` recognizers are predicates over the `Expression` datatype that are true when their argument is constructed using the corresponding constructor. For instance, `CLOSURE?(e)` is true when `e` is a closure expression. Suppose that we have the `UNION(e1,e2)` union expression, where `e1` and `e2` are expressions. We can use the `l` and `r` accessors to access the left and right expressions. For example, the `l(UNION(e1,e2))` expression yields the `e1` expression. When a datatype is type checked, a new theory is created that provides the axioms and induction principles needed to ensure that the datatype is the initial algebra defined by the constructors [83].

Additionally, the language defines seven kinds of formulas. Besides formulas representing true and false, there are negations, conjunctions, universal quantifications, subset and equality formulas. The set membership formula can be expressed in terms of the subset formula. Similar to expressions, we create a PVS datatype for formulas.

```
Formula: DATATYPE BEGIN
  IMPORTING Expression, Names
    TRUE: TRUE?: Formula
    FALSE: FALSE?: Formula
    NOT(f: Formula): NOT?: Formula
    AND(l,r: Formula): AND?: Formula
    FORALL(x:VarName, t:SigName, f:Formula): FORALL?: Formula
    SUBSET(l,r: Expression): SUBSET?: Formula
    EQUAL(l,r: Expression): EQUAL?: Formula
END Formula
```

Semantic conformance now needs to be redefined for the general framework. The `satisfies` predicate is defined as a recursive function. For example, assuming a universal quantification formula ∀ `x:exp • f`, the `satisfies` predicate checks whether the `f` formula is valid in the original interpretation extended with a value given to the variable name `x`, as formalized next. Notice that the same predicate is recursively applied for the quantification's sub-formula.

```
∀ v: i.mapSig(T) |
  satisfies(f,i ⊕ [i.mapSig:= i.mapSig ⊕ [x ↦ {v}]])
```

The predicate for semantic conformance states, for a given `mapping` formula, whether every heap of a given program corresponds to a specific interpretation that satisfies the formulas from `mapping` (represented by the `satisfies` predicate). In this case, the heap satisfies the invariants from the model – there is a corresponding interpretation.

```
semanticConformance(m:Model, p:Program): boolean =
  ∀ h:heaps(p) •
    ∃ i:semantics(m) • satisfies(mapping(m,p),i,h)
```

The evaluations of other formulas are very similar and have an the standard semantics. For example, an interpretation satisfies a conjunction formula when it satisfies each sub-formula. Moreover, an interpretation satisfies an equality formula (`exp1 = exp2`) when both subexpressions have the same values in the interpretation, as declared next.

```
evalExpression(exp1,i) = evalExpression(exp2,i)
```

The full specification of `satisfies` is declared next. For simplicity, we omit the details for recursion specification in PVS. Also, we use standard logical and set operators rather than PVS operators.

```
satisfies(f:Formula,i:Interpretation,h:Heap): RECURSIVE boolean=
  CASES f OF
    TRUE_: TRUE,
    FALSE_: FALSE,
    NOT_(f1): ¬satisfies(f1,i,h),
    AND_(f1,f2):
      satisfies(f1,i,h) ∧ satisfies(f2,i,h),
    FORALL_(x,t,f1):
      ∀v:objValue • v ∈ i.mapSig(t) ⇒
        satisfies(f1,i⊕[mapVar:=mapVar(i)⊕[x↦{ v1:objValue|v=v1 }]],h),
    EQUAL(e1, e2):
      evalExpression(e1,i,h) = evalExpression(e2,i,h),
    SUBSET(e1, e2):
      evalExpression(e1,i,h) ⊆ evalExpression(e2,i,h)
  ENDCASES
```

The `evalExpression` relation is a recursive PVS function that evaluates an expression for the given interpretation and heap values. Next, we specify the evaluation of a union expression (`exp1 ∪ exp2`).

```
evalExpression(e1,i,h) ∪ evalExpression(e2,i,h)
```

The evaluation of other expressions is specified similarly. For instance, the evaluation of a product expression is the product of each subexpression's evaluation. Next we specify the complete specification of `evalExpression`. We only consider a product with two single values (two signatures in Alloy). Symbols ~ and + denote relational inversion and transitive closure, respectively.

```
evalExpression(e:Expression, i:Interpretation, h:Heap):RECURSIVE ℙValue=
  CASES e OF
    SIGNAME(n): i.mapSig(n),
    RELNAME(n): i.mapRel(n),
    CLASSNAME(n): h.mapClass(n),
    ATRIBNAME(n): h.mapField(n),
    UNION(e1,e2):
      evalExpression(e1,i,h) ∪ evalExpression(e2,i,h),
    INTERSECTION(e1,e2):
      evalExpression(e1,i,h) ∩ evalExpression(e2,i,h),
    DIFFERENCE(e1,e2):
      evalExpression(e1,i,h) − evalExpression(e2,i,h),
    JOIN(e1,e2):
      {v:Value|∃v1,v2:Value •
                  v1 ∈ evalExpression(e1,i,h) ∧
                  v2 ∈ evalExpression(e2,i,h)  ∧
                  v = v1 ⨾ v2
      },
    PRODUCT(e1,e2):
      {v:objPairValue|∃v1,v2: objValue •
                  v1 ∈ evalExpression(e1,i,h) ∧
                  v2 ∈ evalExpression(e2,i,h) ∧
                  v = v1 × v2
      },
    TRANSPOSE(e1):
      evalExpression(e1,i,h)~
    CLOSURE_(e1):
      evalExpression(e1,i,h)+
  ENDCASES
```

5.4.2 Instantiations

The syntactic conformance relationship presented in Section 5.2 can be seen as an instance of this general framework. We now instantiate the indicated hot spot for the conformance relationships previously showed in this section. We also describe a technique for defining mapping formulas in terms of types of syntactic mappings observed in practice.

When establishing conformance, a syntactic conformance relationship is provided, linking each signature in the object model to a corresponding class in the heap (same for relations). Rinard and Kuncak [87] provide a classification for usually applied relationships. The traditional syntactic conformance for signatures and relations, respectively, are described as follows:

- **Class-Based mapping.** a set is mapped to all objects of a given class (including its subclasses);
- **Field-Based mapping.** a relation is mapped to all values for the corresponding field name – pairs of objects from the two corresponding classes.

Syntactic Conformance 1 uses class and field-based mappings (definitions `sigMapping` and `relationMapping`; `FSObject` and `Dir` signatures are implemented as classes, with `contents` and `name` as fields. These mappings were already formalized in Section 5.2 for our conformance relationship. The following PVS theorem formalizes the definition of mapping formula from these mappings. The theorem states that for every pair model-program in syntactic conformance following the traditional mappings, a mapping set of formulas is automatically defined, in terms of equalities for signature-class and relation-field pairs of names. For each signature and class with the same name (for instance `FSObject`), an equality formula (`EQUAL`) between those names is implied; the same is observed for a relation and an field name.

```
mappingFromClassfieldBased: THEOREM
  ∀ m:Model,p:Program,s:sigs(m),r:relations(m)  •
    sigMapping(s,p) ∧ relationMapping(r,p) ⇒
    mapping(m,p) =
      {f:Formula_ |
        ∃ s:sigs(m),c:classes(p) •
          s.name=c.name ∧
          f = EQUAL(SIGNAME(s.name),CLASSNAME(c.name))
        ∨
        ∃ r:relations(m),a:fields(p)) •
          r.name=a.name ∧
          f = EQUAL(RELNAME(r.name),FIELDNAME(r.name))}
```

As an alternative for relation mapping, Syntactic Conformance 2 follows a *collection-based* mapping:

- **Collection-Based mapping.** a relation is mapped to the values referenced by a collection object.

In our example, relation `contents` in the model is mapped by associating a file system object to the elements of its `List` objects in the heap. The following fragment defines this mapping for all relations from a model `m`, in which `targetType` denotes the function yielding its target type, which must be subtype of `Collection`, as in Java [48]. In the mapping relation, the formula for the `contents` is given by $contents_m{=}contents_p.elems$, where `elems` denotes the field from the collection to its set of elements.

```
CollectionBasedmapping(r:Relation,p:Program): boolean =
  ∃ f:fields(p) • r.name=f.name ∧  leftType(r)=leftType(f) ∧
    (isScalarR(r) ⇒ isScalarF(f) ∧ rightType(r)=rightType(f)) ∧
    (¬(isScalarR(r)) ⇒ isScalarF(f) ∧ rightType(f)=Collection)
```

The conformance framework can be general enough to allow the definition of different kinds of mappings for elements in the same model. For instance, some relations may be defined as class-based, while others use a collection-based mapping. In this case, only parts of the mapping relation are generated by following these kinds of mappings; other mappings can be used for particular names in the same model.

Although mapping types can be useful for developers in the indication of how concepts are implemented in source code, more complex definitions can be used to indicate this correspondence. In our example, Syntactic Conformance 3 indicates the subtype relationship as a field `type` into the `FSObject` class; when the value of `type` is `1`, it is considered a directory. Therefore, there is no direct mapping for the `Dir` concept in the program, indicating that correspondence is *content based* – the `Dir` concept in the heap is represented by `FSObject` objects whose field has a particular value.

The language for mapping formulas presented allows for more flexible definitions than simple correspondence between names, which offers the capability in defining complex content-based relationships (not only for inheritance, for sets and relations as well). For instance, the formulas for `FSObject` and `Dir` in Syntactic Conformance 3 could be represented respectively as follows, where value `1` for type represents directories as a set comprehension:

$$FSObject_m = FSObject_p$$
$$Dir_m = \{obj : FSObject_p \mid obj.type = 1\}$$

Although some set comprehension constructs do not appear in our definition for the formula language, we could easily derive it as shorthands for the core constructs, for making this logic practical. For instance, the existential quantifier can be built from the universal quantifier.

5.5 Chapter summary

In this chapter, we presented a formal definition for conformance between object models and programs, in order to relate refactorings at both levels of abstraction. This conformance relationship is broken into two separate definitions: syntactic and semantic conformance. These definitions have been specified and type-checked with the PVS language [83]. For semantic conformance throughout program executions, we delimited a notion of heaps of interest, using special statements **unpack** and **pack**. This defined conformance relationship is used as basis for the model-driven approach to refactoring detailed in the next chapter.

The notion of semantic conformance presented in this thesis can be applied as well to other syntactic conformance relationships, using the formal framework presented in Section 5.4. The hot spot of this framework is a correspondence definition between syntactic declarations in object models and programs, which generates a mapping formula used for establishing the semantic conformance.

Chapter 6

Model-Driven Formal Refactoring

Given a specific conformance relationship between object models and programs, we relate refactorings on both levels. For model refactoring, we adopt the approach of primitive transformations - based on sound laws of modeling - being composed into refactorings that may be applied by designers to improve object models. Our investigation then lies on the relationship between these formal model transformations and program refactoring; transferring those changes is still a challenge for model-driven development approaches. Starting from investigations on the relationship of laws of modeling and programming, we devised an approach to semi-automatically refactor programs, based on model refactorings applied by designers.

In this chapter, we present the core contribution of this thesis: a model-driven approach to refactoring. We first present a motivating example (Section 6.1), and the approach itself is delineated in Section 6.2. The key concept for realizing model-driven refactoring – *strategies* – is detailed in Section 6.3. Section 6.4 applies the solution to the motivating example. Finally, complementary topics are discussed in the last three sections.

6.1 Motivating Example

First, we present an example that illustrates problems in conformance maintenance during evolution, specifically during refactoring. The following program fragment shows a direct implementation for the file system model from Chapter 3, using the BN language. The `main` method, within the `Main` class, reads a file system object from the user (**self**`.inout`), and in the case of a directory, adds a new `File`; otherwise, none is added. The keyword `is` corresponds to Java's `instanceof`. Following the invariant from the object model, only directories may have contents. In addition, the multiplicities are guaranteed, as, for instance, every file system object has exactly one name, and there is only one `Root` instance. We assume `contents` as a public field, to illustrate accesses in the subclasses.

```
class FSObject{
  Name name;
  pub set FSObject contents;
  set Dir getContents() { result:= self.contents }
  unit setContents(set FSObject c) { self.contents:= c }
}
```

```
class Dir extends FSObject{...}
class File extends FSObject{
   constr { .. self.contents:= ∅ }

class Main{
   unit main(){
      File f:= new File,currentFSObj:= null in ..
        currentFSObj:= (FSObject)self.inout;
        if (currentFSObj is Dir)
          then currentFSObj.setContents({f})
          else currentFSObj.setContents(∅)
    }
}
```

The program shows a situation for refactoring: there is a field in the superclass – `contents` – which is only useful in one of the subclasses – a “bad smell” [37]. The refactoring can be accomplished by moving `contents` declaration down to the `Dir` class. Additional changes are needed for preserving behavior, such as updating accesses to `contents` with casts. Issues with evolution arise as the object model is updated for conformance. Evidently manual updates to the model soon become impractical in evolving systems, thus not considered here. RTE tools [90] are a popular choice for automation, applying reverse engineering for recovering object models from code. Usually in this case model invariants are hardly recovered or correctly modified. Also, models become rather concrete, as simple *visualizations of source code*. This is commonly seen in generally mainstream CASE tools [91], in which the reverse engineering task creates UML models (usually class diagrams) that faithfully duplicate declarations and relationships taken from the program. The process produces cluttered diagrams with difficult visualization.

On the other hand, the mentioned “bad smell” may also be detected at the object model level. The binary relation may then be pushed down from `FSObject` to `Dir`. Likewise, conformance is desirable; in the context of RTE, a usual technique for automatic updates to the source code marks previously edited code fragments as immutable, in order to avoid loss of non-generated code that is marked by the tool. However, if the field is simply moved to `Dir`, *these immutable statements become incorrect*; for instance, accesses to the field with left expressions not exactly typed as `Dir` within `FSObject` and `File`, as highlighted in the following fragments.

```
class FSObject{ ..
  set Dir getContents() { result:= self.contents }
  unit setContents(set FSObject c) { self.contents:= c } }
class File extends FSObject{ ..
   constr { .. self.contents:= ∅} }
```

Due to the representation gap between object models and programs, the immutable code may rely on model elements that were modified, showing a recurring evolution problem in RTE-based tools. Manual updates must be applied, making evolution more expensive. Also, no evidence is provided on whether conformance is maintained, as the updates are manual.

A few tools present an alternative by following the MDA [66]. Models in these tools include programming logic written in a platform-independent language – for instance, action semantics of executable UML [7]. Tool support then generates source code for that logic in a specific implementation platform. Under such approach the model refactoring in this section’s example would also include changes to the programming logic attached to models, maintaining an updated source code. Although the MDA approach might be promising, it lacks a more abstract view of the domain given by object models, which is still useful for an overall understanding. Our approach investigates model-driven refactorings in this context.

6.2 The Approach

Our solution considers the context in which object models can be refactored using the laws of modeling, directly or indirectly (in the latter case, by using refactorings derived from primitive transformations). To each Alloy law from the catalog in Table 3.1 we associate a set of conditional program transformations – *strategies* – to be applied to a program, with minimal user intervention, guided by laws of programming.

The mechanics of the approach is depicted in Figure 6.1, where OM represents an object model, and P a program. The first step, described in (a), repeats the first two law applications from Refactoring 1 (*push down relation*); the model refactoring is the first step, requiring intervention from the developer in choosing the appropriate refactoring to apply. *After the model refactoring*, all law applications and information about the intermediate models are recorded (for instance, in a CASE tool), for association with the strategies. We associate each application direction of an Alloy law (depicted as "corresponds" in Figure 6.1; for instance, `Law 2` corresponds to `Stg 2`) with a specific strategy, which semi-automatically refactors the program in (b), resulting in a program consistent with the refactored program. Strategies are constructed as the semi-automatic application of laws of programming, denoted as L_k.

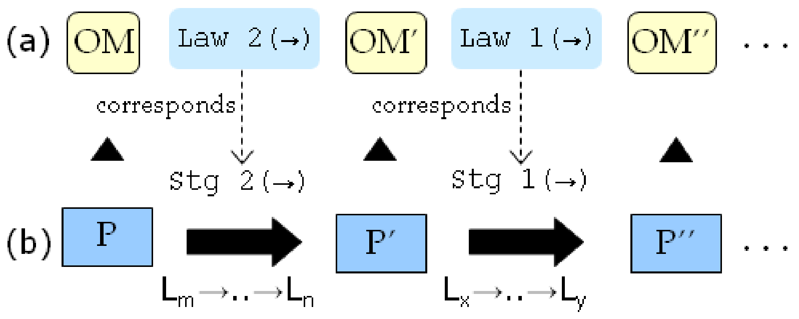


Figure 6.1: Model-driven refactoring with strategies

A strategy carries out program refactoring by applying a set of transformations, on the assumption of the conformance relationship defined in Chapter 5. Therefore, the strategies are especially conceived to exploit the model invariants that are known to be met by the program; hence, program transformations can be more powerful with high-level assumptions about program entities, such as classes and fields. In fact, the only preconditions for the application of strategies is that the program must have confinement for a subset *Own* of classes and in syntactic and semantic conformance with the original model (before the model refactoring), as depicted in Figure 6.2.

In this context, a strategy must exhibit the following characteristics:

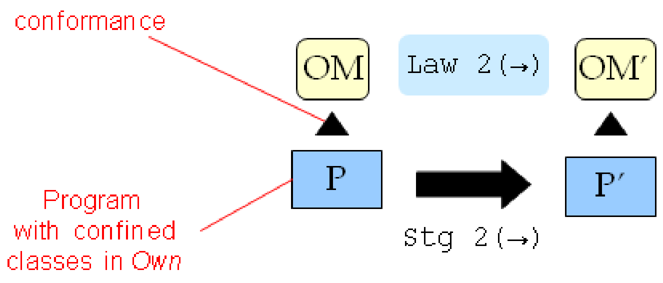


Figure 6.2: Conformance as Preconditions for Strategies

- It rewrites programs for updating correspondent abstractions that were refactored in the object model;
- It preserves program behavior.

Therefore, every strategy fulfills a few requirements: its application results in programs that are *refinements* and establish the *conformance relationship with the refactored model*, syntactically and semantically. In addition, confinement is maintained. Proofs of these properties are developed in Chapter 7.

Our approach differs from RTE as it avoids re-generation of source code from models, which allows developers to avoid most manual updates. In fact, object model and program refactorings take place independently. Similarly, MDA-based tools are different as they include programming logic information, which yields more concrete models.

6.3 Strategies

Strategies are composed of a number of law applications in the program. The resulting program refactoring preserves the behavior of the original program by construction, resulting from the application of laws of programming. In addition, strategy steps were designed to accept any program within the conformance relationship, which is relatively open for a number of possible implementations. We formalized strategies in a notation for refinement tactics, which we explain next. Examples of strategies corresponding to two of the Alloy laws are fully explained afterwards.

6.3.1 Refinement Tactics: Angel

Strategies are formalized as *refinement tactics*, based on the Angel language [71]. The language adds preciseness to the description, allowing easier implementation in the transformation language of choice. Also, Angel constructs are appropriate for describing law applications, with the needed parameters and features for repeating applications to several elements throughout the program. In fact, the language is employed to document strategies as refinement tactics, used later in our approach as single transformation rules. In addition, the language presents a formal semantics [71].

The language allows *tactics declarations*. Atomic tactics may be a simple rule applications, which in this work are adapted to the called *law application*, indicating the law name to be applied with arguments. These arguments may be predicates, expressions, variables, classes, fields or methods. A law application may have two possible outcomes: if the law is applicable (all provisos are satisfied), then the application actually occurs, and the program is changed. Otherwise, the application of the tactic fails. The following fragment applies Law 3 (*class elimination*) to the program, with two arguments: the class to be eliminated (C) and the application direction (in this case, from Left to Right $\rightarrow$). Law names are reduced tags for the laws of programming presented in this work, in Chapter 4 and Appendix B.

law $\mathit{classElimination}(C, \rightarrow)$

Similarly, atomic refactoring tactics can be applied with the **refactoring** clause, and auxiliary tactics are applied with the **tactic** clause. There two possible outcomes after applying an atomic tactic: if the law or refactoring matches the expression (program, for instance) then the tactic is applied, producing a transformed expression. But if the tactic does not match, then the application fails [71]. Two special atomic tactics may be used: **skip** and **fail**. The first always succeeds, leaving the expression unchanged, while the latter defines a failed application.

Tactics can be composed of sequences or alternations; in order to sequentially composing two tactics, the $t_1;\ t_2$ construct can be used. t_1 is first applied to the program, and then t_2 is applied to the result of the application of t_1. If either t_1 or t_2 fails, so does the whole composite tactics. Similarly, tactics combined in alternation have the form $(t_1 \mid t_2)$. First, t_1 is applied to the program; if this application is successful, then the composite tactic succeeds. Otherwise t_2 is applied. Finally, if t_2 fails, then the whole tactic aborts (which is a more critical situation than failure). When the tactic contains many choices, the first choice that succeeds is selected – an application of the angelic nature of nondeterminism, from which the language earned its name [71].

We extended the language with built-in query functions with meta-information about object-oriented declarations. For instance, **firstCommonSuper**$(< C_1..C_n >)$ yields a class that is the first common superclass of the classes in the argument list. Similarly, built-in operations are added for changing program declarations. **setExtends**(C, CC) replaces C's **extends** clause with class CC. These operations do not present any side effects on the program, as it is only transformed by tactic applications.

Tactics are more often useful if repeatedly applied, until their application is impossible. A common need in strategies is quantification when applying tactics. For repeated application of a single tactic (law or refactoring) to several declarations of the same kind, we consider an additional parameter for a tactic. For instance, the tactic **law** $\mathit{moveRedefinedMethodToSuperclass}($**redefinedMeths**$(X),$ **super**$(X), \rightarrow)$ applies Law 35 *move redefined method to superclass* repeatedly, from left to right, to all redefined methods in class X (**redefinedMeths**(X)), moving these methods to X's superclass (**super**(X)). For this to work, we consider that there are two versions for each RN law, although not showed in the thesis: one for single application, and other for multiple application (only the first version appears in Appendix B).

In addition, the language allows us to define pattern matching within a program, with the constraint **applies to**. For instance, **applies to** $\mathit{cmd}[(X)e]$ **do**t applied the t tactic to every command in the program that includes an expression e cast with X. In particular, this pattern matching may be applicable to a specific program declarations; in this case, we incorporate *structural combinators* in the tactic language [71], permitting the controlled application of tactics to sub-expressions. For example, the following fragment applies the t tactic only to methods of class S.

$\boxed{\textbf{class}\ \texttt{S}}\ t\ \boxed{\}}$;

The **Tactic** $n(a)$ t **end** declaration defines a predefined tactic that can be applied as a single law. Strategies in our approach are declared as such. Also, we extended the syntax by introducing types to the parameters in a. Although law applications may generate proof obligations, due to their provisos, in strategies we consider that law applications are valid, an the provisos are fulfilled. The proof developed for ensuring soundness of law applications are developed in Chapter 7 and Appendix E. In addition, we take the closed-world assumption that strategies have access to the full source code of a program.

6.3.2 Examples of strategies

For each recorded model transformation, strategies are semi-automatically applied following the correspondence in Table 6.1. Other Alloy laws from the relatively complete catalog [42] do not correspond to strategies, as they deal with syntactic sugar in the model, which does not affect source code. In this section, we describe a number of strategies that will be applied in the file system example. These and other strategies that we defined are presented in Appendix D.

Table 6.1: Strategies corresponding to Alloy laws

Alloy Law	Strategy →	Strategy ←
1.Introduce Relation	*introduceField*	*removeField*
2.Introduce Subsignature	*introduceSubclass*	*removeSubclass*
7.Introduce Signature	*introduceClass*	*removeClass*
8.Introduce Generalization	*introduceSuperclass*	*removeSuperclass*
16.Split Relation	*splitField*	*removeIndirectReference*
18.Remove Lone Relation	*fromOptionalToSetField*	*fromSetToOptionalField*
17.Remove One Relation	*fromSingleToSetField*	*fromSetToSingleField*

introduceSubclass

Alloy Law 2 (L-R) introduces a subsignature for one of the declared signatures. All objects that were instances of the supersignature are now instances of the newly-introduced signature `X`, making the `U` supersignature abstract using an invariant (`X=U-S-T`). The corresponding strategy – whose intuition is depicted in Figure 6.3 as a UML class diagram representing the program's classes and fields – accepts a conforming program, in which there is a superclass `U`. However other (abstract) classes can be declared in the hierarchy – we adopt a free representation of inheritance in which a dotted line replaces a undetermined hierarchy. For instance, an additional class could be implemented for the file system model which is not in the model, but is declared in the program as a subclass of `FSObject` and a superclass of `Dir`. The notation $CT[X]$ represents the other declarations of the class table, also indicating that class `X` may be already declared in CT.

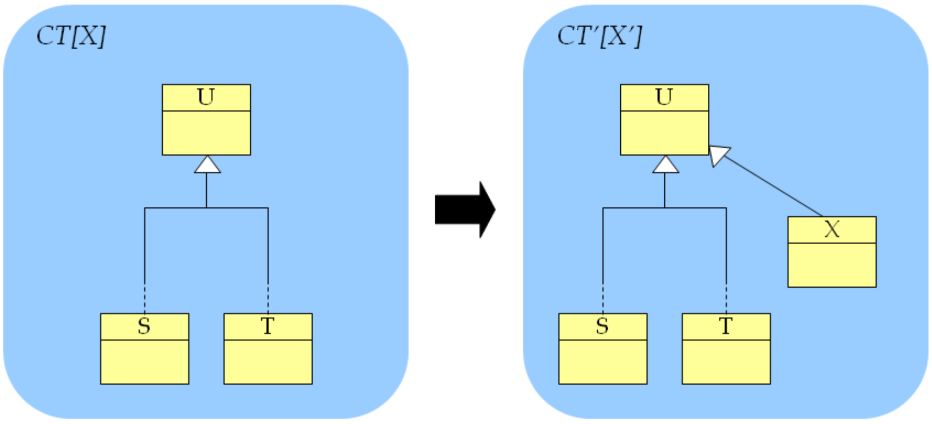


Figure 6.3: Introduce subclass strategy

If class `X` is already present, it is freely renamed to `X'`, because `X` is considered in this case an *implementation detail* that was not modeled; $rename(X, X')$ is a trivial law that is applied to the entire program, renaming `X` to `X'`. If it fails (for the case in which the class is not declared), nothing happens, with the **skip** tactic. Renaming contributes to make conditions of the laws valid, affecting program declarations; this action does not have impact on the conformance relationship, as the renamed declarations are not in the model. The strategy introduces the new class `X` as a direct subclass of `U`, even though the subclasses implementing other `U`'s subsignatures are not direct in the program – **setExtends** defines `U` to be in `X`'s **extends** clause. Next, every `U` object creation in the entire program is replaced by `X` instantiations, by Law 4 (`[`**new** `X/`**new** `U]`).

Tactic $introduceSubclass(X, U : Class)$
    ($\textbf{law}\ \ rename(X, X')\ \ |\ \ \textbf{skip}$);
    $\textbf{law}\ \ classElimination(\textbf{setExtends}(X, U), \leftarrow)$;
    $\textbf{law}\ \ newSuperclass(U, X, \rightarrow)$;
end

The laws within a tactic are applicable; the strategy was developed for fulfilling every proviso defined for the law – the previous law always adjusts the program for allowing the application of the next law. However, for not cluttering strategy definitions, we leave this discussion for the proofs in Chapter 7 and Appendix E, more specifically in the refinement proof.

removeSubclass

The opposite direction of Alloy Law 2 removes subsignature `X` with the presence of the invariant (`X=U-S-T`); `S` and `T` become the only `U`'s subsignatures. The strategy removes the corresponding `X` class, although, differently from the model, the program class may declare fields and methods, and may have subclasses with no correspondence in the model. These implementation details must be rearranged, as showed in Figure 6.4.

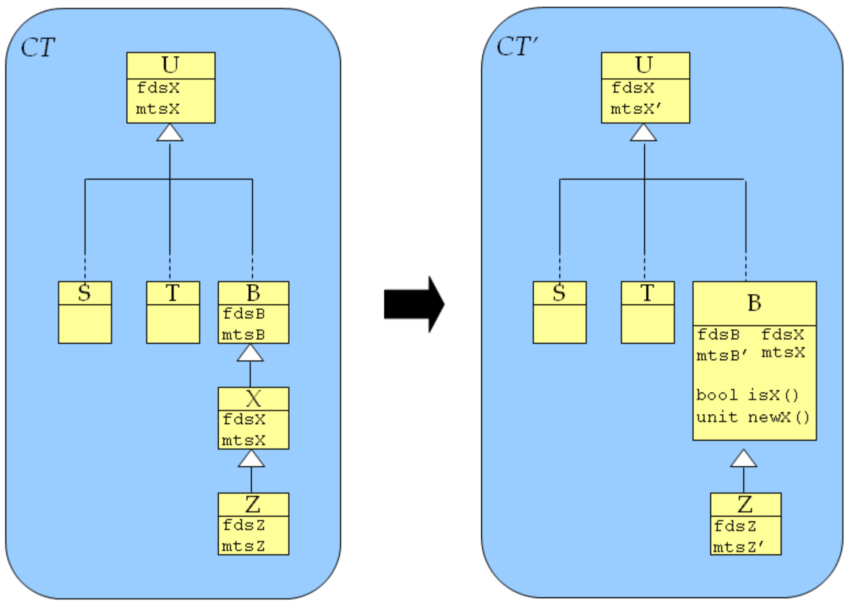


Figure 6.4: Remove subclass strategy

We describe the auxiliary tactics first, then the strategy's main tactic. The auxiliary tactic *moveUpFields* pulls up the fields declared in `X` (**fields**(X)) to the immediate superclass `B`. This operation is possible even if another superclass in the same hierarchy declares a field with the same name – we ignore shadowing in the language. An alternation is presented for pulling up fields: if `B` has any superclass declaring the moving field, Refactoring 5, as formalized by Cornélio [26], moves two or more fields with the same name to the superclass in one step; if it is not the case, a simple law of programming (Law 37) is applied. The query **super**(X) returns `X`'s immediate superclass.

```
Tactic  $moveUpFields(X : Class)$
    (law  $moveFieldToSuperClass(\mathbf{fields}(X), X, \mathbf{super}(X), \rightarrow)$  |
      refactoring  $pullUpPushDownField(\mathbf{fields}(X), \mathbf{super}(X), \rightarrow)$;
end
```

Next, `X`'s methods are pulled up as well, with the auxiliary tactic *moveUpMethods*. In this case, the strategy must deal with two cases: redefined and non-redefined methods. For the first, the methods yielded by **redefinedMeths**(X) are removed from `X` and the corresponding method body in the superclass is modified with an **if** command that adds the body of the moved method (Law 35). Yet, these steps will not work if any of these methods contain accesses to **super**; in this case, the auxiliary tactic named *adjustHierarchyForPullingUpMethods* eliminates all **super** method calls by inlining those calls from **object** to `X`, top-down in the hierarchy; for this, all private fields in this hierarchy are first made public. Next, the non-redefined methods (**nonRedefinedMeths**(X)) must be copied to other `B`'s subclasses (**immedSubs** (**super**(X))), with an empty body, so no type errors occur when `B` receives the moved method declaration with Law 36 *moveoriginalmethodtosuperclass*.

```
Tactic  $adjustHierarchyForPullingUpMethods(C : Class)$
    law  $changeVisibilityPrivatePublic(\mathbf{getHierarchyTopDown}(C),$
        $\mathbf{priFields}(\mathbf{getHierarchyTopDown}(C)), \rightarrow);$
    law  $eliminateSuper(\mathbf{getHierarchyTopDown}(C), \rightarrow);$
    applies to  $exp.x, exp.m(\bar{e})\{\mathbf{isExactly}(exp, C)\}$  do
        law  $introduceCastToExpression(C, \rightarrow);$
end
```

```
Tactic  $moveUpMethods(X : Class)$
    law  $moveRedefinedMethodToSuperclass(\mathbf{redefinedMeths}(X), \mathbf{super}(X), \rightarrow);$
    (law  $MethodElimination(\mathbf{immedSubs}(\mathbf{super}(X)),$
      $\mathbf{nonRedefinedMeths}(X), \leftarrow)$  |  skip);
    law  $moveOriginalMethodToSuperclass(\mathbf{meths}(X), \rightarrow);$
end
```

After removing its fields and methods, class `X` is replaced next by its superclass on all declarations over the program, with tactic *changeDeclarationsTypetoSuper*.

```
Tactic  $changeDeclarationsTypetoSuper(C : Class)$
    applies to  $C\ x$  do  law  $changeFieldType(x, \mathbf{super}(C), \rightarrow);$
    applies to  $C\ x := e$  in  do  law  $changeVariableType(x, \mathbf{super}(C), \rightarrow);$
    applies to  $T\ meth(C\ x, pars)$  do  law  $changeParameterType(x,$
      $\mathbf{super}(C), \rightarrow);$
    applies to  $C\ meth(pars)$  do  law  $changeReturnType(x, \mathbf{super}(C), \rightarrow);$
end
```

The auxiliary tactic *eliminateTypeTests* removes type tests involving `X` works by replacing inheritance as the source of typing information. For that, we create a string field, named `type`, which is used for type tests, rather than the **is** command, which will be removed for tests involving class `X`. The tactic begins by creating a method named `isX()` that includes a test to `X`. Then all `X` type tests are replaced by a call to this method, as exemplified in the following program fragment:

```
class B { ..
  bool isX(){
    result:= self is B
  }
}
..
if (x=null)
  then test:= false
  else test:= x.isX()
```

This replacement is elaborate, since every occurrence of `x` **is** `X` must be replaced by a special statement, a *parameterized command* [16], by Laws 22 *var block-val* (R-L), 23 *var block-res* (L-R), 24 *pcom elimination-res* (R-L) and 25 *if identical guarded commands* (R-L), in this order. A parameterized command of the form `test:= (`**result**`:= x` **is** `X)` may then be replaced by a method call to `isX`. For avoiding null pointer errors, we introduce an **if** statement for ensuring that the expression being tested is not null. After these replacements, field `type` is then introduced, with Law 28(R-L), and initializations to this field are added to `X`'s constructor and every constructor in `X`'s subclasses, as shown in the following declarations:

```
class B { ..
  string type; ..
}
class X extends B {
  constr { ..; self.type:= "X"}
}
class Z extends X {
  constr { ..; self.type:= "Z"}
}
```

Next, by class refinement, the remaining type test in method `isX` is replaced by a test over the `type` field, in terms of the values given by the constructors, which maintains the desired behavior for the test (either `X` or `Z` instances are typed `X`). The following example considers `Z` as `X`'s only subclass.

```
class B { ..
  string type; ..
  bool isX() {
    result:= (self is X) ∨ (self is Z)
  }
}
```

In the tactic, this replacement is represented as string concatenation involving names of the subclasses – we added this feature to Angel, extending the language for representing statement construction. A special *disjunction* variable statement is created and manipulated (with the **createDisjunction** operation and the **addToDisj** special law applications), being constructing from the names of all `X`'s subclasses. The exemplified `isX()` method is represented in the following tactic by the *isMethod* parameter. The special function **replace** substitutes the fragment as argument by the second argument.

```
Tactic $eliminateTypeTests(X : Class, isMethod : Method)$
      law $methodElimination(isMethod, \textbf{super}(X), \leftarrow)$;
      applies to $cmd[x$ is $X]$ do
         law $varBlockValue(cmd[x$ is $X], test, \textbf{true}, \rightarrow)$;
         law $varBlockResult(\textbf{bool}\ test := x$ is $X, \textbf{result}, \rightarrow)$;
         law $pcomEliminationResult(\textbf{result} := x$ is $X, \leftarrow)$;
         law $ifIdenticalGuardedCommands(test := (\textbf{result} := x$ is $X)$,
               “$x = \textbf{null}$”, “$test := \textbf{false}$”, “$test := (\textbf{result} := x$ is $X)$”, $\leftarrow)$;
         law $methodCallElimination(test := (\textbf{result} := x$ is $X)$,
               “$test := x.isX()$”, $\leftarrow)$;
         law $varBlockValue(\textbf{bool}\ b, \leftarrow)$;
      law $fieldElimination($“string $type$”, $\textbf{super}(X), \leftarrow)$;
      applies to $cmd[x$ is $X]$ do
       law addtoEnd(constructor$(cc)$, “self.$type$ = ” + name$(cc)$);
      $disjunction :=$ createDisjunction();
      addToDisj$(subclasses(X),\ disjunction,$ “self is ” $+\ name(cc))$;
      applies to $isMethod$ do
         law replace(“self is $X$”, “self.$type$ = ” + name$(X)$ +“∨” + $disjunction)$;
end
```

After the previous tactic, `X` declares only a constructor, which must be replaced, since we intend to replace every **new** `X` with **new** `B`. Hereafter, we consider a command of type `x:=` **newX** to be a syntactic sugar for the following sequential composition: `x:=` **new'** `X; x.newX()`, in which **new'** is the regular instantiation of an object, whose reference is assigned to `x`. It is followed by a call to `newX`, a method of class `X` containing the actual constructor body, used for initializing fields. After defining this replacement for every `X` instantiation, the strategy moves `newX` to the superclass `B` (which contains the initialization for the `type` field). After this, the **new'** `X` commands can be replaced by **new'** `B` commands in the whole program. An excerpt of the result can be seen next, before the *eliminateNew* tactic.

```
class B { ..
  string type; ..
  bool isX() {
    result:= self is X ∨ self is Z
  }
  unit newX() {
    { .. self.type:= "X"}
  }
}
class X extends B { }
..
B x:= new' B; x.newX(); ..
```

```
Tactic eliminateNew(X : Class)
  applies to "x := new " + name(X) do
    law replace("x := new " + name(X), "x := new'"
      + name(X) + "; x.newX()");
  law moveOriginalMethodToSuperclass(X, method("newX"), →);
  law newSuperclass(X, super(X), rightarrow);
end
```

In the main tactic, casts to `X` are removed with Law 39 (type tests and **new** commands are also eliminated, in rather elaborate tactics that we explain later in this section). After changing the **extends** clause of `X`'s subclasses, `X` can finally be eliminated.

```
Tactic removeSubclass(X : Class)

    tactic moveUpFields(X);

    tactic adjustHierarchyForPullingUpMethods(X);
    tactic moveUpMethods(X);

    tactic changeDeclarationsTypetoSuper(X);
    applies to cmd[(X)e] do law eliminateCastExpressions(cmd[(X)e], →);
    tactic eliminateTypeTests(X, "bool isX(){ result := self is X }");
    tactic eliminateNew(X);

    law changeSuperfromEmptyToImmediateSuperclass(immedSubs(X),
        super(X), →);
    law  classElimination(X, →);
end
```

introduceField

Alloy Law 1 introduces a relation to a given signature, along with a definition of its values by means of an invariant. Any type-compatible expression can be assigned to the new relation; for a conforming program, *class refinement* may be applied, since we may add a new member to the target class, changing the methods in a way that satisfies a coupling invariant. This context results in an issue for program refactoring: designing a general solution for generating coupling invariants for *any* relation expression. This problem is discussed in detail in Section 6.5; we developed strategies for a few specific cases of relation expressions. Here, we show the strategy for introducing a new field `r` whose values are taken from a composition of two other fields, `x` and `y`. Figure 6.5 depicts the intuition behind the strategy steps. From the syntactic conformance, fields `x` and `y` are declared. The methods in class `S` must be modified for satisfying the coupling invariant (from the model, **self**`.r = `**self**`.x.y`). Also, any name conflicts with the new field are managed with renaming.

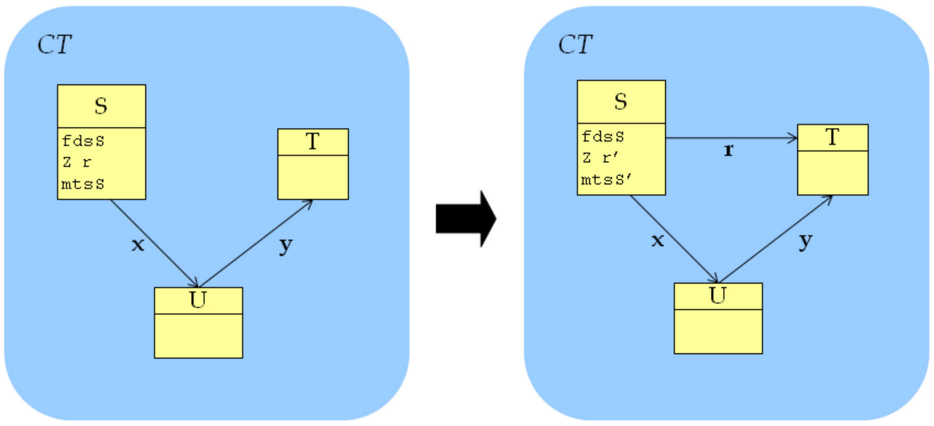


Figure 6.5: Introduce field strategy

The strategy changes the visibility of field `y` to **pub**, in order to make it visible from `S`. The class refinement is applied: field `r` is introduced, then changes to the methods in `S` are carried out, based on the coupling invariant. For instance, any writing to **self**`.x.y` must be duplicated to **self**`.r`, in order to maintain the invariant; we freely adopt $exp.f$ when exp is a set, whose outcome is a set of objects that result from dereferencing f in every element of exp ($exp.f = \{o \mid \exists\ el : exp \bullet el.f = o\}$). On the other hand, we consider that the language does not allow method calls from set expressions, which would bring additional complexity to the language. Also, from the confinement property we consider that field `y` is changed only by method `setY`, and this method is added by Refactoring 4 that encapsulates the field – formal program refactorings from Cornélio's thesis are applied in the strategy with the **refactoring** tactic. In this case, the getter method has *module scope* (for a module containing `S` and `U`), so it cannot be called from client classes; for this kind of method, the confinement rules can be relaxed [8], then `getY` is allowed in `U`. The validity of the class refinements applied in this strategy is proved by simulation in Chapter 7.

```
Tactic $introduceCompositionField(r, x, y : Field, S, U : Class)$
    (law $rename(r, r', S)$ | skip);
    (law $changeVisibilityPrivatePublic(y, U, \rightarrow)$ | skip);
    law $fieldElimination(r, S, \leftarrow)$;
    refactoring $selfEncapsulateField(y, U, \leftarrow)$;
    $\boxed{\texttt{class S}}$
      applies to self.$x.y := exp$ do
        law replace(“self.$x.y := exp$”,
        “unpack self; self.$x.setY(exp)$; self.$r := \{$self.$x.getY()\}$; pack self”);
    $\boxed{\}}$
    $\boxed{\texttt{class S}}$
      applies to self.$x := exp$ do
        law replace(“self.$x := exp$”,
        “unpack self; self.$x := exp$; $if$(self.$x$ = null) then self.$r := \varnothing$
        else self.$r := \{$self.$x.getY()\}$; pack $self$”);
    $\boxed{\}}$
end
```

removeField

The opposite application of Alloy Law 1 removes a relation based on its invariant definition. Again, we exemplify the strategy for removing a field based on a composition of two other fields, `x` and `y`. Figure 6.6 depicts the changes on the involved classes. The methods in class `S` must be modified by using the model invariant for establishing the refinement (**self**.`r` = **self**.`x`.`y`).

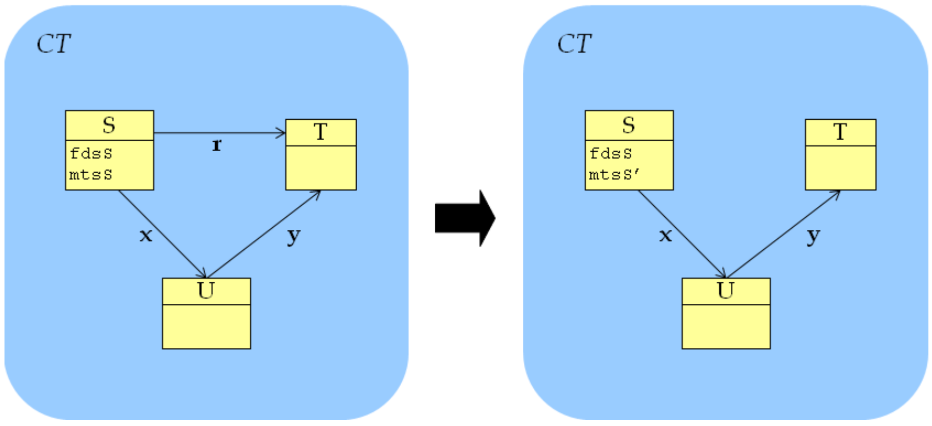


Figure 6.6: Remove field strategy

The refinement is the most important transformation in the strategy. First, the `r` field must be made private, by Refactoring 4, which encapsulates `r` and creates the appropriate `get/set` methods. The methods in `S` can then be modified for removing references to `x`, replaced by the `x.y` composition.

```
Tactic $removeCompositionField(r, x, y : Field, S, U : Class)$
    ($\textbf{refactoring}\ selfEncapsulateField(r, S)\ \mid\ \textbf{skip}$);
    ($\textbf{law}\ changeVisibilityPrivatePublic(y, U, \rightarrow)\ \mid\ \textbf{skip}$);
    $\textbf{replace}(``\textbf{self}.r := exp", ``\textbf{self}.x.y := exp;\ ")$;
    $\textbf{replace}(``exp[\textbf{self}.r]", ``exp[\textbf{self}.x.y]")$;
    $\textbf{law}\ fieldElimination(r, S, \rightarrow)$;
end
```

6.4 Illustrating Strategies in the Example

In this section, we apply the presented approach with a few strategies to the motivating example. Regarding the file system, the object model invariant guarantees that the `contents` relation is empty for any instance of `FSObject` subclasses, except for `Dir` instances. In our conformance relationship, the invariant is always true for any program state, so we consider that a program in conformance maintains the invariant for all executions. This allows strategies to be semi-automatically (as detailed in the next section) applied for each law employed in the object model refactoring.

After the application of Alloy Refactoring 1, in fact a number of Alloy laws were applied. First, Law 2 (L-R) was applied; it corresponds to the strategy *introduceSubclass*, according to Table 6.1. Accordingly, the empty class `X` is introduced as a subclass of `FSObject` (since `FSObject` is abstract, no **new** command is substituted).

```
Class X extends FSObject { }
```

Next, auxiliary fields `contentsDir`, `contentsFile` and `contentsX` are introduced with strategy *introduceField*, establishing a new invariant for each subclass (**self**.`contentsFile` = **self**.`contents` ∧ **self** **is** `File`); this refinement cannot be automatic, even though the coupling invariant may be generated from the model invariant; in this case modifications based on this coupling invariant are not trivial (automation issues are fully discussed in Section 6.5). Writings to **self**.`contents` are then extended with additional commands, as exemplified for class `File` below. In the outcome the desired invariant is enforced.

```
class File { ..
  constr {
    pack self;
      self.contents:= ∅;
     if (self is File) then ((File)self).contentsFile:= self.contents
    unpack self;
  }
}
```

Next, manipulation of model formulas, such as the deduction of invariant `contents = contentsDir + contentsFile + contentsX`, do not affect the source code, as identity strategies are associated with these Alloy laws. Formulas can only be introduced when they are implied by other invariants in the model, which by definition are fulfilled by the program. The subsequent strategy – *removeField* – removes the `contents` field based on a given invariant definition for the corresponding relation in the model; this strategy uses class refinement for removing the field, after replacing its occurrences with the equivalent expression from the invariant. Also, auxiliary fields `contentsFile` and `contentsX` can be removed, since they are always empty sets, from the model.

After renaming the new `contentsDir` field to `contents` and removing the auxiliary subclass `X` (with strategy *removeSubclass*), a partial view of the resulting program is showed in the following fragment. The access to the previous `contents` field is preceded by **if** commands and casts, making the resulting program syntactically correct and conforming to the refactored model.

```
class FSObject{ ..
  set Dir getContents() {
    if (self is Dir) then result:= ((Dir)self).contents
  }
  unit setContents(set FSObject c) {
    ((Dir)self).contents:= c
  }
}
class Dir extends FSObject{
  pri set Dir contents ..
}
class File extends FSObject{
  constr { .. }
}
class Main{ ..
  unit main(){ ..
      if (currentFSObj is Dir)
        then currentFSObj.setContents({f})
        else currentFSObj.setContents(∅)
  ..
}
```

6.5 Discussion on Strategy Automation

When describing our approach, we mentioned that strategies are *semi-automatic*, as their automation is limited by a number of issues, which are delineated in this section. The main problems occur in strategies that include class refinement steps, whose proofs are not automatic. Also, even though strategies refactor the program correctly, the quality in the resulting program – an important requirement for refactorings – is not guaranteed.

6.5.1 Data refinement

Some of the strategies involve changes in the internal representation of a class or hierarchy. For instance, this is crucial in the strategies corresponding to Law 1, which introduce a field with a class invariant, and a value for the field is enforced. The strategy *introduceCompositionField* is defined as a class refinement that introduces a new field, with an invariant establishing its value to be a composition of two other fields. Other strategies demanding class refinement are *fromOptionalToSetField* and *splitField* (Appendix D).

When introducing a field, the relation definition restricts possible implementations of the newly-added field, having a clear impact on the strategies; the definition drives how the coupling invariant for the simulation is specified. Since the law is open for any equational definition, full automation of these strategies is impossible. In this context, we chose to define strategies for two common definitions besides composition: empty field (CI : **self**.$r = \varnothing$) and clone field (CI : **self**.$r =$ **self**.t). These strategies are showed in Appendix D.

The coupling invariants for those strategies are straightforward translations of the formula in the Alloy model. However, this is not always the case; for example, in one of the case studies in Chapter 8 a relation is defined as `r = ~x + ~y` (the `~` operator yields the *transpose* from the operand relation). The generation of a coupling invariant from this invariant is not straightforward (transpose of a field dereference is not directly supported by currently used programming languages), and even harder it is to calculate the refinement for an arbitrary class.

This problem is identified in thesis, albeit we do not intend to provide a solution – we only discuss potential ways to tackle the problem. Two main challenges for general class refinement became evident during our investigation: coupling invariant translation and proof.

Invariant translation

In our approach, steps that apply class refinement are model-driven. Therefore, it is feasible to adapt modeled invariants as coupling invariants. Alloy Law 1 allows modelers to include a relation, as long as the relation is defined out of other previously declared relations (relation `r` with no specific definition can be introduced as a special case of the law, with definition `r=r`). In this context, coupling invariants can be easily established from simple definitions. Figure 6.7 shows a coupling invariant that is established from an Alloy formula defining a clone relation – `r` is created with the same values as `t`. The coupling invariant is directly taken from the relation definition, and its semantic effect is clearly the same as modeled. The simulation might be easily determined in the constructor with an additional assignment **self**`.r:=` **self**`.t`, and this assignment must be repeated within every method in `mts` that updates `t`.

However, invariants may take the form of any predicate formed by Alloy logical operators, which may not be directly translated to a programming language-based coupling invariant. Although human-based analysis is successful in translating invariants, automatic translation is a significant challenge. Table 6.2 shows two examples of invariants involving relation `r`. The first example defines `r` for each `S` object to be **lone** – the modifier means that the result is either empty or contains one element; in this case, the coupling invariant is a disjunction over the cardinality of the resulting set of objects (with the `#` operator). In the second example, `r` is taken from the set of pairs in the transpose of relation `x`, but only pairs that have `Type1` as the right type (in Alloy, `&` denotes set intersection while `->` is cartesian product). The translated coupling is defined as a set comprehension which define **self**`.r` as `Type1` objects whose `x` field contains **self** – the transpose forces invariants over the relation's target, in this case `Type1`.

A possible solution to this problem is a mechanism for systematic translation between Alloy operands to BN counterparts. Semantic equivalence of invariants and coupling is a critical requirement, according to the semantic conformance specification in Chapter 5. The definition of a general form of invariants might ease the translation, as it reduces

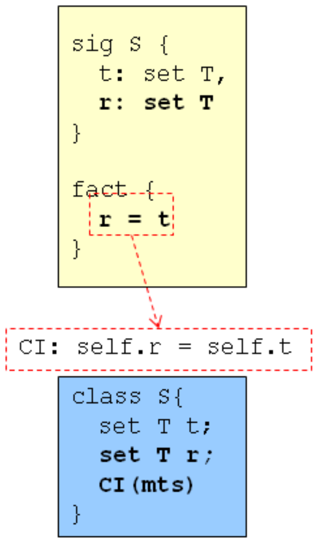


Figure 6.7: Coupling invariant from clone relation

Model invariant	**Coupling Invariant**
all `s:S` | **lone** `s.r`	`(#`**self**`.r=0)` $\vee$ `(#`**self**`.r=1)`
`r = ~x & (SuperType->Type1)`	**self**`.r=`$\{$`s:Type1 |` **self** $\in$ `s.x`$\}$

Table 6.2: Translation of relation definitions to coupling invariants

the scope; translation rules could then be defined for this reduced scope. The rules must also be proved compositional to every invariant, which is beyond the scope of this thesis.

The conformance relationship we use in this thesis establishes that every signature and relation is implemented as class and field, respectively, which makes translation easier. Other aspects of improvement is the addition of extra checking on the left expressions of coupling invariants, in order to avoid errors, such as null pointer issues. For instance, invariant `r=x.y` could be translated to **self**`.x` $\neq$ **null** $\Rightarrow$ **self**`.r = `**self**`.x.y`, which will affect the result of the simulation in the changed methods.

Refinement calculation

After establishing a coupling invariant, the methods of the refined class must be modified in order to incorporate the changed fields and satisfy the invariant. This is done in BN by simulation, which enforces the following properties [8]:

- The coupling invariant holds initially (once the constructor has been executed);
- The two versions of every method are executed from related states, and the results are also related by the coupling invariant.

For **self**`.r = `**self**`.t`, it is straightforward to enforce simulation: every writing to **self**`.r` is augmented with the corresponding update **self**`.t:= `**self**`.r`. Yet, coupling invariants like **self**.r={s:Type1 | **self** ∈ s.x} demand human knowledge for carrying out simulation. This particular case demands a refinement that also involves changes to class `Type1` – changes to field `x` must be introduced. Automation in these arbitrary cases is far from trivial.

6.5.2 Quality of refactorings

According to Opdyke, who coined the term, program refactoring is "*the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*" [82]. The last sentence is critical to the definition, meaning that some quality factors of the source code are improved after refactoring. Nevertheless, strategies, as defined in this thesis, in some circumstances present deficiencies concerning quality, which are not captured by the proposed automation of strategies.

Recalling the file system refactoring from Section 6.4, the outcome of the automatic strategy applications is a program with the following characteristics:

- The program is in syntactic and semantic conformance with the refactored model;
- The program preserves the behavior of the original program;
- The program presents confinement for a subset of the class table.

These characteristics are the main goal of strategies. Nevertheless, quality factors such as cohesion and legibility still requires some improvement in the resulting design. In the file system example, the cohesion property in the `Dir` class could be increased by moving the methods referring to `contents` – `getContents` and `setContents` – to `Dir`. As a consequence, the casts and tests related to **self** could be eliminated, since these methods would be declared in `Dir`. The following fragment depicts the resulting `Dir` class.

```
class Dir extends FSObject{
  pri set Dir contents; ..
  set Dir getContents() {
    result:= self.contents
  }
  unit setContents(set Dir c) {
    self.contents:= c
  }
}
```

The transformations demanded by these improvements are formalized as laws of programming from the catalog listed in Appendix B, applied in the following sequence:

1. Law 27 (L-R) makes the private field to be public first, with no further changes in the program;
2. `contents` now is a public field, thus with Law 36 (R-L) methods `setContents` and `getContents` are moved to `Dir`, which is valid as all accesses to the field are made with a `Dir` left expression, and no access to private fields is made within the methods;

3. Law 41 (L-R) allows making the type test **true** to the **if** statement, which can then be removed by Law 19 (L-R);

4. Finally, type casts on `contents` accesses are removed by using Law 40 (R-L).

Although theoretically feasible, these law applications could not be automatically applied in the example, since our initial assumption is that each strategy is recorded and independently applied in order, disregarding the enclosing refactoring (*Push Down Relation*) that was applied to the model.

Nevertheless, we see *user feedback* as a possible answer to this challenge, in addition to *complementary strategies*. In this case, the application of the whole model refactoring could bring additional information that is then applied in the program refactoring, according to feedback from the user of a supporting tool. If the user agrees, a complementary strategy, containing the five indicated law applications, is automatically applied. Figure 6.8 depicts this example; it is important to notice that the outcome of the complementary strategy is an improved program, yet still conforming, syntactically and semantically, to the refactored model. This additional strategy is conditional to the employment of the whole model refactoring, not linked to any of the isolated strategies. This solution could help tackling the quality issue with strategies, mainly in cases that successive application of strategies could decline quality (as seen in the case studies presented in Chapter 8).

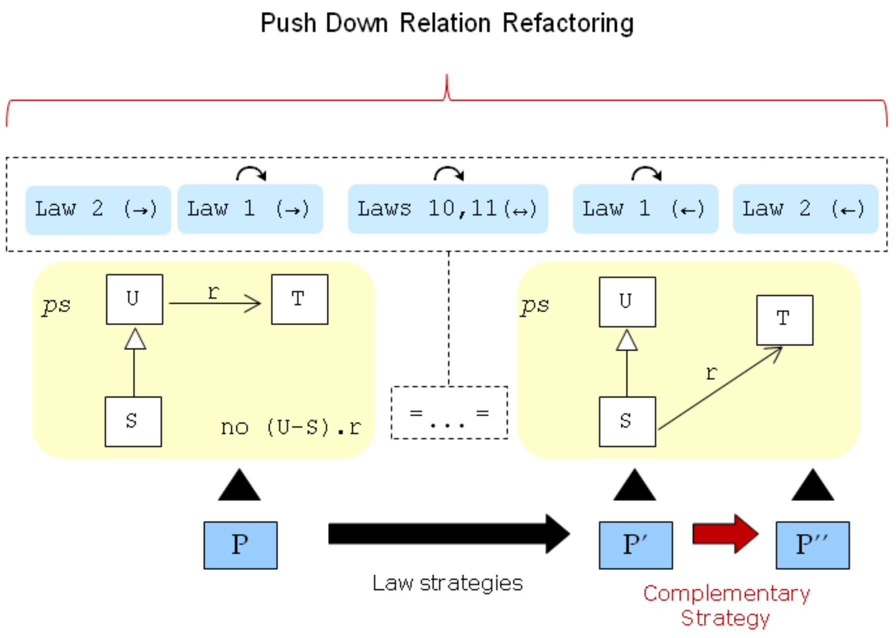


Figure 6.8: Complementary strategy example

From other strategies, we can mention a few other examples that could benefit from user interaction and complementary strategies. For instance, renaming operations in strategies can prompt the user to provide the desired name, instead of a default primed name. Also, strategies whose outcome contains public fields, which may be undesirable for encapsulation and representation independence, may prompt users to *encapsulate* *the field* [37] – making the field private and creating its getter and setter methods. This action would invoke a complementary strategy using Refactoring 4, which is based on laws of programming [26]. Finally, a representative example of quality issue can be seen in strategy *removeField*, which often results in programs with duplicate assignments, of the form **self**`.x:= exp;` **self**`.x:= exp`, as one of the statements is a replacement from the removed field, using the relation definition; in this case, `x` is replacing another field `y`, whose definition was `y = x`.

6.6 Conformance Relationship and Strategies

The required conformance relationship for our approach was established during strategy definition. In this work, several choices were considered, having impact on the conformance relationship used as precondition. This scenario allowed us to gather evidences on how the chosen conformance affects the final results of model-driven program refactorings.

In this context, the more abstract are the models, the looser (different possible implementations for the same object model) the syntactic conformance relationship is. The required syntactic mappings between model and program declarations drive the freedom of implementation for modeled signatures and relations. At the end, we adopted a tighter conformance relationship than initially expected, probably due to inexperience in conformance: signatures must be implemented as classes and relations as fields in the corresponding class. Nevertheless, the required conformance relationship still preserves some abstraction: methods and additional classes can be freely implemented, and hierarchies can contain more classes then modeled. In addition, we concluded that *for model refactoring to be useful, the main declarations must be maintained.* As refactoring is a structural modification, the declarations in the model must be reflected in the source code for desired transformation; otherwise, the task would be rather pointless.

The main conclusion from this investigation: the looser is the syntactic conformance, the more complex the program transformations needed to refactor the program become. When giving more freedom of implementation to a specific model declaration, strategies must consider every implementation option for this declaration, in order to achieve automation. In this context, strategies must be more elaborate, which, based on our experience, *often clutters the program, decreasing quality.* Our main goal is to apply our approach to the highest possible abstraction level, while still allowing interesting and reasonable refactorings, that will not depreciate program quality.

In order to justify these conclusions, we present a few examples from our investigations. First, the Alloy **extends** clause could be implemented at least in two different ways: regular **extends** between the corresponding classes or a content-based implementation, with a `type` field in the superclass. The first option is straightforward, thus chosen as the single alternative in our syntactic conformance; the second option would harden the strategy task, mainly when model transformations require changes to the subsignature.

Another example related to inheritance is the flexibility for implementing signature hierarchies. Our first option was to allow freedom of additional classes in a hierarchy of modeled classes (the program's hierarchy may be larger). However, during our work with strategy *introduceSubclass*, problems were detected with this choice, as it would be much harder to maintain the desired invariant, consequently the semantic conformance with the refactored model. As described in Section 6.3.2, the refactored model includes an invariant over the newly-introduced subsignature `X` – `X=U-S-T` –, which takes values previously assigned exclusively to the superclass `U`. In the program, non-modeled intermediate classes can contribute with instances to the invariant, so they have to be made abstract. Making these classes abstract involves creating auxiliary subclasses to each one, replacing the corresponding **new** statements, which would involve many additional changes to the program with no direct relation to the desired refactoring. Our choice: intermediate classes are allowed between modeled classes, although they must already be abstract, as depicted in Figure 6.9. Even though this might be restrictive, these intermediate classes are implementation details, and commonly these classes are only used to better organize the hierarchy. This constraint is represented by the `abstractConstraint()` predicate from the syntactic conformance definition in Section 5.2.

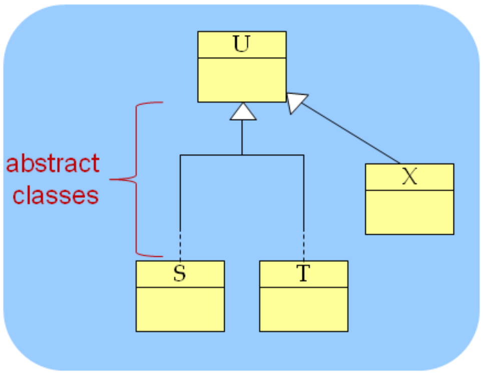


Figure 6.9: Constraint on classes in hierarchies between modeled classes

A third example is illustrated by our choice of implementation for relations, as set or single fields. A simple and reasonable choice was to directly reflect the modeled decisions: unconstrained (**set**) and more-than-one (**some**) relations must be implemented by **set** fields; in contrast, zero-or-one (**lone**) and one (**one**) relations are implemented as single fields. Freedom of implementation for this case would make strategies much more complex, as for each case both options would be considered, without a relevant benefit for abstraction.

Figure 6.10 shows the two extreme points that we see as likely conformance relationships between object models and programs, taking into account the relative proximity between Alloy and BN. The highest level of abstraction presents total freedom of implementation, whereas the lowest point represents models that include methods and some form of attached programming logic. The conformance relationship chosen for this thesis aims to be located in the leftmost point in this line that allows interesting refactorings in the quality point of view, as indicated in the figure. Other conformance relationships on this line could be specified with the formal framework for conformance presented in Chapter 5.

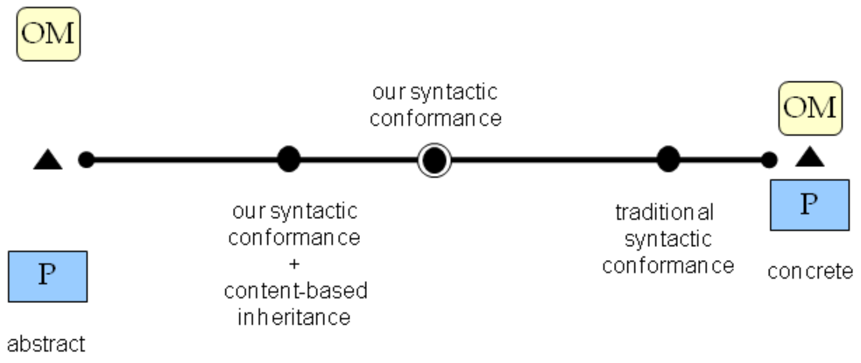

OM
P
abstract
our syntactic conformance + content-based inheritance
our syntactic conformance
traditional syntactic conformance
OM
P
concrete

Figure 6.10: Conformance Relationships

6.7 Chapter summary

The main contribution of this thesis is presented in this chapter: a model-driven approach to formal refactoring, which is based on the correspondence of Alloy law applications to strategies, which define sets of programming law applications. Strategies differ from common program refactoring as it uses the information given by the object model for carrying out more powerful program transformations.

The strategies have been specified as refinement tactics. For each Alloy law that may have impact on program structures we defined two strategies, one for each application direction. In particular, strategy *removeSubclass* induces an additional contribution: the need for eliminating a subclass induced a solution for eliminating type tests that may be used to refine the reduction to a normal form in programming language definition, which is useful for defining a relative completeness of a law catalog [16].

A key aspect of model-driven refactoring is observed in the *introduceField* and *removeField* strategies, which are correspondent to the Alloy law 1 (*introduce relation and its definition*). The automation degree achieved for these strategies is minimized due to the generality of the involved class refinement. We designed strategies amenable to automation for basic cases of relation definition, although we provide a detailed analysis of the issue and possible ways to solve the problem. Furthermore, we discuss several aspects of program refactoring quality, proposing a feasible solution with extended strategies, and establish a clearer relationship between conformance relationship and automation within our investigation with strategies.

Chapter 7

Soundness of Strategies

Formal proofs are the verification method carried out for the results presented in the previous chapter. In this chapter, we establish a soundness theorem for strategies, proving that they are applicable to any conforming pair object model-program. The semi-automation of strategies in our approach depends on a specific syntactic and semantic conformance relationship (Chapter 5) and two additional properties:

- they must express *refinements*;
- they must preserve program *confinement*.

So, for each strategy, these properties configure proof obligations that must be discharged. These proofs are accomplished manually in this thesis. Even though we used the PVS language and tool for type-checking formal definitions, the associate prover has not been applied; from the experience that our research group had with the PVS prover [42], unifying two theories – for Alloy and RN – is beyond the scope of this work. Additionally, the relatively low complexity of most proofs in this work did not justify the adoption cost of the mentioned theorem prover.

Figure 7.1 graphically depicts these obligations for an arbitrary strategy. In Section 7.1 we describe the general theorem, which is broken into lemmas that are proved for every strategy. Also, Section 7.2 presents proofs for two of the representative strategies that we have chosen to prove: *introduceField* and *removeField*, the most applied in the examples throughout this thesis. The other proofs developed for this thesis are detailed in Appendix E.

7.1 Definitions

Our approach to validate strategies is to develop soundness proofs, with which we ensure that the goals for each strategy are fulfilled by their definitions. In fact, along with the case studies presented in Chapter 8, the developed proofs helped us gathering a number of bugs in strategies, also motivating changes to the required conformance relationship. For instance, the abstract hierarchy constraint explained in Chapter 5 was added due to problem found during the proof for strategy *introduceSubclass*.

First we provide the general theorem, proved for each strategy. In this definition, an arbitrary object model is given by OM, and an arbitrary program is P. Refactored

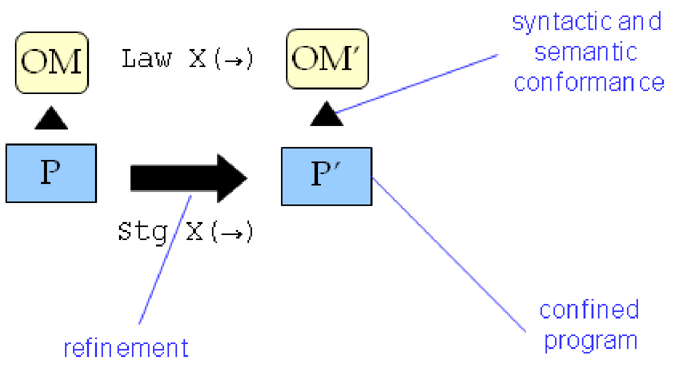


Figure 7.1: Soundness proof obligations

models or programs are indicated with the prime (′) symbol. The following predicates, formally defined in Chapter 5, are applied in the definition and proofs. When we refactor syntax and semantics of programs may change, which may break conformance; therefore we need a theorem that guarantees its preservation in terms of syntax and semantics after the refactoring.

- $syntConformance(OM, P)$ states that P obeys the required syntactic conformance with model OM;
- $semanticConformance(OM, P)$ states that P is in semantic conformance with OM regarding its heaps of interest.

Furthermore, we use predicates that are straightforward from law definitions: *Refines(OM',OM)* or *Refines(P',P)*, in which the second argument refines the first, and *Confined(P)*, stating that P satisfies the adopted static analysis confinement rules [8] for a subset *Own* of the class table. Several additional functions and predicates from Chapter 5 are used to back up definitions in this chapter; for instance, $sigs(OM)$ yields the set of signatures declares in OM. We explain these as they appear in definitions.

7.1.1 General Theorem

Based on the general format of model-driven refactoring with strategies, we formalize a soundness proof for strategies using a *premise* that form the basis for proving the obligations showed in Figure 7.1. For each of these obligations a *supporting lemma* is defined and proved. Theorem 7.1 is generally defined for an arbitrary strategy. The theorem's meta-variables OM, OM', P and P' are specified in each of the strategies for its proof in Section 7.2. OM' is the result of the respective Alloy law application to OM; similarly, P' results from the execution of the corresponding strategy to P.

Theorem 7.1. $\forall\ OM, OM', P, P' \bullet$
$$\begin{aligned}
&syntConformance(OM, P) \ \wedge\ Confined(P) \ \wedge \\
&\quad Refines(OM', OM) \ \wedge\ semanticConformance(OM, P) \ \Rightarrow \\
&syntConformance(OM', P') \ \wedge\ Confined(P') \ \wedge \\
&\quad Refines(P', P) \ \wedge\ semanticConformance(OM', P')
\end{aligned}$$

In general terms, the premise states that the refactored model OM' refines OM, which is the case as OM' is the result of an Alloy law application to OM [42]. Also, the initial object model and program are assumed to be in syntactic and semantic conformance, due to the precondition for strategy application. Finally, the initial program is assumed to be confined, not allowing internal representation leaks from several classes in the program.

Given the premise is valid, four lemmas must be proved for the soundness of strategies. First, program manipulation in the strategy must result in a program that is in syntactic conformance with the refactored model as indicated in the Alloy law. Second, the refactored program is still confined for the same subset of the class table. Third, the strategy refines the program subject to refactoring. The final predicate determines that the refactored program is in semantic conformance with the refactored model.

In order to facilitate explanation and proof, we split this theorem in four lemmas regarding each of the expected conclusions. First, we reduce the premise to the following predicate:

$$\begin{aligned}&\mathit{premise}(OM,\ \ OM',\ \ P) = \mathit{syntConformance}(OM, P)\ \ \wedge\ \ \mathit{Confined}(P)\ \ \wedge\\ &\qquad \mathit{Refines}(OM', OM)\ \ \wedge\ \ \mathit{semanticConformance}(OM, P)\end{aligned}$$

The four lemmas are enunciated in the following sections, by isolating the conclusions from Theorem 7.1. Afterwards, the general theorem is proved based on the validity of these lemmas.

7.1.2 Syntactic conformance lemma

This lemma states that from the premises, the resulting program maintains the syntactic conformance with the refactored model. Once the lemma is valid, we ensure that model and program declarations match according to the syntactic conformance relationship defined for our approach.

Lemma 7.1. $\forall\, OM, OM', P, P' \bullet \mathit{premise}(OM,\ OM',\ P) \Rightarrow \mathit{syntConformance}(OM', P')$

In order to prove Lemma 7.1 for a given strategy, we adopt the following methodology: $\mathit{syntConf}(OM', P')$ is written in terms of the $\mathit{syntConf}(OM, P)$ predicate, which is taken as premise. Our goal is to make the predicate *true* after applying definitions from Chapter 5 and logic rules of inference.

7.1.3 Confinement lemma

In this lemma, the resulting program preserves confinement, which is taken as valid in the original program. It means that strategies do not break the confinement property, which is used as premise for applying class refinement with reference semantics in the programming language.

Lemma 7.2. $\forall\, OM, OM', P, P'\ \bullet\ \mathit{premise}(OM,\ OM',\ P) \Rightarrow \mathit{Confined}(P')$

This lemma is proved by a *case analysis* over the confinement static analysis rules presented in Chapter 4. For each strategy we define the *Own* and *Rep* sets of classes: the first represents top-level classes that own instances of *Rep* classes, which denote the representation classes of owners. We enumerate those rules for reference in the proof:

1. Public methods declared in *Own* or subclasses cannot return *Rep* types;
2. Methods inherited by *Own* cannot have parameters of *Rep* types;
3. *Rep* classes cannot inherit any methods from non-*Rep* superclasses;
4. For any field access $e.f$, if e is of type *Own*, it cannot access fields of type *Rep*, unless e is **self**;
5. For assignments `x:=` **new** `B` in *Client*, *B* cannot be *Rep* or any of its subclasses;
6. For method calls `x:= e.m(`$\bar{e}$`)`, if e is a *Client* object, and the call is within *Own* or *Rep* (or subclasses), m cannot have *Rep* parameters;

7.1.4 Refinement lemma

The refinement lemma establishes a critical property of refactoring: the original behavior of the program must be preserved.

Lemma 7.3. $\forall\, OM, OM', P, P' \;\bullet\; premise(OM,\ OM',\ P) \Rightarrow Refines(P', P)$

The proof for this lemma is based on showing application of laws of programming; behavior preservation is given by construction, in the same way as previous work on refactoring [26]. In this case, we specify the laws used in the development of each strategy, along with justifications on how their provisos are fulfilled by the program at hand – we reference the catalog presented in Appendix B.

In addition, class refinement is commonly applied in strategies for changing class and hierarchy declarations. In this case, the proof is not based on laws, as the law based on class refinement is specific for copy semantics. Therefore our proof is founded on a refinement theorem presented by Banerjee and Naumann [8]; they enunciate and prove an *abstraction theorem* for the BN language: if the methods of a given confined class module (*Own*, which may include several classes or a hierarchy) have the simulation property, so do all methods of all classes, which entails representation independence. In our proof, we establish a coupling invariant for two versions of this confined class module, proving the simulation for the constructor and an arbitrary method of the main class.

7.1.5 Semantic conformance lemma

This lemma states that from the premises semantic conformance is maintained, but now with the refactored model.

Lemma 7.4. $\forall\, OM, OM', P, P' \;\bullet\; premise(OM,\ OM',\ P) \;\Rightarrow\; semanticConformance\,(OM',\ P')$

This lemma is the most complex to prove. The proof methodology is similar to Lemma 7.1: from premise *semanticConformance(OM,P)*, we proof predicate *semanticConformance(OM',P')*; several auxiliary lemmas are used in the proof, which are enunciated and proved before the main proof. In special, we apply several definitions from Alloy's semantics as formalized in Gheyi's work [42]. For instance, the semantics of an Alloy model *OM* is given by the following definition. It is constituted by a set of interpretations that satisfy two properties: implicit (*satisfyImpInvs(OM,i)*) and explicit (*satisfyExpInvs(OM,i)*) invariants. While explicit invariants consist in invariants packed in fact paragraphs, implicit invariants are given by signature and relation declarations.

$$semantics(OM) = \{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge satisfyExpInvs(OM, i)\}$$

The only implicit invariant that we consider is the one that comes with the **extends** clause, in which the all instances of a subsignature are contained in the supersignature, and subsignatures of the same directly extended signature are disjunct (as defined in Section 5.3). Also, we always consider well-formed Alloy models.

7.1.6 Proof for the general theorem

Assuming that the lemmas are proved for each strategy, we developed a proof for the general theorem. This result is then considered for every strategy.

Proof. We start with the predicate that defines Theorem 7.1 and, with inference rules, reduces it to true.

$$\begin{aligned}
&\quad \forall\, OM, OM, P, P' \bullet premises(OM, OM', P') \Rightarrow \\
&\quad syntConformance(OM', P') \wedge Confined(P') \wedge Refines(P', P) \wedge \\
&\quad semanticConformance(OM', P') \\
&= [\text{choosing arbitrary } OM, OM, P, P'] \\
&\quad premises(OM, OM', P') \Rightarrow \\
&\quad syntConformance(OM', P') \wedge Confined(P') \wedge Refines(P', P) \wedge \\
&semanticConformance(OM', P') \\
&= [\text{definition of } \Rightarrow] \\
&\quad \neg premises(OM, OM', P) \vee \\
&\quad (syntConformance(OM', P') \wedge Confined(P') \wedge Refines(P', P) \wedge \\
&\quad semanticConformance(OM', P')) \\
&= [\text{distribution on } \vee] \\
&\quad (\neg premises(OM, OM', P) \vee syntConformance(OM', P')) \wedge \\
&\quad (\neg premises(OM, OM', P) \vee Confined(P')) \wedge \\
&\quad (\neg premises(OM, OM', P) \vee Refines(P', P)) \wedge \\
&\quad (\neg premises(OM, OM', P) \vee semanticConformance(OM', P')) \\
&= [\text{definition of } \Rightarrow] \\
&\quad (premises(OM, OM', P) \Rightarrow syntConformance(OM', P')) \wedge \\
&\quad (premises(OM, OM', P) \Rightarrow Confined(P')) \wedge \\
&\quad (premises(OM, OM', P) \Rightarrow Refines(P', P)) \wedge \\
&\quad (premises(OM, OM', P) \Rightarrow semanticConformance(OM', P'))
\end{aligned}$$

= [from the four lemmas proven for each strategy]
  true

7.2 Proofs

The following strategies were proved, and their proofs are representative for the other strategies not proved (as showed in parentheses).

1. introduceClass (removeClass);
2. introduceSuperclass;
3. removeSuperclass;
4. introduceSubclass;
5. removeSubclass;
6. introduceField – split (empty field, clone field);
7. removeField – split (empty field, clone field);
8. fromSetToOptionalField (optional to set, set to single, single to set);
9. splitField (Remove Indirect Reference).

The *removeClass* is not proved due to its similarity to *introduceClass*, which is not seen in strategies *introduceSuperclass,removeSuperclass,introduceSubclass* and *removeSubclass*; in these cases, we prove strategies for both directions of the corresponding laws of modeling. For introducing and removing fields, we only prove one of the three cases that we chose to be automated by strategies (composition of other two fields). Changing the qualifier of a field from **set** to optional was chosen for the proof, and the other related strategies present a very similar proof. Also, *splitField* is proved, in representation for Law 16.

In this chapter, we present and explain two of these proofs: *introduceField* and *removeField*, since they are the most used strategies in examples and case studies in this thesis. The other proofs are detailed in Appendix E.

7.2.1 introduceField

In this strategy, a new field is introduced with values from a composition of other two fields. We define the meta-variables OM, OM', P, P' for this strategy as templates, similarly to the notation used for laws of modeling and programming. Expressions in the templates are defined in the **where** clause; in this clause, we make extensive use of substitutions in the form $e' = e[x/y]$, which represents e' as taking e, but replacing all occurrences of y by x. Program templates are taken from the strategy definition. For simplicity, **[extends]** symbolizes any signature or class, being possibly empty.

Let OM, OM' be any two object models and P, P' two programs as follows:

OM

```
ps
sig S [extends]{
   rs
   x : set U
}
sig U [extends]{
   rs'
   y : set T
}
fact F{
   forms
}
```

$\Longrightarrow$

OM′

```
ps
sig S [extends]{
   rs
   x : set U,
   r : set T
}
sig U [extends]{
   rs'
   y : set T
}
fact F{
   forms
   r = x.y
}
```

P

```
CT
class S [extends]{
   fds; X r;
   set U x;
   mts
}
class U [extends]{
   fdsU; set T y;
   mtsU
}
```

$\Longrightarrow$

P′

```
CT
class S [extends]{
   fds; X r';
   set U x;
   set T r;
   mts'
}
class U [extends]{
   fdsU; set T y;
   mtsU
}
```

where:

$mts''' = mts[\textbf{self}.r'/\textbf{self}.r]$

$mts'' = mts'''[\textbf{unpack self};\ \textbf{self}.x.y := exp;\ \textbf{self}.r := exp;\ \textbf{pack self}/\textbf{self}.x.y := exp]$

$mts' = mts''[\textbf{unpack self};\ \textbf{self}.x := exp;\ \textbf{if}\ (\textbf{self}.x = \varnothing)\ \textbf{then self}.r := \varnothing\ \textbf{else self}.r := \textbf{self}.x.y;\ \textbf{pack self})/\textbf{self}.x := exp]$

Assuming *premise*$(OM,\ OM',\ P)$, we now prove the four lemmas enunciated in Section 7.1, following the described methodology. We intercalate proof steps, formal justifications and textual explanations.

Syntactic conformance

Proof. The goal is to prove validity of the predicate, based on the premise predicates, mainly *syntConf*(OM, P). We begin by expanding the lemma's conclusion predicate, from its definition in the required conformance relationship (every signature and relation is accordingly mapped in the program, and the abstract hierarchy constraint is fulfilled).

$syntConf(OM', P')$

$= $ [definition]
$\forall s : sigs(OM') \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM') \bullet relationMapping(r, P') \wedge$
$abstractConstraint(OM', P')$

Replacements are then made by comparing signatures and relations before and after the law application. In this case, signatures are not affected, and an additional relation is declared in the refactored model ($r : \textbf{set}\ T$). Later for signatures no effects are seen, so the predicate takes the form of the premise.

$=$ [from definitions of OM, OM',$sigs(OM') = sigs(OM)$ and $rels(OM') = rels(OM) \cup \{r : \textbf{set}\ T\}$]
$\forall s : sigs(OM) \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM) \cup \{r : \textbf{set}\ T\} \bullet relationMapping(r, P') \wedge$
$abstractConstraint(OM', P')$

$=$ [from definitions of $OM', P, P', \forall r : sigs(OM) \bullet\ sigMapping(s, P) = sigMapping(s, P')$]
$\forall s : sigs(OM) \bullet sigMapping(s, P) \wedge$
$\forall r : rels(OM) \cup \{r : \textbf{set}\ T\} \bullet relationMapping(r, P') \wedge$
$abstractConstraint(OM', P')$

Since P' maintains the same classes from P, no class hierarchy is changed. Then from the premise, *abstractConstraint* is still valid for OM' and P'. Also from the premise, the quantification over signatures is valid as well.

$=$ [from definitions of OM', P', $abstractConstraint(OM', P') = abstractConstraint(OM, P)$, valid from the premise]
$\forall s : sigs(OM) \bullet sigMapping(s, P) \wedge$
$\forall r : rels(OM) \cup \{r : \textbf{set}\ T\} \bullet relationMapping(r, P')$

$=$ [from definition of $syntConf(OM, P)$, and premise, $\forall s : sigs(OM) \bullet sigMapping(s, P)$ is valid]
$\forall r : rels(OM) \cup \{r : \textbf{set}\ T\} \bullet relationMapping(r, P')$

Now we concentrate efforts on the predicate concerning relations. From set theory, we separate the united parts of the quantified set into two conjunctions, which allow us to deal with the newly-introduced relation apart from the existing relations. Regarding the latter, nothing changes, so the premise is applied.

$=$ [set theory]
$\forall r : rels(OM) \bullet relationMapping(r, P') \ \wedge\ relationMapping(r : \textbf{set}\ T), P')$

$=$ [from $premise(OM, OM', P)$,$\forall r : rels(OM) \bullet\ r \neq (r : \textbf{set}\ T) \Rightarrow relationMapping(r, P') \Leftrightarrow relationMapping(r, P)$, which is valid]

$relationMapping(r : \mathbf{set}\ T, P')$

For the new relation, we expand the definition of the *relationMapping* predicate to prove that `r` is properly implemented as a set field in the program. This is done by giving the `r` field as an instance for the existential quantifier.

= [definition of *relationMapping*, simplified]
$\exists f : fields(P') \bullet (r : \mathbf{set}\ T).name = f.name \wedge$
$(r : \mathbf{set}\ T).leftType = f.leftType \ \wedge\ (r : \mathbf{set}\ T).rightType = type2(f) \wedge$
$(\neg isScalarR(r : \mathbf{set}\ T) \Rightarrow \neg isScalarF(f))$

= [predicate calculus, choose newly-introduced field r in P']
$(r : \mathbf{set}\ T).name = r.name \wedge$
$(r : \mathbf{set}\ T).leftType = type1(r) \ \wedge\ (r : \mathbf{set}\ T).rightType = r.rightType \wedge$
$(\neg isScalarR(r : \mathbf{set}\ T) \Rightarrow \neg isScalarF(r))$

= [from definition of P', names and types exactly match]
$(\neg isScalarR(r : \mathbf{set}\ T) \Rightarrow \neg isScalarF(r))$

= [from definitions of OM', P', neither field nor relation are scalar]
true □

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement. For each rule, we justify its maintenance in terms of the premise and P'. In this case, S, U $\in$ *Own* and U, $T \in$ *Rep*.

1. No method interface is changed, thus from $premise(OM,\ OM',\ P)$ there are no methods in *Own* with *Rep* return types;
2. No inherited methods are added, thus from $premise(OM,\ OM',\ P)$ no inherited methods have *Rep* parameters.
3. Same as above, thus from $premise(OM,\ OM',\ P)$, *Rep* classes do not inherit methods from non-*Rep* classes;
4. No public fields of *Own* classes are used outside their declaring module, thus from $premise(OM,\ OM',\ P)$ no `e.f` is seen, `e` $\leq$ *Own*, unless `e` is **self**;
5. No **new** is changed, thus from $premise(OM,\ OM',\ P)$ *Rep* instance is created outside *Own* classes;
6. No method call is affected, thus from $premise(OM,\ OM',\ P)$, `e.m` calls within *Own* or *Rep* do not have *Rep* parameters.

□

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. If `r` is already declared in class `S`, we rename it to `r'`, which is a straightforward equivalence;
2. Law 28 *Field Elimination* (R-L) is applied, whose provisos are valid:
   (a) `r` is not declared in `S` or in its super or subclasses in CT.
3. Refactoring 4 *Self-Encapsulate Field* (L-R) is applied, whose provisos are valid:
   (a) `getY` or `setY` are not declared in super or subclasses in CT.
4. From the abstraction theorem for BN [8], refinement is given by a simulation proof. In this case, the coupling invariant is $\textbf{self}.r = \textbf{self}.x.y$:
   - Constructors in `S` force changes to field `x` and combination `x.y` to be duplicate for field `r`, so the invariant is established;
   - Same for methods, maintaining the invariant.

□

Semantic conformance

Finally, we prove semantic conformance, with the help of a few auxiliary lemmas. Lemma 7.5 allows us to introduce any name and a set of values to this name to an interpretation, and it still will be part of the model's semantics. Next, Lemma 7.6 establishes a definition of $semantics(OM')$ in terms of $semantics(OM)$, helping the proof for the main lemma. Similarly, Lemma 7.7 establishes that heaps for P' can be defined in terms of analogous heaps for P.

Lemma 7.5. Any interpretation can be augmented with mappings to the `r` relation, assuming `r` is a new name.

$$\begin{aligned}&\forall\, OM : Model, r : Relation, i : Interpretation, v : \mathbb{P}\, Value \bullet\\&\quad \neg(r.name \subseteq relNames(OM)) \wedge i \in semantics(OM) \Rightarrow\\&\qquad \{i \oplus r.name \mapsto v\} \in semantics(OM)\end{aligned}$$

Proof. The addition to any new mapping to an interpretation does not change its condition as part of the model's semantics, since interpretations are only checked over the mappings for model names.

Lemma 7.6. The semantics of OM' can be defined in terms of $semantics(OM)$ as follows. In this mathematical domain, $\oplus$ and $\fatsemi$ denote relational overriding and relation composition, respectively.

$$\begin{aligned}&\mathit{semantics}(\mathit{OM}') = \{i \oplus (r : \mathbf{set}\ T).\mathit{name} \mapsto \\ &\quad i.\mathit{mapRel}(x : \mathbf{set}\ U) \fatsemi i.\mathit{mapRel}(y : \mathbf{set}\ T) \mid i \in \mathit{semantics}(\mathit{OM})\}\end{aligned}$$

Proof. We start from the definition of $\mathit{semantics}(\mathit{OM}')$. In this strategy, implicit invariants are not affected. The most important substitution is related to value for the newly-introduced relation `r`, defined as the composition of relations `x` and `y`, from Lemma 7.5.

$\mathit{semantics}(\mathit{OM}')$
= [definition]
$\quad \{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(\mathit{OM}', i) \land \mathit{satisfyExpInvs}(\mathit{OM}', i)\}$

= [when introducing r, no **extends** clause is affected, thus $\mathit{satisfyImpInvs}(\mathit{OM}, i) = \mathit{satisfyImpInvs}(\mathit{OM}', i)$]
$\quad \{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(\mathit{OM}, i) \land \mathit{satisfyExpInvs}(\mathit{OM}', i)\}$

= [from definition of $\mathit{satisfyExpInvs}(\mathit{OM}', i)$]
$\quad \{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(\mathit{OM}, i) \land \forall f : \mathit{factInvs}(\mathit{OM}') \bullet$
$\qquad \mathit{satisfyFormula}(f, i)\}$

= [from definitions of OM, OM',$\mathit{factInvs}(\mathit{OM}') = \mathit{factInvs}(\mathit{OM}) \cup (r = x.y)$]
$\quad \{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(\mathit{OM}, i) \land$
$\qquad \forall f : \mathit{factInvs}(\mathit{OM}) \cup \{r = x.y\} \bullet \mathit{satisfyFormula}(f, i)\}$

= [set theory]
$\quad \{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(\mathit{OM}, i) \land$
$\qquad \forall f : \mathit{factInvs}(\mathit{OM}) \bullet \mathit{satisfyFormula}(f, i) \land$
$\qquad \mathit{satisfyFormula}((r = x.y), i)\}$

= [from definition of $\mathit{semantics}(\mathit{OM})$]
$\quad \{i : \mathit{Interpretation} \mid i \in \mathit{semantics}(\mathit{OM}) \ \land\ \mathit{satisfyFormula}((r = x.y), i)\}$

= [from Lemma 7.5, where $v = i.\mathit{mapRel}(x : \mathbf{set}\ U) \fatsemi i.\mathit{mapRel}(x : \mathbf{set}\ U)$]
$\quad \{i \oplus (r : \mathbf{set}\ T).\mathit{name} \mapsto$
$\qquad i.\mathit{mapRel}(x : \mathbf{set}\ U) \fatsemi i.\mathit{mapRel}(y : \mathbf{set}\ T) \mid i \in \mathit{semantics}(\mathit{OM})\}$

Lemma 7.7. For P and P':

$$\begin{aligned}&\mathit{heaps}(P', \mathit{filter}) = \{h \oplus (\mathbf{set}\ T\ r).\mathit{name} \mapsto \\ &\quad h.\mathit{mapField}((\mathbf{set}\ U\ x).\mathit{name}) \fatsemi h.\mathit{mapField}((\mathbf{set}\ T\ y).\mathit{name}) \mid h \in \mathit{heaps}(P, \mathit{filter})\}\end{aligned}$$

Proof. The changed commands do not add or remove heaps (as they are always included within guarded blocks); all `S` instances include a new field, whose values are always equal to the composition of sets `x.y` as dereferenced by each `S` instance.

Main proof. Lemma 7.4 is then proved, using the result of the three auxiliary lemmas and the premise.

$semanticConformance(OM', P')$
$= $ [definition]
$\quad \forall\, h : heaps(P', filter) \bullet$
$\quad\ \exists\ i : semantics(OM') \bullet$
$\quad\quad \forall\, s : sigs(OM') \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, r : rels(OM') \bullet\ i.mapRel(r.name) = h.mapField(r.name)$

Similarly to Lemma 7.1, $sigs(OM')$ and $rels(OM')$ are replaced accordingly. An additional relation is declared in the refactored model ($r : \textbf{set}\ T$). Later, set theory is applied for relation mappings, isolating the mapping for `r`.

$=$ [from definitions of OM, OM',$sigs(OM') = sigs(OM)$ and $rels(OM') = rels(OM) \cup \{r : \textbf{set}\ T\}$]
$\quad \forall\, h : heaps(P', filter) \bullet$
$\quad\ \exists\ i : semantics(OM') \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, r : rels(OM) \cup \{r : \textbf{set}\ T\} \bullet\ i.mapRel(r.name) = h.mapField(r.name)$

$=$ [set theory]
$\quad \forall\, h : heaps(P', filter) \bullet$
$\quad\ \exists\ i : semantics(OM') \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)\ \wedge$
$\quad\quad i.mapRel(r : \textbf{set}\ T).name) = h.mapField((r : \textbf{set}\ T).name)$

With Lemma 7.6 we can rewrite the refactored model's semantic definition with the definition for the original model. Further, Lemma 7.7 helps rewriting the set of heaps from the program semantics.

$=$ [Lemma 7.6]
$\quad \forall\, h : heaps(P', filter) \bullet$
$\quad\ \exists\ i : \{i \oplus (r : \textbf{set}\ T).name \mapsto$
$\quad\ i.mapRel(x : \textbf{set}\ U) \mathbin{⨾} i.mapRel(y : \textbf{set}\ T) \mid i \in semantics(OM)\} \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)\ \wedge$
$\quad\quad i.mapRel((r : \textbf{set}\ T).name) = h.mapField((r : \textbf{set}\ T).name)$

$=$ [Lemma 7.7]
$\quad \forall\, h : \{h \oplus (\textbf{set}\ T\ r).name \mapsto h.mapField((\textbf{set}\ U\ x).name) \mathbin{⨾} h.mapField((\textbf{set}\ T\ y).name) \mid$
$h \in heaps(P, filter)\} \bullet$
$\quad\ \exists\ i : \{i \oplus (r : \textbf{set}\ T).name \mapsto$
$\quad\ i.mapRel(x : \textbf{set}\ U) \mathbin{⨾} i.mapRel(y : \textbf{set}\ T) \mid i \in semantics(OM)\} \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)\ \wedge$
$\quad\quad i.mapRel((r : \textbf{set}\ T).name) = h.mapField((r : \textbf{set}\ T).name)$

From the program heaps, we use $h1$ as an arbitrary heap, eliminating the universal quantifier. Next, we use the specific interpretation $i1$ to skolemize the predicate and eliminate the existential quantifier. The chosen $i1$ has mappings to all names in OM, in addition to the specific mapping to `r` as a composition of `x` and `y`.

= [from predicate calculus, choosing an arbitrary $h1$]
$\exists\ i : \{i \oplus (r : \mathbf{set}\ T).name \mapsto$
$i.mapRel(x : \mathbf{set}\ U) \fatsemi i.mapRel(y : \mathbf{set}\ T) \mid i \in semantics(OM)\} \bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h1.mapField(r.name) \wedge$
$i.mapRel((r : \mathbf{set}\ T).name) = h1.mapField((r : \mathbf{set}\ T).name)$

= [for existential quantification, we choose interpretation $i1$ which repeats the mappings for names in $h1$, adding the mapping to relation r as indicated]
$\forall s : sigs(OM) \bullet\ i1.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\forall r : rels(OM) \bullet\ i1.mapRel(r.name) = h1.mapField(r.name) \wedge$
$i1.mapRel((r : \mathbf{set}\ T).name) = h1.mapField((r : \mathbf{set}\ T).name)$

= [from $premise(OM, OM', P)$]
$i1.mapRel((r : \mathbf{set}\ T).name) = h1.mapField((r : \mathbf{set}\ T).name)$

= [from definitions of $i1$ and $h1$]
true

□

7.2.2 removeField

In this strategy, a specific field is removed, based on an invariant established in the model – in this case, the field is a composition of two other fields.

Let OM, OM' be any two object models and P, P' two programs as follows:

OM		OM'

ps
sig S [**extends**]{
  rs
  $r :$ **set** T
}
fact F{
  forms
  $r = x.y$
}
	$\Longrightarrow$	*ps*
		sig S [**extends**]{
		  rs
		}
		fact F{
		  forms
		}

P

```
CT
class S [extends]{
   fds; set U x;
   set T r;
   mts
}
class U [extends]{
   fdsU; set T y;
   mtsU
}
```

$\Longrightarrow$

P'

```
CT
class S [extends]{
   fds; set U x;
   mts'
}
class U [extends]{
   fdsU; set T y;
   mtsU
}
```

where:

$mts' = mts[\textbf{self}.x.setY(exp)/\textbf{self}.r := exp]$
$mts' = mts[exp[\textbf{self}.x.getY()]/exp[\textbf{self}.r]]$

Notice that the program does not consider accesses to `r` outside `S`, due to the confinement assumption. We now prove the four lemmas enunciated in Section 7.1, based on $premise(OM,\ OM',\ P)$.

Syntactic conformance

Proof. We begin by expanding the lemma's conclusion predicate, from its definition in the required conformance relationship.

$syntConf(OM', P')$

= [definition]

$\forall s : sigs(OM') \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM') \bullet relationMapping(r, P') \wedge$
$abstractConstraint(OM', P')$

For replacements signatures are not affected, and a relation is not declared in the refactored model (r : **set** T). Next, for signatures no effects are seen, so the predicate takes the form of the premise.

= [from definitions of OM, OM',$sigs(OM') = sigs(OM)$ and $rels(OM') = rels(OM) - \{r : \textbf{set}\ T\}$]

$\forall s : sigs(OM) \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM) - \{r : \textbf{set}\ T\} \bullet relationMapping(r, P') \wedge$
$abstractConstraint(OM', P')$

= [from definitions of OM', P, P', $\forall r : sigs(OM) \bullet sigMapping(s, P) = sigMapping(s, P')$]

$\forall s : sigs(OM) \bullet sigMapping(s, P) \wedge$
$\forall r : rels(OM) - \{r : \textbf{set}\ T\} \bullet relationMapping(r, P') \wedge$
$abstractConstraint(OM', P')$

Also, *abstractConstraint* is valid for OM' and P'. From the premise, the quantification over signatures is valid.

= [from definitions of OM', P', $abstractConstraint(OM', P') = abstractConstraint(OM, P)$, as hierarchies are not affected]
$\forall s : sigs(OM) \bullet sigMapping(s, P) \land$
$\forall r : rels(OM) - \{r : \textbf{set } T\} \bullet relationMapping(r, P')$

= [from definition of $syntConf(OM, P)$, and premise,$\forall s : sigs(OM) \bullet sigMapping(s, P)$ is valid]
$\forall r : rels(OM) - \{r : \textbf{set } T\} \bullet relationMapping(r, P')$

Regarding relations, from set theory we extract an implication from the previous quantification, in order to isolate the removed relation in the predicate.

= [set theory]
$\forall r : rels(OM) \bullet (r \neq (r : \textbf{set } T)) \Rightarrow relationMapping(r, P')$

We then choose an arbitrary relation, which is not `r`. For this arbitrary relation, from the premise it is implemented as a field. The only field removed from P is `r`.

= [predicate calculus, choosing arbitrary $r1$]
$(r1 \neq (r : \textbf{set } T)) \Rightarrow relationMapping(r1, P')$

= [assuming $(r1 \neq (r : \textbf{set } T))$ as a premise]
$relationMapping(r1, P')$

= [from definitions P, P', no other field is removed but r, from $premise(OM,\ OM',\ P)$, $relationMapping(r1, P')$ is valid]
true

□

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement. In this case, $S, U \in Own$ and $U, T \in Rep$.

1. No method interface is changed, thus from $premise(OM,\ OM',\ P)$ there are no methods in *Own* with *Rep* return types;
2. No inherited methods are added, thus from $premise(OM,\ OM',\ P)$ no inherited methods have *Rep* parameters.
3. Same as above, thus from $premise(OM,\ OM',\ P)$, *Rep* classes do not inherit methods from non-*Rep* classes;
4. No public fields are affected, so from $premise(OM,\ OM',\ P)$ no `e.f` is seen, `e` $\leq Own$, unless `e` is **self**;

5. No **new** is changed, thus from *premise*(OM, OM', P) *Rep* instance is created outside *Own* classes;

6. No method call is affected, thus from *premise*(OM, OM', P), `e.m` calls within *Own* or *Rep* do not have *Rep* parameters. The new method calls to `setY()` are within the module, thus it does not break confinement.

□

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. Refactoring 4 *Self-Encapsulate Field* (L-R) is applied, whose provisos are valid:
   (a) `getR` or `setR` are not declared in super or subclasses in CT.

2. From the abstraction theorem for BN [8], refinement is given by a simulation proof. The coupling invariant is also **self**.r = **self**.$x.y$:
   - Constructors in `S` replaces changes to field **self**`.r` with **self**`.x.setY(e)`, and reads with **self**`.x.getY()`, following the invariant;
   - Same for methods, which maintain the invariant.

3. Law 28 *Field Elimination* (L-R) is applied, whose provisos are valid:
   (a) `exp.r` is not used in `S`, as all reads and writings were replaced by equivalent expressions.

□

Semantic conformance

Finally, for semantic conformance, we first prove a few auxiliary lemmas as they are used in the proof. Lemma 7.8 formalizes the relationships between interpretations for the two models. Next, Lemma 7.9 establishes a definition of *semantics*(OM') in terms of *semantics*(OM), helping the proof for the main lemma. Similarly, Lemma 7.10 establishes that heaps for P' can be defined in terms of analogous heaps for P.

Lemma 7.8. Given *factInvs*(OM') = *factInvs*(OM) − {r = *exp*} and *factInvs*(OM') does not contain any formula with `r`, then:

$$\begin{array}{l}\forall\, OM : Model, OM' : Model, r : rels(OM), i : Interpretation \bullet \\ \quad i \in\ semantics(OM) \Rightarrow i \rhd r.name \in\ semantics(OM')\end{array}$$

Proof. If `r` is not defined in any formula except for its definition in the original model, it does not have any influence on the interpretation values mapped to the remaining model names. In this mathematical domain, ⩥ denotes anti-domain restriction for the mapping.

Lemma 7.9. The semantics of OM' can be defined in terms of $semantics(OM)$ as follows.

$$semantics(OM') = \{i : Interpretation \mid i \in semantics(OM) \wedge i \rhd (r : \mathbf{set}\ T).name\}$$

Proof. We start from the definition of $semantics(OM')$.
$semantics(OM')$

$=$ [definition]
$\{i : Interpretation \mid satisfyImpInvs(OM', i) \wedge satisfyExpInvs(OM', i)\}$

$=$ [when removing r, no **extends** clause is affected, thus $satisfyImpInvs(OM, i) = satisfyImpInvs(OM', i)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge satisfyExpInvs(OM', i)\}$

$=$ [from definition of $satisfyExpInvs(OM', i)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge$
$\forall f : factInvs(OM') \bullet satisfyFormula(f, i)\}$

$=$ [from definitions of OM, OM', $factInvs(OM') = factInvs(OM) - \{r = x.y\}$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge$
$\forall f : factInvs(OM) - \{r = x.y\} \bullet satisfyFormula(f, i)\}$

$=$ [set theory]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge$
$\forall f : factInvs(OM) \bullet (f \neq (r = x.y)) \Rightarrow satisfyFormula(f, i)\}$

$=$ [from Lemma 7.8, i is part of the semantics of OM]
$\{i : Interpretation \mid i \in semantics(OM) \wedge i \rhd r.name\}$

□

Lemma 7.10. For P and P':
$heaps(P', filter) = \{h : Heap \mid h \in heaps(P, filter) \wedge h \rhd name(\mathbf{set}\ T\ r)\}$

Proof. The changed commands do not add or remove heaps; field `r` is private, so it was only accessed in *mts*.

□

Main proof. Lemma 7.4 is then proved, using the result of the auxiliary lemmas and premise.

$=$ [definition]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : semantics(OM') \bullet$
$\forall s : sigs(OM') \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\forall t : rels(OM') \bullet\ i.mapRel(name(t)) = h.mapField(name(t))$

$=$ [from definitions of OM, OM',$sigs(OM') = sigs(OM)$ and $rels(OM') = rels(OM) - \{r : \textbf{set}\ T\}$]

$\forall\, h : heaps(P', filter) \bullet$
$\quad \exists\ i : semantics(OM') \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, t : rels(OM) - \{r : \textbf{set}\ T\} \bullet\ i.mapRel(name(t)) = h.mapField(name(t))$

$=$ [set theory]

$\forall\, h : heaps(P', filter) \bullet$
$\quad \exists\ i : semantics(OM') \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, t : rels(OM) \bullet\ (t \neq (r : \textbf{set}\ T)) \Rightarrow i.mapRel(name(t)) = h.mapField(name(t))$

With Lemma 7.9 we can rewrite the refactored model's semantic definition with the definition for the original model. Further, Lemma 7.10 helps rewriting the set of heaps from the program semantics.

$=$ [Lemma 7.9]

$\forall\, h : heaps(P', filter) \bullet$
$\quad \exists\ i : \{i : Interpretation \mid i \in semantics(OM) \wedge i \rhd r.name\} \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, t : rels(OM) \bullet\ (t \neq (r : \textbf{set}\ T)) \Rightarrow i.mapRel(name(t)) = h.mapField(name(t))$

$=$ [Lemma 7.10]

$\forall\, h : \{h : Heap \mid h \in heaps(P, filter) \wedge h \rhd name(\textbf{set}\ T\ r)\} \bullet$
$\quad \exists\ i : \{interpretation \mid i \in semantics(OM) \wedge i \rhd (r : \textbf{set}\ T).name\} \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name)\ \wedge$
$\quad\quad \forall\, t : rels(OM) \bullet\ (t \neq (r : \textbf{set}\ T)) \Rightarrow i.mapRel(name(t)) = h.mapField(name(t))$

$=$ [from predicate calculus, choosing an arbitrary $h1$]

$\exists\ i : \{i : Interpretation \mid i \in semantics(OM) \wedge \rhd (r : \textbf{set}\ T).name\} \bullet$
$\quad \forall\, s : sigs(OM) \bullet\ i.mapSig(s.name) = h1.mapClass(s.name)\ \wedge$
$\quad \forall\, t : rels(OM) \bullet\ (t \neq (r : \textbf{set}\ T)) \Rightarrow i.mapRel(name(t)) = h1.mapField(name(t))$

$=$ [for existential quantification, we choose interpretation $i1$ which repeats the mappings for names in $h1$, removing mappings from $r.name$]

$\forall\, s : sigs(OM) \bullet\ i1.mapSig(s.name) = h1.mapClass(s.name)\ \wedge$
$\forall\, t : rels(OM) \bullet\ (t \neq (r : \textbf{set}\ T)) \Rightarrow i1.mapRel(name(t)) = h1.mapField(name(t))$

$=$ [from $premise(OM,\ OM',\ P)$]

$\forall\, t : rels(OM) \bullet\ (t \neq (r : \textbf{set}\ T)) \Rightarrow i1.mapRel(name(t)) = h1.mapField(name(t))$

$=$ [from definitions of $i1$ and $h1$]

$true$

□

7.3 Chapter summary

The soundness of strategies is given by the proofs developed in this chapter. It begins by providing the general definition of soundness for strategies, followed by the construction of auxiliary lemmas that will make part of the proof. In this chapter, proofs for two strategies are presented in detail. However, we developed proofs for a representative subset of the defined strategies, which are completely developed in Appendix E.

Chapter 8

Case Studies

In this chapter, we present more extensive examples of our approach to model-driven refactoring. An example from Martin Fowler's book on refactoring [37] was used in Section 8.1 as basis for a video store's object model, whose implementation is refactored by strategies. Next, in Section 8.2 we refactored the object model for a simple type-checker based on the Java language [48]. Finally, we compared two versions of the Push Down Relation refactoring presented in Chapter 4 to check whether the resulting programs match the strategies. These case studies revealed a number of issues that strategies must deal with, in particular regarding automation, helping pinpointing problems and evolving the solution.

8.1 Video Store

This case study contemplates a recurring example for refactoring, taken from Fowler's book [37]. We extended Fowler's example, generating the initial object model of the video rental domain. From an initial model, we applied three refactorings to deal better with domain-related elements, such as movie copies and different price options.

This initial object model contains signatures representing customers and movie rentals, as in the mentioned book. We added `Dependant`, that is related to the primary registered customer by relation `main`. Both signatures relate to the rented movies by separate relations, `rentalsC` and `rentalsD`. The `Definitions` fact includes an invariant stating that a given rental instance is never registered to both a customer and one of its dependents, using the transpose operator.

```
sig Customer {
  rentalsC: set Rental
}
sig Dependant {
  main: one Customer,
  rentalsD: set Rental
}
sig Rental {
  movie: one Movie
}
```

```
fact Invariants {
  all r:Rental | one (r.~rentalsC + r.~rentalsD)
}
```

Signature `Movie` represents a DVD owned by the video store; each movie is related to exactly one price code, which encapsulates pricing information.

```
sig Movie {
  code: one PriceCode
}
sig PriceCode { }
```

We provided a conforming BN implementation for this initial model, which is partially showed as follows. All classes contain additional fields. We assume that the `main` method maintains the invariant, in which a customer and its dependents do not share rentals, and the class constructors ensure the modeled multiplicities. We only show the methods that will be affected by the refactorings.

```
class Customer {
  string name;
  string id;
  set Rental rentalsC; ..

  unit addRental(Rental r) {
    self.rentalsC:= self.rentalsC ∪ {r};
  }
}
class Dependant {
  string name;
  Customer main;
  set Rental rentalsD; ..
}
class Rental {
  Movie movie;
  int daysRented; ..
}
class Movie {
  string title;
  PriceCode code; ..
}
class Price {
  float price; ..
}
```

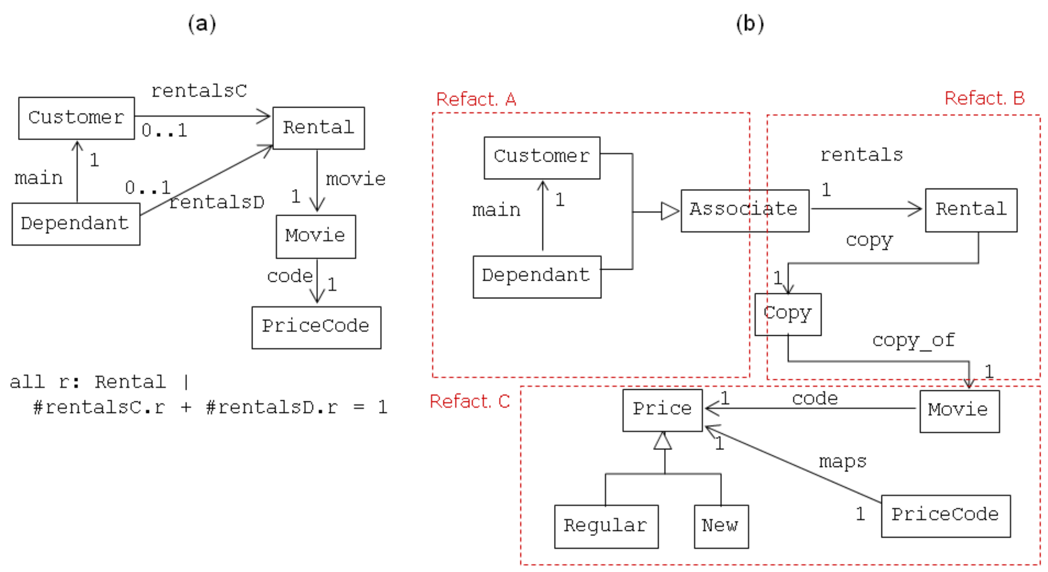
8.1.1 Model refactoring

We prepared and applied three model refactorings, which are composed of Alloy law applications. The refactorings are described as follows, with results highlighted in Figure 8.1(b), compared to the initial model in Figure 8.1(a), represented by UML class diagrams:

A Extract a signature, `Associate`, defining a general structure for individuals who can rent movies. This refactoring is made of generalization introduction and a new relation (`rentals`), removing the original relations `rentalsC` and `rentalsD`;

B Add a new signature – `Copy` – between `Rental` and `Movie`, since more than one copy is available for one movie. For this, we must split the relation `movie` into two new relations, before its removal;

C Restructure the relationship between a movie and its current status – new or regular – based on the State Design Pattern [39]. For this, we must introduce `New` and `Regular`, with the `Price` supersignature, and move `code` to Price.



```
all r: Rental |
  #rentalsC.r + #rentalsD.r = 1
```

Figure 8.1: Video store model refactoring.

For Refactoring A, the `Associate` signature is introduced as a generalization with Alloy Law 8 (L-R), in which `Customer` and `Dependant` become their subsignatures – invariant `Associate = Customer + Dependant` is included. Next, relation `rentals` is introduced with Law 1, in terms of `rentalsC` and `rentalsD`.

```
sig Associate {
  rentals: set Rental
}
sig Customer extends Associate {
  rentalsC: set Rental
}
sig Dependant extends Associate {
  main: one Customer,
  rentalsD: set Rental
} ..
fact Invariants {
  all r:Rental | one (r.~rentalsC + r.~rentalsD)
  Associate = Customer + Dependant
  rentals = rentalsC + rentalsD
}
```

Next, invariants for defining `rentalsC` and `rentalsD` are deduced and introduced to fact `Invariants` with Law 14. In addition, from the original invariant over these relations, we can deduce that every rental relates to exactly one associate.

```
..
fact Invariants {
  Associate = Customer + Dependant
  rentals = rentalsC + rentalsD
  rentalsC = rentals & (Customer -> Rental)
  rentalsD = rentals & (Dependant -> Rental)
  all r: Rental | one r.~rentals
}
```

The previous steps allows us to remove both `rentalsC` and `rentalsD` with Alloy Law 1 (R-L), along with their definitions. This step concludes refactoring A.

```
sig Associate {
  rentals: set Rental
}
sig Customer extends Associate { }
sig Dependant extends Associate {
    main: one Customer
} ..
fact Invariants {
  Associate = Customer + Dependant
  all r: Rental | one r.~rentals
}
```

Refactoring B aims to introduce the concept of *movie copies* to the object model, which is commonly used in the video rental domain. We first remove the multiplicity of relation `movie` – written now as an invariant – in order to create an alternative path from `Rental` to `Movie` afterwards, introducing a new signature (`Copy`) with two relations (`copy,copy_of`) and an invariant (`movie = copy.copy_of`).

```
sig Rental {
  movie: set Movie,
  copy: set Copy
}
sig Copy {
  copy_of: set Movie
}
fact Invariants {
  ..
  all r:Rental | one r.movie
  movie = copy.copy_of
}
```

Deductions from the invariants in the model allows us to introduce new formulas replacing `movie` with `copy.copy_of`. Therefore, the `movie` relation can then be removed from the model, with Law 1 (R-L), establishing the expected outcome from refactoring B.

```
sig Rental {
  copy: set Copy
}
sig Copy {
  copy_of: set Movie
}
fact Invariants {
  ..
  all r:Rental | one r.copy.copy_of
}
```

Finally, in the last refactoring, the State Design Pattern is introduced for types of movies available in the video store. First, signatures `New` and `Regular` are included with Alloy Law 7 (L-R), representing the types of movies that will establish the cost for each rental. Next, both are made subsignatures of `Price`, a new signature added by Law 8 (L-R) representing the state abstract entity. The `PriceCode` signature now declares relation `mapsTo`, with no definition.

```
sig Movie {
  code: one PriceCode
}
sig Price { }
sig New extends Price { }
sig Regular extends Price { }
sig PriceCode {
  mapsTo: set Price
}
..
fact Invariants {
  ..
  Price = New + Regular
}
```

Next, we manipulate the `code` relation in `Movie`. The goal is to move the target of this relation from `PriceCode` to `Price`, in order to establish the structure of the desired pattern. Our technique is to reverse and move `code` to the `Price` signature, for finally reversing the relation, as showed in Figure 8.2. Each step is carried out with the defining invariant for each relation introduced with Alloy Law 1 (L-R) – while one relation is created, the previous one is removed by deducing its defining formula and applying Law 1 (R-L). For example, after introducing `code'` with invariant `code'=~code`, `code` can be removed with a deduced invariant `code=~code'`.

During these law applications, an interesting aspect of this example came up: we cannot replace `code'` by `code''` unless `mapsTo` is a *bijective function* (1:1 multiplicities between `Price` and `PriceCode`, as depicted in Figure 8.2). Formally, the invariant for removing `code'` (`code'=mapsTo.code''`) can only be deduced from the original invariant (`code''=(~mapsTo).code'`) if `mapsTo` is a bijection. By analyzing the example from the book, we realized that this relationship is *informally assumed* by Fowler when moving the field in the program, but not explicitly stated [37]. Therefore, this is an

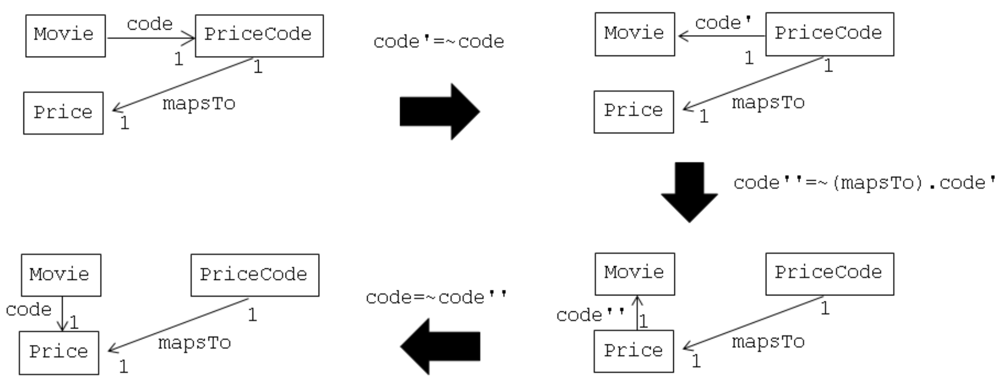


Figure 8.2: Steps for moving the relation

evidence that our formal investigation on refactoring may bring light on similar issues that arise in practice. We then assume this invariant for the refactoring to be applicable. The resulting model is showed in the next Alloy fragment.

```
sig Movie {
  code: one Price
}
sig Price { }
sig New extends Price { }
sig Regular extends Price { }
sig PriceCode {
  mapsTo: set Price
}
..
fact Invariants {
  ..
  Price = New + Regular
  all p:Price | one p.~mapsTo
  all pc:PriceCode | one pc.mapsTo
}
```

8.1.2 Program refactoring

From the sequence of applied laws of modeling in the three model refactorings, the correspondent strategies are executed. Table 8.1 shows the matching laws and strategies.

For the first refactoring, the opening strategy automatically introduces the `Associate` superclass. Other strategies include the new field `rentals`, replacing old references for the previous fields `rentalsC` and `rentalsD`; this replacement cannot be made automatically, due to class refinement issues – invariant translation and simulation proof. Field `rentals` is public; after its introduction, part of the outcome is showed in the following program fragment. Writings to **self**`.rentalsC` are augmented with writings to **self**`.rentals` within class `Customer`.

Table 8.1: Strategies for the Video Rental Case Study

Order	Alloy law	Strategy
1	Introduce Generalization(L-R)[`Associate`]	introduceGeneralization
2	Introduce Relation(L-R)[`rentals`]	introduceField
3	Introduce Formula(L-R)[`rentalsC=rentals & (Customer->Rental)`]	identity
4	Introduce Formula(L-R)[`rentalsD=rentals & (Dependant->Rental)`]	identity
5	Introduce Formula(L-R)[**all** `r:Rental` | **one** `r.~rentals`]	identity
6	Introduce Relation(R-L)[`rentalsC`]	removeField
7	Introduce Relation(R-L)[`rentalsD`]	removeField
8	Remove One Relation(L-R)[`movie`]	fromSingleToSetField
9	Split Relation(L-R)[`copy,Copy,copy_of`]	splitField
10	Introduce Relation(R-L)[`movie`]	removeField
11	Introduce Signature(L-R)[`New`]	introduceClass
12	Introduce Signature(L-R)[`Regular`]	introduceClass
13	Introduce Generalization(L-R)[`Price`]	introduceGeneralization
14	Introduce Relation(R-L)[`code'`]	removeField
15	Introduce Formula(L-R)[`code=~code'`]	identity
16	Introduce Relation(R-L)[`code`]	removeField
17	Introduce Relation(L-R)[`code''`]	introduceField
18	Introduce Formula(L-R)[`code'=mapsTo.code''`]	identity
19	Introduce Relation(R-L)[`code'`]	removeField
20	Introduce Relation(L-R)[`code`]	introduceField
21	Introduce Formula(L-R)[`code''=~code`]	identity
22	Introduce Relation(R-L)[`code''`]	removeField

```
class Associate {
  pub set Rental rentals;
}
class Customer {
  ..
  set Rental rentalsC;
  unit addRental(Rental r) {
    unpack self;
      self.rentalsC:= self.rentalsC ∪ r;
      self.rentals:= self.rentalsC ∪ r;
    pack self;
  } ..
}
..
```

With the coupling invariant (**self is** `Customer`) ⇒ **self**`.rentalsC=`**self**`.rentals`, relation `rentalsC` can be removed (and analogously, `rentalsD`), with strategy *removeField*. For refactoring B, `movie` in class `Rental` is made a single field; for adding movie copies, the *splitField* strategy creates two fields, `copy` and `copy_of`, besides a new class (`Copy`). The class refinement contained in this strategy can automatically duplicate writings using the old field, as exemplified in the next program fragment.

```
  unpack obj;
    obj.movie:= aMovie;
    obj.copy.copy_of:= aMovie;
  pack obj;
```

The `movie` field can then be removed. The refactored program presents an interesting aspect: even though the concept of copy is introduced, the business logic associated with movie copies is not operational, demanding evolutionary changes. For instance, two copies of the same movie are still linked to two different instances of `Movie`. Therefore, although the resulting program has its behavior preserved, being also in conformance with the resulting model, it does not reflect user intent. This aspect may be a limitation of model-driven refactorings from object models. We visualize two potential solutions: more concrete models with programming logic, as in MDA [66], or to consider behavioral models, in addition to object models (a possible future work). The refactored program can be seen next. With user feedback, additional improvements could be done, such as pulling up `name` to `Associate`, or making `rentals` private.

```
class Associate {
  pub set Rental rentals;
}
class Customer {
  string name;
  string id;
  set Rental rentalsC;
  unit addRental(Rental r) {
    unpack self;
      self.rentalsC:= self.rentalsC ∪ r;
      self.rentals:= self.rentalsC ∪ r;
    pack self;
  } ..
}
class Rental {
  Copy copy;
  int daysRented; ..
}
class Copy {
  Movie copy_of;
}
```

Adding the State Design Pattern includes strategies for introducing three new classes and moving the `code` field. As we explicitly defined an one-to-one relation between a `PriceCode` to `Price` (`mapsTo` field), the strategy can be freely applied. When applying the strategies, it becomes visible how reversing a relation in Alloy, which is a rather simple transformation, can clutter up the program with implementation details. The following fragment shows how class `Movie` is affected by reversing `code` with `code'` – only the first field reversion.

```
class Movie {
  string title;
  unit setCode(PriceCode pc) {
    unpack self;
      pc.code':= self;
    pack self
  }
  PriceCode getCode(){
    result:= elem({pc:PriceCode | self ∈ pc.code'})
  }
}
```

This outcome brings out an important conclusion: while changing the direction of relations in object models are straightforward, this is not the case for class fields; it could be in part anticipated, since this is a known complex connection point between analysis and design activities in software processes.

8.2 Java Types Specification

The distribution of the Alloy Analyzer [1] offers an example object model describing the basic notions of typing in Java. It covers reference types (classes and interfaces), with variables and instances of such types. Primitive types are not considered. The following Alloy fragment describes the declaration of Java's main types.

```
abstract sig Type {
   subtypes: set Type
}
sig Class, Interface extends Type {}
one sig Object extends Class {}
```

With the **abstract** keyword, the subsignatures of `Type` are its partition. The `one` keyword constrains the `Object` signature to contain exactly one instance. In addition, the following fact declaration adds three invariants: (1) every type is subtype of `Object`, in which the `subtypes` relation is applied indefinitely until the leaf instances, by a reflexive-transitive closure operator ; (2) no type is a subtype of itself, by a successive dereference of `subtypes` with the transitive closure operation (ˆ); (3) every type is a subtype of at most one class – multiple inheritance is not allowed in Java – by using transpose (˜) of `subtypes`. The keyword **no**, used as a quantifier, argues that there exists no type complying with the formula, while **in** denotes set membership or containment.

```
 fact TypeHierarchy {
   Type in (Object.*subtypes)
   no t: Type | t in (t.^subtypes)
   all t: Type | lone ((t.~subtypes) & Class)
 }
```

Next, we show part of a simple type-checker for expressions. The following signature declarations express that every object relates to a type (its creation type) that is a class. A variable may hold an instance, and has a declared type.

```
sig Instance {
   type: Class
}
sig Variable {
   holds: lone Instance,
   type: Type
}
```

Finally, the `TypeSoundness` fact states that all instances held by a variable have types being direct or indirect subtypes of the variable's declared type.

```
fact TypeSoundness {
  all v: Variable | (v.holds.type) in (v.type.*subtypes)
}
```

[1] http://alloy.mit.edu

Next, we provide an implementation for the object model above. A partial implementation for the `Type` class is showed next. We assume that this implementation follows the required conformance relationship and the confinement requirements. The method `isSupertypeOf` is a simplification, as the correct implementation must check the whole hierarchy, not only the first level below.

```
class Type {
  set Type subtypes; ..
  unit addSubtype (Type t) {
    self.subtypes:= self.subtypes ∪ {t}
  }
  bool isSuperTypeOf (Type t) {
    if (t ∈ self.subtypes)
      then result:= true
      else result:= false
  }
..
}
```

8.2.1 Model refactoring

The system was modeled based on the `subtypes` relation. We assume a refactoring in which this model is redesigned in terms of *supertype* relations, namely `extends` and `implements`. The required transformations affect the main model declarations, requiring changes to a conforming program. We first apply Alloy laws; after all laws are applied, the corresponding strategies are applied to source code in the same order. This refactoring is depicted by the UML diagrams in Figure 8.3, with highlight on the affected part of diagram (b).

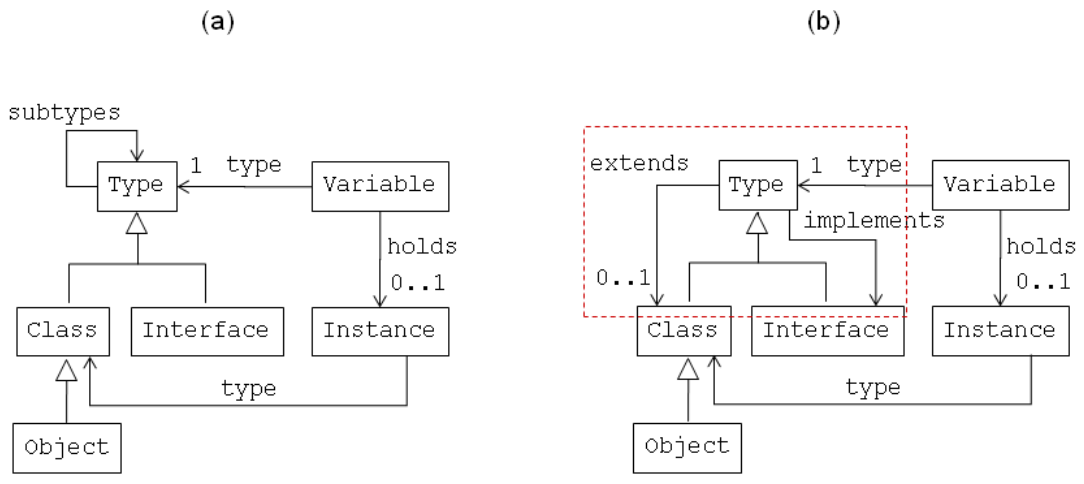


Figure 8.3: Refactoring for the Java Types Alloy model

In the model, we first establish a relationship between the original (`subtypes`) and the intended (`extends`, `implements`) relations. In fact, if B is subtype of A, B implements or extends A; hence, all objects related by `subtypes` are given by the objects related by the inverse of `implements` and `extends` combined. In Alloy, `subtypes=~extends + ~implements`.

In order to introduce the new relations, we include an empty fact; this transformation is formalized by Alloy Law 15 (L-R). Next, we introduce `extends` and `implements` to the `Type` signature, applying Law 1 (L-R) twice; their definitions are added to the newly-introduced `Definitions` fact. For instance, `implements` is defined as the transpose of `subtypes`, but with range restricted to `Interface` instances (intersection with the `Type->Interface` cartesian product).

```
abstract sig Type {
   subtypes: set Type,
   extends: set Class,
   implements: set Interface
}
fact Definitions {
   implements = (~subtypes & (Type->Interface))
   extends = (~subtypes & (Type->Class))
}
```

Our aim is to derive the desired definition of `subtypes`, in order to replace its occurrences and eventually remove the relation from the model. Invariant `subtypes=~extends + ~implements` can be deduced from the `Definitions` fact and introduced to the model. A deduced formula can be included to a fact by Law 14.

```
fact Definitions {
   implements = (~subtypes & (Type->Interface))
   extends = (~subtypes & (Type->Class))
   subtypes = (~extends + ~implements)
}
```

From the third formula of the `TypeHierarchy` fact, we can deduce a formula stating that `extends` is a partial function (**all**`t:Type | `**lone**`(t.extends)`), by replacing `subtypes` with its definition. With this new formula, we can change qualifier to **lone**, with Law 18 (R-L).

```
abstract sig Type {
   subtypes: set Type,
   extends: lone Class,
   implements: set Interface
}
fact Definitions {
   implements = (~subtypes & (Type->Interface))
   extends = (~subtypes & (Type->Class))
   subtypes = (~extends + ~implements)
}
```

Next, references to `subtypes` are replaced by its definition. The first two invariants in fact `Definitions` can now be reduced to tautologies, from the intersections (`implements=implements`, for example). They reduce to true, being removed from the model.

```
fact Definitions {
   subtypes = (~extends + ~implements)
}
```

We can finally remove `subtypes` and its definition using Law 1(R-L), as all references to `subtypes` have been replaced. Next, the `Definitions` fact becomes empty, being also removed. The resulting model is shown in the Alloy fragment below.

```
abstract sig Type {
   extends: lone Class,
   implements: set Interface
}
sig Class, Interface extends Type { }
one sig Object extends Class { }
fact TypeHierarchy {
   Type in (Object.*(~extends + ~implements))
   no t:Type | t in (t.^(~extends + ~implements))
   all t:Type | sole ((t.~(~extends + ~implements)) & Class)
}
sig Instance {
   type: Class
}
sig Variable {
   holds: lone Instance,
   type: Type
}
fact TypeSoundness {
   all v: Variable | (v.holds.type) in (v.type.*(~extends + ~implements))
}
```

8.2.2 Program refactoring

Given the applied sequence of laws of modeling is known, we are able to apply the correspondent strategies to the underlying source code. Table 8.2 shows the strategies matching the applied Alloy laws. Some of the Alloy laws correspond to *identity* strategies; these laws deal with either Alloy syntactic sugar or invariants; they do not affect the program.

First, strategy *introduceField* introduces new fields to the *Type* class based on the relations included in the model. For example, the `extends` field is introduced with a class refinement, using a coupling invariant from the relation definition. In this case, the invariant is formulated as follows:

```
self.extends = {s: Class | self ∈ s.subtypes}
```

This refinement is not automatic, due to the complexity of the coupling invariant. First, `extends` is made public, for allowing changes to the field (Refactoring 4 for encapsulating field could be used as well, adding getter and setter methods). As the coupling invariant is applied to the program, the assignment **self**.`subtypes:=` **self**.`subtypes` ∪ {`t`} in the `addSubtype` method must be extended with an assignment to `extends`, enforcing the coupling invariant. An **if** statement is included, as showed next.

Table 8.2: Strategies for the Java Types Case Study

Order	Alloy law	Strategy
1	Introduce Fact(L-R)[`Definitions`]	identity
2	Introduce Relation(L-R)[`extends`]	introduceField
3	Introduce Relation(L-R)[`implements`]	introduceField
4	Introduce Formula(L-R)[`subtypes = (~extends + ~implements)`]	identity
5	Introduce Formula(L-R)[**all** `t:Type` | **lone**`(t.extends)`]	identity
6	Remove lone relation(R-L)[`extends`]	fromSetToOptionalField
7	Introduce Formula(R-L)[`extends=extends`]	identity
8	Introduce Formula(R-L)[`implements=implements`]	identity
9	Introduce Relation(R-L)[`subtypes`]	removeField

```
class Type {
  set Type subtypes;
  pub set Class extends;
  ..
  unit addSubtype (Type t) {
    unpack self;
      self.subtypes:= self.subtypes ∪ {t};
      if (self is Class) then t.extends:= {self}
    pack self;
  }
  ..
}
```

The strategy is analogously reapplied to introduce the `implements` field. The resulting *Type* class is presented as follows:

```
class Type {
  set Type subtypes;
  pub set Class extends;
  pub set Interface implements;
  ..
  unit addSubtype (Type t) {
    unpack self;
      self.subtypes:= self.subtypes ∪ {t};
      if (self is Class) then t.extends:= {self};
      if (self is Interface) then t.implements:= {self}
    pack self;
  }
  bool isSuperTypeOf (Type t) {
    if (t ∈ self.subtypes ∨ self ∈ t.extends ∨ self ∈ t.implements)
      then result:= true
      else result:= false
  }

  ..
}
```

After a number of identity transformations, the *extends* field can be turned into a single field. In this case, strategy *fromSetToOptional* can be automatically applied.

```
class Type {
  set Type subtypes;
  pub Class extends;
  pub set Interface implements;
  ..
  unit addSubtype (Type t) {
    unpack self;
      self.subtypes:= self.subtypes ∪ {t};
      if (self is Class) then t.extends:= elem({self});
      if (self is Interface) then t.implements:= {self}
    pack self;
  }
  bool isSuperTypeOf (Type t) {
    if (t ∈ self.subtypes ∨ self ∈ {t.extends} ∨ self ∈ {t.implements})
      then result:= true
      else result:= false
  }
  ..
}
```

Finally, after additional formula manipulation, `subtypes` is removed from `Type`, by applying the appropriate strategy. In order to eliminate the reading accesses to the field, definition **`subtypes=(~extends + ~implements)`** is used as a class invariant, which is presented as manually translated below.

self`.subtypes=`{`s:Type | `**self** ∈ `s.extends`} ∪ {`s:Type | `**self** ∈ `s.implements`}

The field can then be removed, along with its writing accesses, and reads are replaced according to the class invariant. The resulting `Type` class is shown in the following fragment. Regarding quality of the refactored program, improvements cannot be made automatically simply from model assumptions, so user feedback can help establishing that the newly-introduced fields are better encapsulated as private fields. Even though confinement is maintained in the outcome, this can be restricted for future modifications. Furthermore, several expressions could be simplified: within the **if** condition, the first disjunction member can be eliminated, since it is redundant; also, **self** ∈ {`t.extends`} can be simplified to **self** `= t.extends`, by a property of the unitary set.

```
class Type {
  pub Class extends;
  pub set Interface implements;
  ..
  unit addSubtype (Type t) {
    unpack self;
      if (self is Class) then t.extends:= elem({self});
      if (self is Interface) then t.implements:= {self}
    pack self;
  }
  bool isSuperTypeOf (Type t) {
    if (t ∈ {s:Type | self ∈ s.extends} ∪ {s:Type | self ∈ s.implements} ∨
    self ∈ {t.extends} ∨ self ∈ {t.implements})
      then result:= true
      else result:= false
  }
  ..
}
```

8.3 Pull Up/Push Down Relation Refactorings

Section 3.3 describes how Alloy laws can be composed into model refactorings, that may be directly applied by modelers. For that, we use a refactoring to push down a relation to a subsignature, as long as there is an invariant constraining the relation to the receiving subsignature (Refactoring 1). In contrast, previous versions of the law catalog regarded this refactoring as a separate law, influenced by anterior definitions of strategies [41]. For this law, two strategies were defined. Our goal is to compare the resulting program in two versions of this refactoring: as an isolated strategy or as a sequence of strategies. We present both results, along with a discussion over the two outcomes. For illustration, we use the previous file system example.

8.3.1 Law

As an example of law, we can push down a relation with an invariant stating that the relation only relates elements of the subsignature (**no** `(T-S).r`), and no potential type errors is encountered – as formalized in the following law (*push down relation*). Similarly, we may pull up a relation from a signature to its parent by adding a formula stating that this relation only maps elements of the subsignature, given no name conflicts are found.

Law ⟨*push down relation*⟩

$$\boxed{\begin{array}{l} ps \\ \textbf{sig}\ T\ \{ \\ \quad rs, \\ \quad r : \textbf{set}\ U \\ \} \\ \textbf{sig}\ S\ \textbf{extends}\ T\ \{ \\ \quad rs' \\ \} \\ \textbf{fact}\ F\ \{ \\ \quad forms \\ \quad \textbf{no}\ (T - S).r \\ \} \end{array}} \quad =_{\Sigma,v} \quad \boxed{\begin{array}{l} ps \\ \textbf{sig}\ T\ \{ \\ \quad rs \\ \} \\ \textbf{sig}\ S\ \textbf{extends}\ T\ \{ \\ \quad rs', \\ \quad r : \textbf{set}\ U \\ \} \\ \textbf{fact}\ F\ \{ \\ \quad forms \\ \} \end{array}}$$

provided
($\rightarrow$) $E.r$, where $E \leq T$ and $E \not\leq S$, does not appear in *ps* or *forms*;
($\leftarrow$) T's hierarchy in *ps* does not declare any relation named r.

For this law, two strategies were defined, for refactoring a conforming program in both directions. First, strategy *pushDownField* refactors a program containing the two involved classes, yet intermediate classes can also be declared within the hierarchy. Also, several statements may access the field in expressions – these expressions must be modified before actually pushing down the field. The file system implementation showed in Section 6.1 can be refactored by this strategy, resulting in the following program fragment.

```
class FSObject{
  Name name;
  set Dir getContents() {
    if (self is Dir)
      then result:=((Dir)self).contents
      else result:= ∅
  }
  unit setContents(set FSObject c) {
    if (self is Dir) then ((Dir)self).contents:= c
  }
```

```
}

class Dir extends FSObject{ pub Dir set contents; .. }
class File extends FSObject{
  constr { .. }

class Main{
   unit main(){
      File f:=new File,currentFSObj:=null in ..
          currentFSObj:=(FSObject)self.inout;
          if (currentFSObj is Dir)
            then currentFSObj.setContents({f})
            else currentFSObj.setContents(∅)
    }
}
```

In strategy *pushDownField*, `r` first becomes a public field. Several accesses to `r` are rewritten in two ways: (1) when the left expression is a subtype of `T` (`FSObject`) but not typed as `S` (`Dir`) – **if** statements and casts are distributed for reading (`cmd[cc.r]`) and writing accesses, since the expression may either yield or not a `S` instance at runtime; (2) the left expression may be subtype of `FSObject`, but not in the hierarchy branch of `Dir` – (**classes()-getBranch()**) gives a list of classes in the program except for classes in the branch from `FSObject` that includes `Dir` until leaf classes. In this case, only `File` is selected. For this class, the invariant states that accesses to `contents` must always be empty, thus they are rewritten as such. Figure 8.4 defines these two cases in the `FSObject` hierarchy.

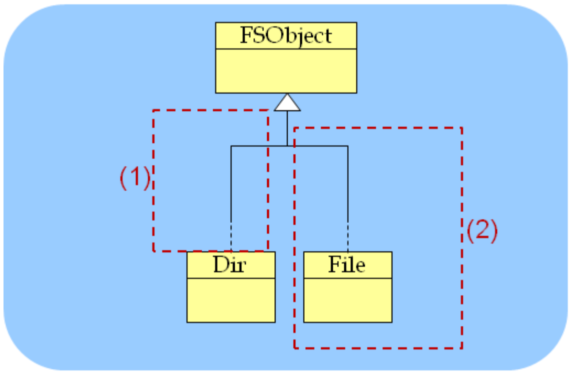


Figure 8.4: Cases of expression rewriting for moving a field from `FSObject` to `Dir`

After those statement replacements, the field is moved to `S`, from `T` throughout its subclasses, until it gets declared in `S`.

```
Tactic pushDownField(r : Field, T, S : Class)
        (law changeVisibilityPrivatePublic(r, →) | skip);
        law replace(getHierarchyTopDown(T, super(S)), "cmd[cc.r]",
                "if (exp is S) then cmd[((S)exp).r] else cmd[∅]");
        law replace(getHierarchyTopDown(T, super(S)), "cc.r := exp",
                "if (cc is S) then ((S)le).r := exp");
        applies to le.r{isExactly(exp, classes() − getBranch(T, S))} •
           law replace("cmd[le.r]", "cmd[∅]");
        law replace("$le.r := \ exp$","skip");
      law $moveFieldToSuperclass$(getHierarchyTopDown($T, S$), $r$, $\leftarrow$);
end
```

The opposite strategy (*pullUpField*) requires no changes to statements, as Law 37 presents no provisos for its L-R application. Shadowing is ignored, thus nothing is done regarding name conflicts. For supporting this change, **getHierarchyBottomUp**`(S,T)` yields a list of classes, in the example from `Dir` to `FSObject`.

```
Tactic $pullUpField(r : Field, S, T : Class)$
   (law $changeVisibility : priTopub(r, \rightarrow)$ | skip);
   $\mu$ $cc$ : getHierarchyBottomUp($S, T$) $\bullet$
      law $moveFieldToSuperclass(r, cc,$ super$(cc), \rightarrow)$;
end
```

Assuming the outcome from *pushDownField* as input, the automatic application of *pullUpField* results in the following declarations. Additional quality improvements may be carried out as strategy extension, such as eliminating casts in `FSObject`.

```
class FSObject{
  Name name;
  pub set FSObject contents;

  set Dir getContents() {
    if (self is Dir)
      then result:=((Dir)self).contents
      else result:= ∅
  }
  unit setContents(set FSObject c) {
    if (self is Dir) then ((Dir)self).contents:= c
  }
}

class Dir extends FSObject{ .. }
class File extends FSObject{
  constr { .. }
}
..
```

8.3.2 Composite Refactoring

Chapter 3 presents Refactoring 1 defined as a composition of Alloy laws, which is the current representation for pushing down a relation. From that definition, we discuss in this section the application of strategies in the file system example for both directions of the refactoring.

Push Down Relation

Table 8.3 details the order of law applications for Refactoring 1(L-R), along with the corresponding strategies, as previously explained in Chapter 3 for the file system example.

The resulting code for the example is showed in the next fragment. With class refinement, fields `contentsDir`, `contentsFile` and `contentsX` were introduced, but

Table 8.3: Strategies for Push Down Refactoring

Order	Alloy law	Strategy
1	Introduce Subsignature(L-R)[`X`]	introduceSubclass
2	Introduce Relation(L-R)[`contentsDir`]	introduceField
3	Introduce Relation(L-R)[`contentsFile`]	introduceField
4	Introduce Relation(L-R)[`contentsX`]	introduceField
5	Introduce Formula(L-R)[`contents=contentsDir+contentsFile+contentsX`]	identity
6	Introduce Relation(R-L)[`contents`]	removeField
7	Introduce Formula (L-R)[`contentsFile={}`]	identity
8	Introduce Formula (L-R)[`contentsX={}`]	identity
9	Introduce Relation(R-L)[`contentsFile`]	removeField
10	Introduce Relation(R-L)[`contentsX`]	removeField
11	Introduce Subsignature(R-L)[`X`]	removeSubclass

only the first is maintained (we did not rename the field for highlighting differences from the law version). From these removals, residual implementation details from the auxiliary `X` class are seen in the source code, specially field `type` and method `isX`, assuming the type test for `X` that had to be removed from `getContents`; also, the **if** statements within `getContents` preserve details from the removed fields.

```
class FSObject{
  Name name;
  string type;
  set Dir getContents() {
    if (self is Dir)
      then result:= ((Dir)self).contentsDir
      else if (self is File)
        then result:= ∅
        else if (self.isX()) then ∅
  }
  unit setContents(set FSObject c) {
    if (self is Dir) then ((Dir)self).contents:= c
  }
  bool isX(){
    result:= self.type = "X"
  }
  ..
}
class Dir extends FSObject{
  pub Dir set contentsDir; ..
}
class File extends FSObject{
   constr {
      if (self is File) then skip
    }
} ..
```

We conclude that for the refactoring case, the strategies are able to perform their functionality – preserves behavior and conformance with the refactored model. Nevertheless, the quality of the resulting code is better preserved by applying a single, specialized strategy. For an isolated law, the strategy is tailored for the specific problem of pushing down a field with the defined invariant, which does not require class refinements. The refinements in the second case clutter up the program, concerning the auxiliary class and fields used in the refactoring; this result corroborates the issues with quality factors described in Section 6.5.

Pull Up Relation

Likewise, Table 8.4 shows law applications for Refactoring 1(R-L) and corresponding strategies, which is simpler than pushing down relations.

Table 8.4: Strategies for Pull Up Refactoring

Order	Alloy law	Strategy
1	Introduce Relation(R-L)[`contentsFSObj`]	introduceField
2	Introduce Formula(L-R)[`no (FSObject-Dir).contents`]	identity
3	Introduce Formula(L-R)[`no (FSObject-Dir).contentsFSObj`]	identity
4	Introduce Relation(R-L)[`contents`]	removeField

The resulting source code for the file system is showed next, being very similar to the one for the law version. In this case, refinements are applied only twice, with less auxiliary elements.

```
class FSObject{
  Name name;
  pub set FSObject contentsFSObj;

  set Dir getContents() {
    if (self is Dir)
      then result:= self.contentsFSObj
      else result:= ∅
  }
  unit setContents(set FSObject c) {
    if (self is Dir) then self.contentsFSObj:= c
  }
}
class Dir extends FSObject{ .. }
class File extends FSObject{
  constr { .. }
}
..
```

8.4 Chapter summary

The case studies presented in this chapter served as a practical evaluation for the developed strategies. Three examples were used for applications: a video store model from Fowler's book on refactoring [37], a major refactoring on the Java types specification and a comparison between two different versions of push down relation refactoring. These cases helped find errors and inconsistencies in strategy definitions, as well as provide a more elaborate of the issues reported in Chapter 6.

Chapter 9

Conclusion

In this work, we provide an approach for semi-automatically refactoring programs based on object model refactorings founded by semantics-preserving laws. A sequence of formal behavior preserving program transformations is associated to each predefined model refactoring. Applying a model refactoring triggers the corresponding sequence of program refactorings, which (1) update code declarations as refactored in the model and (2) adapt statements according to the modified declarations. This is accomplished with invariants, which are assumed throughout all program's executions, through a conformance relationship. Although developers only apply refactorings to object models, both artifacts get refactored, avoiding most of the required manual updates on source code.

9.1 Summary of Contributions

The formal solution devised in this thesis is the outcome of investigations on refactoring at different levels of abstraction, particularly object models, from analysis and design activities, and source code resulting from implementation activities. Our solution represents a feasible alternative for conformance maintenance between object-oriented programs and object models, at least for refactoring tasks. Although we deal only with refactoring, other evolution tasks could potentially apply the principles behind our approach. Therefore, knowing, from Section 6.1, that code generation and reverse engineering have showed inappropriate in evolutionary development, a different approach to evolution could be tried.

The idea of primitive transformations composing useful refactorings is promising, not only in formal contexts; our research group has been using this idea [45, 44, 73, 75]. We rely on this principle create strategies for each Alloy law. In this way, model refactoring can be semi-automatically replicated to the program (answering to the Research Question #1 in Section 1.3). This may encourage maintenance of object models during refactoring, since manual updates to maintain conformance and correctness are considerably avoided, with the exception of changes in class representation (class refinement).

This approach can serve as basis for refactoring and evolution-related tool support. In this case this tool could record every law used in an applied composition and search for the corresponding strategies (some would be identity strategies), applying those, in order, to the implementation. The resulting implementation will be in conformance with the original Alloy model; such a relation is preserved throughout the strategies.

This approach may also be useful to improve tool support for refactoring, since the semantic properties from object models can aid refactoring automation. Refactoring tools, such as Eclipse [30], help updating programs in refactorings that span several modules. In a number of strategies from this work, the invariants expressed in the object model offers an instrument for more efficient tools, using this semantic information to extend its automatic refactoring capabilities. This is an important feature in strategies such as *removeGeneralization*, in which an invariant stating that a superclass is abstract (semantic information) allows this class to be removed, along with their field and methods. This information could not be obtained solely from the source code. Similar results can be acquired in strategies *removeSubclass*, *fromSetToOptional* and *removeField* [74]; in the latter, the benefit is only seen for the three cases chosen to be automatic (empty field, clone field and composition field).

According to our experience, the most important result from this investigation, was to match the conformance relationship with the strategies (Research Question #2). Our general methodology has been to try to devise a syntactic conformance relationship as loose as possible, according to strategy definitions. We believe that the results show a relatively abstract conformance relationship, whereas still amenable to interesting model-driven refactorings. A more abstract conformance relationship would considerably make strategies more complex, cluttering the program in order to maintain some degree of automation.

Another important task for maturing the solution was the process of proof development (Research Question #3). Although the first case studies helped to improve strategy definitions, the detailed view over strategies demanded by proofs – specially semantic conformance proofs – allowed us to detect problems in strategies and adjust the required conformance relationship. For instance, the *abstractConstraint* from the syntactic conformance was essential for the soundness of strategies involving introduction and removal of super and subclasses, not previously detected in the case studies (Research Question #4).

9.2 Assumptions and Limitations

A few assumptions have been adopted in this thesis. First, strategies can only work if conformance checking has been applied to guarantee that the program's behavior maintains the invariants introduced by the object model. Although conformance checking tool support is evolving in order to aid verification (including static [36, 60] and runtime checking [27]), it is still incipient in practice. In addition, we take the closed-world assumption that we have access to the full source code of a program – some strategies must have access to the whole class table.

The semantics of the chosen programming language drives much of the study on program transformations. For this thesis, we rely on a catalog of transformations that must be behavior preserving. Results from work on the ROOL language [16] include a catalog of primitive laws of programming; this language was used during most of the work for this thesis. However, the language presents copy semantics for objects, whereas Java and other widely-used languages present a reference semantics. These characteristics simplify the language for formal reasoning, however restraining its practical application. Further, we detected an important issue: the semantic difference between model and program semantics requires choices that make the conformance relationship significantly complex.

In this context, we adopted a language with reference semantics, similar to mainstream programming languages. Nevertheless, a different problem arises from this choice: aliasing via pointers, in which shared mutable objects allows breaking encapsulation, even for private fields, as described in Chapter 4. If a uncontrolled client updates the state of a shared object, an invariant of a given class may be violated. Therefore, the need for confinement is ubiquitous in practice; we adopted the discipline for confinement described by Banerjee and Naumann [8], which was chosen from a number of other confinement proposals, due to its less restrictive and statically checkable approach. Although confinement restricts the number of programs amenable to model-driven refactoring, it helps in re-engineering object-oriented software by exposing potential software defects, or at least making subtle dependencies visible, possibly motivating widespread utilization.

Although object models allow reasoning about program properties, other modeling views are usually applied in software development. For instance, modeling dynamic aspects of a system is commonly seen in object-oriented design; sequence diagrams, from the UML [15], and Alloy models with traces [58] are exemplars. Our approach focuses on object models, thus conformance with dynamic models is not guaranteed.

System implementations usually include database support, especially for information systems. In this context, database structures (schemas) strongly follow the domain information originated in object models; minor differences come from performance adjustments such as denormalization [78]. A clear limitation from our approach is that we do not deal with database particularities for refactoring tables and relationships according to the structures changed in source code and object models. Nevertheless, a similar approach could be applied, given the conformance relationship between program and database structures is well-established.

9.3 Research Conjectures

The investigation of model refactoring and its implications to source code can provide considerable evidence over the challenges that effective MDA tool support will have to face in order to support evolution. While code generation is effective by the use of *markings* on abstract models and patterns of tranformations [66], constructing artifacts from others contributes to the problems previously seen in evolution. By now, MDA-based tools [25, 80, 91] show similar limitations as those observed in round-trip engineering tools, as manual updates are still required to complete model-driven evolution (custom implementation details may depend on the obsolete structures).

Furthermore, program refactoring in practice may also obtain benefits with some of our ideas. Developers usually follow a uniform sequence of activities when adding functionality to software (at least in agile methodologies strongly based on refactoring, like XP [12]):

- Refactor the code to remove bad dependencies that might make additions hard;

- Write new tests to test whether the observable behavior is maintained;
- Add new functionality unless it involves need for additional refactoring;
- Repeat until done.

In this context, major refactorings, involving module declarations and relationships, are usually avoided, due to dependencies that must be updated, including the tests that may depend on the changed structures. These tasks require resources on which development teams are usually not willing to spend effort. A possible application of our approach would have developers specifying, or generating from source code (using static and dynamic analysis techniques), models or annotations. Structural refactorings would be applied on the model, and strategies could improve automation of the updates, based on the invariants. In thesis, even tests could be automatically updated to reflect the refactored structures, by using special strategies, for example.

One of the issues with our approach is related to refactoring quality after strategy application, as pointed out in Section 6.5.2. Due to the independence of strategy applications, information regarding the composed model refactoring as a whole is not used for improving the resulting program, often missing the refactoring's original goal – we observed this effect clearly during the *PushDownRelation* case study, presented in Section 8.3. An alternative for dealing with this problem is to refactor programs exclusively based on *the refactoring's initial and final models*, ignoring the intermediate law applications. For such approach, we see two possible alternatives: (1) a fixed catalog of major refactorings (for instance, for design pattern [39] introduction), whose corresponding strategies would be tailored for these refactorings; and (2) automatically generate a strategy from the applied model refactoring. The first option seems to be easier, but likely to end up with the same issue. The second option is visibly more complex; from the modified model structures and invariants, a tool would discover and apply the corresponding program transformations. We have not investigated this option; however, we believe that refactoring provisos could indicate some of the needed program transformations.

Our solution is centered on model refactoring, and this refactoring on models requires human intervention. Nevertheless, we may speculate on *code-driven refactoring*, taking a similar approach but centered on program refactoring. In this case, the solution aims at a different context, in which designers refactor primarily source code, although requiring conformance maintenance with object models. Analogous to our solution, program refactorings could be composed from laws of programming [26], and these laws are linked to *model strategies* containing a sequence of model transformations. The aim is to maintain semantics and conformance with the program. Some consequences can be predicted: model strategies are certainly much simpler than program strategies, albeit forming a considerably larger strategy table. Many of these corresponding strategies would be identities, since many program refactorings do not change model entities. Further, in order to offer an effective refactoring method, the programming language must offer strong support for invariant definition and manipulation – in this case, annotation languages such as JML [18] could be used.

9.4 Related Work

Our work is directly linked to the body of research on conformance and co-evolution, and indirectly to program and model refactoring. This section details some of these approaches, relating to this thesis' contributions.

9.4.1 Conformance

Our model-driven approach to refactoring strongly relies on conformance between software artifacts at different levels of abstraction; hence, methods for establishing conformance relationships are valuable for positioning our ideas in the community. A specific approach to conformance is addressed by Guéhéneuc and Albin-Amiot in their work on the relationship between object-oriented modeling and programming languages, in particular regarding binary associations, aggregations and compositions in UML class diagrams [53]. They describe algorithms that detect automatically these relationships in code. As the semantics of such constructs is not well-defined, the authors provide their own interpretation from textual descriptions. This approach introduces a more flexible conformance relationship than traditional conformance, while still maintaining the model abstract, which may be applicable to our solution as well. However, further investigation is required for assessing potential benefits of this notion to model-driven refactoring. We believe that their conformance relationship can be established by the formal framework presented in Section 5.4.

A distinct conformance relationship is dealt with by Zhang et al. [101]. Flow analysis is used for automatically establishing mapping of implementation structures to design modules, starting from an initial partially-built mapping specification. Their solution focuses on the syntactic relationship between those structures, not dealing with semantic mapping and maintenance of semantic conformance.

Another popular technique used for establishing conformance relationships is the use of meta-models as high-level descriptions of languages for software artifacts, mainly modeling languages. Paige et al. [84] investigates conformance between different modeling views – for instance, class and sequence diagrams – for a UML-similar modeling language. The authors present two approaches for building meta-models and performing "multiview consistency checking": using PVS specifications with theorem proving, and programs in the Eiffel language with regular dynamic checking. In our formal framework, specified also in PVS, we focused on basic conformance definitions, not a verification method; still, their specifications and proof strategies can be combined with our definitions for a more general conformance checking. Although the authors mention the possibility of specifying source code with meta-models, they do not deal with programs in their solution. We believe that meta-models are hardly useful to represent programming languages, as they are limited to syntax and well-formedness rules specification and checking.

Additionally, automatic conformance checking is critical to develop tools to support our approach. Conformance checking consists in automatically verifying whether an implementation is in conformance with a given specification or model. Techniques for conformance checking can be classified into at least two categories: static checking, which only applies to the implementation's source code, and dynamic analysis, which makes use of information available during the implementation's execution, not limited to artifacts available at compile time.

Regarding static analysis, a method has been devised by Jackson and Vaziri, which check properties on code based on an Alloy specification [60] extracted from source code. However, it does not scale well for large programs, as all procedure calls are inlined for the analysis, which is not appropriate for loops and recursive calls. An alternative approach [95] eliminates that problem by performing a conservative analysis of procedure abstractions, which are automatically inferred from the procedure code. Specifications representing procedure behavior are not required from the user for checking a given property. This method can be employed for checking conformance between a program and the associated model before refactoring, with likely faster analysis times, since the structural properties are previously known from the object model. Another option for conformance checking, this one with dynamic analysis, is the Embee tool [27], which captures the runtime state of a Java program at certain user-specified points. If the runtime states at those points conform with the object model, the program follows a structural correspondence with Alloy at least for that execution. Embee can be very useful in practice for testing our conformance relationship for a BN program, although dynamic analysis only guarantees conformance for a finite number of executions.

9.4.2 Co-evolution

The concept of coupled transformation by Lammel's overview [67] has a close correspondence to the law-strategy pairs in this thesis. Coupled transformations occur when "two or more artifacts of potentially different types are involved, while transformation at one end necessitates reconciling transformations at other ends such global consistency is reestablished" [67], which is the scenario for model-driven refactoring. We also agree that any notion of coupled transformations requires a notion of consistency (conformance in our solution). Model-driven refactorings can be seen as "symmetric reconciliation", where two distinct transformations – for model and program – are defined for a given conformance relationship, adapting changes according to the specific level of abstraction for which they are defined.

Also on the foundations of multi-artifact transformations is the work with the Harmony tool, based on the concept of bi-directional transformations [14]. The authors introduce the concept of relational lenses, which are pairs of transformation functions, namely *get* and *putback*, between source and target artifacts. The *get* function transforms a source artifact into a target artifact. Updates can be performed on the target artifacts, then an updated source artifact can be obtained with the *putback* function, with information from the original source artifact and the updated target artifact. Establishing an analogy, in our approach *get* is similar to the required conformance relationship, although the function is defined for generation, which is not our case. The source artifact can be a program, and the target can be an object model. An update is analogous to laws of modeling – their study considers general evolution, not only refactoring. Also, *putback* relates to strategies, in which the program is updated in terms of the modified model.

More specifically, a relational lens is acceptable when updates to models that are identity must take back to the same source artifact (program). Following this category, our approach fulfills the acceptability property, considering the identity strategy, that does not change the program. Also according to the provided classification, our approach is a very well behaved case of transformation set, in which a given target (model) is related to several source artifacts (programs). In this aspect, this related work corroborates with relevant results from our investigation: when the lens is very well behaved and acceptable, they observed that *putback* may yield more than one source, then some extra information is needed to choose one of the possible results. This is exactly what we observed in strategies that needed user feedback (Section 6.5) for improving quality factors and complex class refinement refactorings, confirming this important conclusion over the relationship between model and program evolution. Further investigation must be carried out in order to understand how our approach may be implemented into the Harmony tool.

Cazzola et al. [22] present an approach for refactoring programs, using Fowler's catalog [37], and reflecting the corresponding changes to design models, represented by a set of class, sequence and activity diagrams. For that, developers must add meta-data to the source code – Java annotations – indicating what structures have been modified during the refactoring activity. A supporting tool can then extract this meta-data in order to update the design model, using reflection. We believe that our approach could be adapted to work similarly; developers could apply program refactorings based on laws of programming (as described in detail in Cornelio's thesis [26]) and reflect the changes to Alloy models using *model strategies*. However, these strategies would be rather simple, since the abstraction gap is larger in our context. The cited work considers co-evolution in more concrete models, in which there is a direct correspondence between model and program elements.

Another approach for co-evolving design and programs was devised for *intensional views* [76], defining sets of related classes or methods as a more abstract view that is consistent with changed code. In this case, intensional views are usually generated from source code, being more concrete than the analysis information conveyed by object models. In this approach, the given tool – *IntensiVE* – focuses on syntactic conformance checking between the intensional views and programs; the artifacts are not evolved in conjunction, rather models are generated and the tool helps in checking if specified syntactic rules are consistent. Therefore, IntensiVE could be used for supporting one part of our approach: whether a program is syntactically conforming to an object model, according to the required conformance relationship.

Co-evolution between models and programs is dealt with by several related approaches. Greenhouse et al. [50] addresses several problems with concurrent programs: expression of design intent for Java, analysis-based assurance of conformance of that intent with source code and support for co-evolution of code and design intent. Since object models specify structural design intent, this work closely related to ours, but focusing on concurrency-related properties. In addition, they define a number of manual refactorings that are performed jointly with design intent and source code. We share the same principle indicating that model information and supporting analysis foster more ambitious refactoring and program transformations, either considering object models or concurrency design intent. They use an annotation language (JML [18]) to represent abstract information, which consists in a feasible form to apply our approach using a single unified artifact. In fact, an annotation language based on Alloy has been proposed [65].

In another work, Harrison et al. [55] show a method for maintaining consistency between models (UML class diagrams) and Java programs, by advanced code generation from models at a higher level of abstraction, compared to simple graphical code visualization. This approach is related to ours, as the relationship between model and source code avoids round-tripping. Their conformance relationship is more flexible, as we require a more strict structural similitude between the artifacts. Also, their work focuses on the mapping from model constructs to source code, although no details are offered on how these mappings will consistently evolve. In addition, OCL constraints are not considered, which further restrains the applicability of the solution.

Also dealing with artifact co-evolution, Bottoni et al. [17] present an approach for maintaining consistency between programs and associated UML diagrams during refactoring. The involved artifacts are represented as graphs, refactorings as graph transformations. Differently from us, they consider behavioral models (UML state and sequence diagrams). However, even though dealing exclusively with structural aspects, object models bring valuable information that allows for consistent changes to the whole program. Furthermore, no evidence is provided on the semantics preservation of the given transformations.

A closely related approach for model-code consistency during evolution is presented by Lam and Rinard [85]. They propose a type system that extends an object-oriented programming language with design information (using tokens). This information is used for automatic model extraction from source code. If a development effort is code-driven, structural and behavioral models in conformance with the given tokens are offered with no cost after any evolution activity. Tokens represent abstractions from structures in code, advancing traditional reverse engineering and yielding visual descriptions of programs. Our approach is distinct as developers maintain object models as separate design intent. Nevertheless, during refactoring tasks, our approach is model-driven, since mostly models are manipulated. When using tokens, the design intent is tangled with implementation code, which must be evolved consistently by developers. In addition, structural constraints are hardly expressed when modeling with tokens, as offered by object modeling languages such as Alloy and UML class diagrams.

Following a generation-based approach for co-evolution, Cabot and Teniente [19] developed an approach for checking whether operations changing a conceptual model conflict with the model's structural invariants. The approach was viable from the use of actions for specifying operations, from UML 2.0 [81]. In this context, although inconsistencies might be caught early, the operations are stated in a rather operational style, compromising abstraction in such models. Furthermore, this approach suffers from the same limitations of round-trip engineering, in which automatic evolution is limited.

Similarly, Van Gorp et al. propose the application of program refactorings to UML class diagrams, given that the UML's metamodel is extended to include source code elements, such as method bodies [47]. Their aim is to refactor design independently of programming languages. However, the approach considers code visualizations, as UML diagrams faithfully represent the source code structure.

Another context for coupled transformations is software that must evolve according to changes in its database schema. Hursch's Ph.D. thesis [57] aims at a similar problem: maintaining consistency after refactoring between database schemas and object-oriented programs, based on a graph terminology (classes are vertexes, relations are edges).

Analogously, they use primitive transformations on models (additions, removals and changes), and algorithms for applying transformations in the related artifacts, similar to strategies, but not only for source code; the whole set of artifacts is transformed by these algorithms. We consider our approach more applicable, since Hursch bases his work on rather constrained languages; for instance, the used modeling language only allows abstract superclasses, and invariants cannot be used. Furthermore, database schemas and programs are very close in terms of conformance relationship; our notion of conformance is more open. In fact, hursch's refactorings involve type-related updates, not using any semantic information.

Cleve and Hainaut [24] presents an approach to automatically transform programs based on specific transformations on database schemas. For that, the authors present a number of simple semantics-preserving transformations for schemas, also defining analogous program transformations to update source code. These transformations are specific for data manipulation statements, such as SQL commands – they do not deal with general program statements. Also, no discussion is added on whether the analogous program transformations preserve semantics and conformance with the transformed schema, as seen in this thesis.

Still related to persistence, a formal investigation has been carried out by Cunha et al. [28] on data transformations made on two levels (two-level transformations). The problem tackled involves the registration of changes in data format, which are transferred to the data's instances or another form of representation; the solution is exemplified by correspondence between XML and database schemas ($XML - DBflattening$). The authors focus on type safety in transformations, using functional programming techniques in the Haskell language. Certainly two-level data transformations are related to law-strategy pairs, even though aiming at different contexts. Their solution is specific to a rather direct conformance relationship, which restricts application; still, there is no formal definition for conformance, only for the relationship between transformations.

Finally, in order to represent strategies, we used the concept of refinement tactics. These tactics result from the problem with refinement calculi and formal developments in general, which are often long and repetitive. Identifying strategies (sequences of law applications), documenting them as refinement tactics, and using them as single transformation rules brings a profit in time and effort [63]. Also, a notation for describing derivations can be used for modifying and analyzing formal derivations, which can even be supported by tools [52].

9.4.3 Program Refactoring

Related work on the formalization and automation of program refactoring has been carried out by several researchers. Opdyke proposes refactorings to which a number of preconditions are attached, which, if satisfied by the target program, ensure the preservation of behavior [82]. His work is similar to ours as it proposes a number of primitive refactorings, including creation, deletion and change of program entities, being composed into coarse-grained refactorings that should preserve behavior as well. This approach originated a well-adopted tool for refactoring Smalltalk programs, after enhancements by Roberts [88]. In contrast, semantics preservation is informally defined as a number of properties – related to inheritance, scope, type compatibility and semantic equivalence

– that are required to hold after applying the refactoring. Also, the authors are not concerned with maintenance of consistency with models.

A closely related approach was developed by Tip et al. [97]. They realized that some enabling conditions and modifications to source code, for automated refactorings in Eclipse [30], depend on relationships between types of variables. This is evident in refactorings that involve generalization. These type constraints enable the tool to selectively perform transformations on source code, avoiding type errors that would otherwise prohibit the overall application of the refactoring. This approach is similar to ours in the sense that both use additional information (in our case, model constraints) in order to achieve advanced refactorings. In fact, both approaches might be even integrated. Due to the expressiveness of object models, they can be used as source of advanced information for applying refactorings, as the set of laws for models is enhanced.

KABA [92], a system for refactoring Java class hierarchies, was developed following a different approach. The system regards refactoring for a target set of client programs accessing a class hierarchy, trying to automatically propose refactorings by investigating (using static and dynamic analysis) the use of such classes by the clients. Semantics preservation is defined concerning the client programs. A similar idea was used by us to define our conformance relationship between object models and programs. However, laws of programming are sound with respect to any well-formed set of class declarations and a main methods.

Last, Bannwart and Müller recently conceived an approach for refactoring using program annotations [9], in order to increase confidence on behavior preservation. In their methodology, semantic preconditions are added to the program as runtime assertions. Therefore, unit tests are more effective in verifying refactoring correctness, although not ensured. In contrast with our approach, their solution is directly applicable in current refactoring tools; however, annotations are not used for modeling like object models. Also, they are not concerned in maintaining these annotations consistent with changing programs in future modifications.

9.4.4 Model Refactoring

Some approaches have been developed for tackling model refactoring. For instance, work has been done [94] on the definition of basic transformations for UML [15] models, which should preserve semantics by measuring the impact of modifications on several design views (including source code). The authors define basic operations for refactoring models, which resemble some of our primitive laws of modeling. Although some of the goals are coincident with our approach, the paper does not consider the Object Constraint Language (OCL, the annotation language for expressing constraints into UML diagrams) [98], which may invalidate the soundness of the informally stated transformations. Moreover, the approach does not clarify the conditions in which source code must be consistently updated, a major concern when working with models.

In order to provide a set of sound deductive transformations for UML class diagrams, Evans [34] defines a formal semantics for a subset of constructs of such diagrams. This approach's goal is a reasoning system based on the manipulation of diagrams. Our laws of modeling have a similar purpose, which formalizes constructs from the language, yet allowing modelers to directly manipulate the modeling language (by applying laws). In fact, these deductions do not make up an interesting catalog of transformations to be used in practice. Also, the transformations do not consider OCL constraints, which usually require enabling conditions for sound transformations.

Another set of transformation rules [46] have been proposed for UML class diagrams. They state when two class diagrams are equivalent. One distinction from our work is that our equivalence notion is necessarily symmetric. Further, some of the rules compare models with different names. Nevertheless, they do not define a general equivalence notion stating when two class diagrams are equivalent. This notion is based on an informal UML semantics. Although the transformations consider OCL constraints, the conditions in which the semantics-preserving transformations are sound are not stated, as done by our laws for Alloy.

9.5 Future Work

The work presented in this thesis raises a few interesting research questions to be further explored. For instance, in order to motivate the adoption of this approach, we can minimize the number of separate artifacts to be refactored, while still maintaining abstraction-related benefits. In this context, *annotation languages* combine abstract specifications and source code in a single artifact. As a disseminated example, the Java Modeling Language (JML) [18] includes class invariants and method pre and postconditions, besides a number of static and dynamic analysis tools. With JML, programmers can develop specifications based on a simple extension of the side-effect free expressions of Java, encouraging its widespread adoption. Our future research plans include transferring model-driven refactorings to JML contexts, in addition to investigating program refactorings and their effect over JML specifications.

Furthermore, the development of a CASE tool to support program refactoring based on the strategies is also a direction for future research. It is necessary, for instance, to investigate techniques that can be used to implement strategies, and integration with a modeling (or JML) tool supporting application of laws. In the context of this tool, we could add support for refactorings related to method changes in object-oriented programs – dynamic models, and their effect over source code. Another concern for tool support is the confluence and termination of strategies, which are not formally guaranteed.

A natural topic for evolving this work is to define support for automatic class refinement for a more comprehensive subset of invariants, using systematic translation of coupling invariants from modeled invariants, and proof techniques for establishing simulation (as described in Section 6.5.1). In fact, there is evidence in previous texts on refinement [99] that when coupling relations turn out to be functions, then the proof obligations can be simplified, or even proved by calculation. Since the coupling relations that appear in our work are functional, we believe that work can be carried out in this direction.

Also, in this thesis we initially used a programming language with copy semantics, which became a critical weakness of our solution. In order to deal with this issue, we adopted a simple work-around: confinement rules were used for ensuring correctness of class refinement, and other laws of programming that are not influenced by the reference semantics are freely applied (proofs were developed for increasing confidence). Still, our solution would benefit from a general theory of refinement for reference semantics.

Our approach only provides support for semantics-preserving transformations, as refactorings which currently have wide adoption in software development. Supporting general evolution in model-driven development environments – in addition to conformance between abstract and concrete artifacts – remains a challenge for future research. The work will probably rely on laws for general evolution, and the abstract information used in strategies for rewriting more concrete artifacts accordingly.

An additional future work is extracted from the outcome of strategy *removeSubclass*. A foundational element of this thesis is a catalog of laws of programming, initially proposed for the ROOL language [16]. The set of laws is shown to be complete in the sense that it is sufficient to reduce an arbitrary program to a *normal form* substantially close to an imperative program. The previous normal form defined for ROOL, however, still maintains class hierarchies [16], which restrains proximity to a program that should be imperative. In the *removeSubclass* strategy, the most complex step is related to eliminating type tests for a class that is not guaranteed to have subclasses. In this case, we came up with a solution, as detailed in the strategy rule *eliminateTypeTests*, that replaces subtyping information from the **extends** clause by a `type` field in the superclass; type tests `x` **is** `X` are replaced by a call to a newly-introduced method – `x.isX()`. We believe that this solution may be used to improve the derivation of a normal form for ROOL, and, consequently, extend the notion of relative completeness of laws of programming.

Appendix A

Alloy Laws

The laws in this catalog are taken from Gheyi's thesis on a refinement theory for Alloy [42].

A.1 Signatures

Law 2. ⟨*introduce subsignature*⟩

$$
\begin{array}{|l|}
\hline
ps \\
\textbf{sig}\ U\ \{\ rsU\ \} \\
\textbf{sig}\ S\ \textbf{extends}\ U\{rsS\} \\
\textbf{sig}\ T\ \textbf{extends}\ U\{rsT\} \\
\textbf{fact}\ F\ \{forms\} \\
\\
\\
\\
\\
\hline
\end{array}
=_{\Sigma,v}
\begin{array}{|l|}
\hline
ps \\
\textbf{sig}\ U\ \{rsU\} \\
\textbf{sig}\ S\ \textbf{extends}\ U\{rsS\} \\
\textbf{sig}\ T\ \textbf{extends}\ U\{rsT\} \\
\textbf{sig}\ X\ \textbf{extends}\ U\{\} \\
\textbf{fact}\ F\ \{ \\
\quad forms \\
\quad X = U - S - T \\
\} \\
\hline
\end{array}
$$

provided
($\leftrightarrow$) if X belongs to Σ, v contains the $X = (U - S - T)$ item;
($\rightarrow$) (1) ps does not declare any paragraph named X; (2) there is no signature in ps that extends U (3) for all names in Σ that are not on the right side model, v must have exactly one valid item for it;
($\leftarrow$) X does not appear in ps, rsU, rsS, rsT and $forms$; (2) there is no expression exp, where $exp \leq U$ and $exp \leq S$ and $exp \leq T$, in ps or $forms$ or any valid item in v; (3) for all names in Σ that are not on the left side model, v must have exactly one valid item for it.

Law 7. ⟨*introduce signature*⟩

```
ps
```

$=_{\Sigma,v}$

```
ps
sig S {}
```

provided
(↔) (1) S is not in Σ; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it;
(→) *ps* does not declare any paragraph named S;
(←) S does not appear in *ps*.

Law 8. ⟨*introduce generalization*⟩

```
ps
sig S {
   rs
}
sig T {
   rs'
}
fact F {
   forms
}
```

$=_{\Sigma,v}$

```
ps
sig U {}
sig S extends U {
   rs
}
sig T extends U {
   rs'
}
fact F {
   forms
   U = S + T
}
```

provided
(↔) (1) if U belongs to Σ, v contains the $U \rightarrow S + T$ item; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it;
(→) *ps* does not declare any paragraph named U;
(←) U does not appear in *ps*, *rs*, *rs'* or *forms*.

Law 9. ⟨*remove abstract qualifier*⟩

```
ps
abstract sig S {
  rsS
}
sig T extends S {
  rsT
}
sig U extends S {
  rsU
}
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
  rsS
}
sig T extends S {
  rsT
}
sig U extends S {
  rsU
}
fact F {
  forms
  S = T + U
}
```

provided
(→) there is no signature X in *ps* that extends S.

Law 10. ⟨*remove signature cardinality qualifier*⟩

```
ps
x sig S {
  rs
}
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
  rs
}
fact F {
  forms
  x S
}
```

where
x ∈ { **one**, **lone**, **some** }.

Law 11. ⟨*separate signature declarations*⟩

```
ps
sig S, T {
   rs
}
```

$=_{\Sigma,v}$

```
ps
sig S {
   rs
}
sig T {
   rs
}
```

A.2 Relations

Law 1. ⟨*introduce relation and its definition*⟩

```
ps
sig S {
   rs
}
fact F {
   forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
   rs,
   r : set T
}
fact F {
   forms
   r=exp
}
```

provided
(↔) if r belongs to Σ, r does not appear in *exp* and v contains the $r \rightarrow$ *exp* item;
(→) (1) the family of S does not declare any relation named r; (2) T is either S or declared in *ps*; (3) r does not appear in *exp*, or *exp* is r; (4) *exp* is a subtype of r in *ps* and *forms*; (5) for all names in Σ that are not on the right-hand side model, v must have exactly one valid item for it;
(←) (1) r is not mentioned in any constraints within *ps*; (2) for all names in Σ that are not on the left-hand side model, v must have exactly one valid item for it.

Law 12. ⟨*separate relation declarations*⟩

```
ps
sig S {
   rs,
   r, s : set T
}
```

$=_{\Sigma,v}$

```
ps
sig S {
   rs,
   r : set T,
   s : set T
}
```

A.3 Formulas

Law 13. ⟨*replace relation expression*⟩

```
ps
sig S {
   rs,
   r :  set exp
}
fact F {
   forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
   rs,
   r : set T
}
fact F {
   forms
   all s                            :
S  |  s.r in exp'
}
```

where
exp′ replaces each reference to a relation x of S (whether declared or inherited) not prefixed by `@` by $s.x$. Every occurrence of **this** must be replaced by s.
provided
(↔) *exp* has the type T, which is a signature declared in *ps* or is S;
(←) r does not appear in *exp*.

Law 14. ⟨*introduce formula*⟩

```
ps
fact F {
   forms
}
```

$=_{\Sigma,v}$

```
ps
fact F {
   forms
   f
}
```

provided
(↔) f can be deduced from the formulae in *ps* and *forms*.

Law 15. ⟨*introduce empty fact*⟩

```
ps
```

$=_{\Sigma,v}$

```
ps
fact F {}
```

provided
(→) *ps* does not declare any paragraph named F.

A.4 Relations

Law 16. ⟨*split relation*⟩

```
ps
sig S {
   rs,
   r : set T
}
fact F {
   forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
   rs,
   r : set T
}
sig U {
   x : set S,
   y : set T
}
fact F {
   forms
   r = ~x.y
}
```

provided
(↔) (1) U, x and y do not belong to Σ; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it;
(→) *ps* does not declare any paragraph named U;
(←) U, x and y do not appear in *ps*, *rs* or *forms*.

Law 17. ⟨*remove one relation*⟩

```
ps
sig S {
   rs,
   r : one T
}
fact F {
   forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
   rs,
   r : set T
}
fact F {
   forms
   all s : S | one s.r
}
```

Law 18. ⟨*remove lone relation*⟩

```
ps
sig S {
   rs,
   r : lone T
}
fact F {
   forms
}
```

$=_{\Sigma,v}$

```
ps
sig S {
   rs,
   r : set T
}
fact F {
   forms
   all s : S | lone s.r
}
```

Appendix B

BN Laws and Refactorings

The laws of programming used in strategies are presented here, in addition to some refactoring rules that are composed of primitive laws. This catalog is mainly taken from Cornélio's thesis [26], with syntax changes that adhere to the BN language. The parameters used in law applications within strategies are taken from the template declarations (for instance, the class subject to the indicated transformation).

B.1 if statements

Law 6. $\langle$*eliminate redundant if*$\rangle$
$\textbf{if}\ \psi\ \textbf{then}\ \mathit{cmd}\ \textbf{else}\ \mathit{cmd} = \mathit{cmd}$

Law 19. $\langle$**if** *true guard*$\rangle$
$\textbf{if}\ (\textbf{true})\ \textbf{then}\ c = c$

Law 20. $\langle$*assumption guard*$\rangle$
$\textbf{if}\ (e)\ \textbf{then}\ S_1\ \textbf{else}\ S_2 = \textbf{if}\ (e)\ \textbf{then}\ \{e\}S_1\ \textbf{else}\ \{\neg e\}S_2$

Law 21. $\langle$*absorb assumption*$\rangle$
$\{\phi\}\ \textbf{if}\ (\psi_i)\ \textbf{then}\ c_i \quad = \textbf{if}\ (\phi \wedge \psi_i)\ \textbf{then}\ c_i$

B.2 Variable blocks

Law 22. ⟨*var block*-**val**⟩

$$c = T\ v\ \bullet\ v := x;\ c[v/x]$$

provided v is fresh—not free in c—; x is not on the left-hand side of assignments, it is not a result value, x is not a method call target, and $x \neq \mathbf{error}$.

Law 23. ⟨*var block*-**result**⟩

$$c = T\ v\ \bullet\ c[v/x];\ x := v$$

provided v is fresh—not free in c—; x is not on the right-hand side of assignments and it is not an argument nor a method call target, x is not used in field updates.

Law 24. ⟨*pcom elimination*-**result**⟩

$$(T\ \mathbf{result}\ \bullet c)(x) = T\ l \bullet c[l/\mathbf{result}];\ x := l$$

provided the variables of l are fresh: not free in c, x. Variables in l are not on the right-hand side of assignments, they are not used as arguments nor are method call targets, and they are not used in field updates.

Law 25. ⟨**if** *identical guarded commands*⟩
If $(\bigvee i : 1..n\ \bullet\ \psi_i = \mathbf{true})$, then
$\mathbf{if}_i(\psi_i)\ \mathbf{then}\ c = c$

B.3 Classes, Fields and Methods

Law 3. $\langle$*class elimination*$\rangle$

$$CT\ cd_1 \ = \ CT$$

provided

($\leftrightarrow$) $cd_1 \neq$ `Main`;

($\rightarrow$) $name(cd_1)$ is not used in CT.

($\leftarrow$) (1) cd_1 is a distinct name; (2) Field, method and superclass types in cd_1 are declared in CT.

Law 4. $\langle$*new superclass*$\rangle$

$$\boxed{\begin{array}{l} \mathbf{class}\ A\ \mathbf{extends}\ C\ \{ \\ \quad ads \\ \quad mts \\ \} \\ \mathbf{class}\ B\ \mathbf{extends}\ A\ \{\ \} \\ CT \end{array}} \ =_{CT}\ \boxed{\begin{array}{l} \mathbf{class}\ A\ \mathbf{extends}\ C\ \{ \\ \quad ads \\ \quad mts' \\ \} \\ \mathbf{class}\ B\ \mathbf{extends}\ A\ \{\ \} \\ CT' \end{array}}$$

where

$CT' = CT[\mathbf{new}\ A/\mathbf{new}\ B]$

$mts' = mts[\mathbf{new}\ A/\mathbf{new}\ B]$

provided

($\rightarrow$) (1) B is not used in type casts or tests in CT or *ops* for expressions of type A; (2) $x := \mathbf{new}\ B$ only appears if $type(x) \leq A$.

($\leftarrow$) Variables of type $T \leq A$ are not involved in tests with type B.

Law 5. ⟨*distribute type tests*⟩

Consider the following class declarations, where $C_i, i = 1..n$ encompasses all subclasses of B in CT.

class B **extends** A {
  fds;
  mts
}
class C_i **extends** B {
  fds_i;
  mts_i
}

Then, for any command *cmd* in CT:
$cmd[exp$ **is** $B] = cmd[\bigvee_i (exp$ **is** $C_i)]$

provided

$(\leftrightarrow)$ $x :=$ **new** B does not appear in CT, *mts* or mts_i

Law 26. ⟨*change superclass: from an empty class to immediate superclass*⟩

$$\boxed{\begin{array}{l} \textbf{class}\ B\ \textbf{extends}\ A\{\ \} \\ \textbf{class}\ C\ \textbf{extends}\ B\{ \\ \quad fdsC \\ \quad mtsC \\ \} \end{array}} =_{CT} \boxed{\begin{array}{l} \textbf{class}\ B\ \textbf{extends}\ A\{\ \} \\ \textbf{class}\ C\ \textbf{extends}\ A\{ \\ \quad fdsC \\ \quad mtsC \\ \} \end{array}}$$

provided

$(\rightarrow)$ (1) C or any of its subclasses in CT is not used in type casts or tests involving expressions of type B;

(2) There are no assignments of the form $le := exp$, for any le whose declared type is B or any of its superclasses and the type of exp is C or any subclass of C;

(3) Expressions of type C or of any subclass of C are not used as arguments in calls with a corresponding formal value parameter whose type is B;

(4) Expressions whose declared type is B are not returned values in calls with a corresponding formal return type C or any subclass of C.

$(\leftarrow)$ (1) Casts to class B are not applied to fields, variables or parameters of type A to which are assigned expressions of type C.

Law 27. ⟨*change visibility: from private to public*⟩

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad T\ \ a;\ fds\\ \quad mts\\ \}\end{array}} =_{CT} \boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad \textbf{pub}\ T\ \ a;\ fds\\ \quad mts\\ \}\end{array}}$$

provided

(←) $B.a$, for any $B \leq C$, does not appear in CT, c.

Law 28. ⟨*field elimination*⟩

$$\boxed{\begin{array}{l}\textbf{class}\ B\ \textbf{extends}\ A\{\\ \quad T\ \ a;\ \ fds\\ \quad mts\\ \}\end{array}} =_{CT} \boxed{\begin{array}{l}\textbf{class}\ B\ \textbf{extends}\ A\{\\ \quad fds\\ \quad mts\\ \}\end{array}}$$

provided

(→) $B.a$ does not appear in *mts*;

(←) a does not appear in *fds* and is not declared as an field by a superclass or subclass of B in CT.

Law 29. ⟨*change field type*⟩

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad \textbf{pub}\ T\ \ a;\ fds\\ \quad mts\\ \}\end{array}} =_{CT} \boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad \textbf{pub}\ T'\ \ a;\ fds\\ \quad mts\\ \}\end{array}}$$

provided

(↔) $T \leq T'$ and every non-assignable occurrence of a in expressions of *mts*, CT and c is cast with T or any subtype of T declared in CT.

(←) (1) every expression assigned to a, in *mts*, CT and c, is of type T or any subtype of T; (2) every use of a as return value is for a corresponding formal parameter of type T or any subtype of T.

Law 30. ⟨*change variable type*⟩
$CT, A \rhd T\ x\ \bullet c = T'\ x\ \bullet c$

provided

($\leftrightarrow$) $T \leq T'$ and every non-assignable occurrence of x in expressions of c is cast with T or any subtype of T;

($\leftarrow$) (1) every expression assigned to x in c is of type T or any subtype of T; (2) every use of x as return value in c is for a corresponding return type T or any subtype of T.

Law 31. ⟨*change parameter type*⟩

```
class C extends D{                 class C extends D{
   fds                                fds
   Z m(T x, Z̄ p̄){ b }       =CT      Z m(T' x, Z̄ p̄){ b }
   mts                                mts
}                                  }
```

provided

($\leftrightarrow$) $T \leq T'$ and every non-assignable occurrence of x in expressions of b are cast with T or any subtype of T;

($\leftarrow$) (1) every argument associated with x in *mts*, *cds*, and c is of type T or any subtype of it; (2) every expression assigned to x in b, is of type T or any subtype of T; (3) every use of x as return value in b is for a corresponding return type T or any subtype of T.

Law 32. $\langle$*change return type*$\rangle$

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad fds\\ \quad T\ \ m(\bar{T}\ \ \bar{x})\{\ \ b\ \ \}\\ \quad mts\\ \}\end{array}} =_{CT} \boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad fds\\ \quad T'\ \ m(\bar{T}\ \ \bar{x})\{\ \ b\ \ \}\\ \quad mts\\ \}\end{array}}$$

provided

($\leftrightarrow$) $T \leq T'$ and every non-assignable occurrence of x in expressions of b are cast with T or any subtype of T;

($\rightarrow$) every argument associated with formal parameter x in mts, cds, and c is of type T' or any supertype of it;

($\leftarrow$) (1) every expression assigned to x in b is of type T or any subtype of T; (2) every use of x as return value in b is for a corresponding return type T or any subtype of T.

Law 33. $\langle$*method call elimination*$\rangle$
Consider that the following class declaration

$$\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad fds\\ \quad T\ \ m()\{pc\}\\ \quad mts\\ \}\end{array}$$

is included in CT and $CT, A \triangleright le : C$. Then

$$CT, A \triangleright le.m(e) \ = \ \{le \neq \textbf{null} \wedge le \neq \textbf{error}\};\ pc[le/\textbf{self}](e)$$

provided

($\leftrightarrow$) (1) m is not redefined in CT and pc does not contain references to **super**; (2) all fields which appear in the body pc of m are public.

Law 34. ⟨*method elimination*⟩

```
class C extends D{          |        | class C extends D{
   fds                      |        |    fds
   T m { pc }               |  =_CT  |    mts
   mts                      |        | }
}                           |        |
```

provided

(→) $B.m$ does not appear in CT nor in mts, for any B such that $B \leq C$.

(←) m is not declared in mts nor in any superclass or subclass of C in CT.

Law 35. ⟨*move redefined method to superclass*⟩

```
class B extends A {         |         | class B extends A {
   ads                      |         |    ads
   T m(T̄ x̄) {S}             |         |    T m(T̄ x̄) {
   mts                      |         |       if ¬(self is C)
}                           |         |          then S
class C extends B {         |         |          else S'
   ads'                     | =_cds,  |    }
   T m(T̄ x̄) {S'}            |         |    mts
   mts'                     |         | }
}                           |         | class C extends B {
                            |         |    ads'
                            |         |    mts'
                            |         | }
```

provided

(↔) (1) **super** and private fields do not appear in S'; (2) **super**.m do not appear in mts';

(→) S' does not contain uncast ocurrences of **self**;

(←) m is not declared in mts'.

Law 36. ⟨*move original method to superclass*⟩

$$
\begin{array}{|l|}\hline
\textbf{class}\ B\ \textbf{extends}\ A\{\\
\quad \mathit{fds}\\
\quad \mathit{mts}\\
\}\\
\textbf{class}\ C\ \textbf{extends}\ B\{\\
\quad \mathit{fds}'\\
\quad T\ m(\bar{T}\ \bar{x})\{\ pc\ \}\\
\quad \mathit{mts}'\\
\}\\ \hline
\end{array}
=_{CT}
\begin{array}{|l|}\hline
\textbf{class}\ B\ \textbf{extends}\ A\{\\
\quad \mathit{fds}\\
\quad T\ m(\bar{T}\ \bar{x})\{\ pc\ \}\\
\quad \mathit{mts}\\
\}\\
\textbf{class}\ C\ \textbf{extends}\ B\{\\
\quad \mathit{fds}'\\
\quad \mathit{mts}'\\
\}\\ \hline
\end{array}
$$

provided

($\leftrightarrow$) (1) **super** and private fields do not appear in *pc*; (2) m is not declared in any superclass of B in CT;

($\rightarrow$) (1) m is not declared in *mts*, and can only be declared in a class D, for any $D \leq B$ and $D \not\leq C$, if it has the same parameters as *pc*; (2) *pc* does not contain uncast occurrences of **self** nor expressions in the form $((C)\mathbf{self}).a$ for any private field a in *fds*$'$;

($\leftarrow$) (1) m is not declared in *mts*$'$; (2) $D.m$, for any $D \leq B$, does not appear in CT, c, *mts* or *mts*$'$.

Law 37. ⟨*move field to superclass*⟩

$$
\begin{array}{|l|}\hline
\textbf{class}\ B\ \textbf{extends}\ A\{\\
\quad \mathit{fds}\\
\quad \mathit{mts}\\
\}\\
\textbf{class}\ C\ \textbf{extends}\ B\{\\
\quad \textbf{pub}\ T\ a;\ \mathit{fds}'\\
\quad \mathit{mts}'\\
\}\\ \hline
\end{array}
=_{CT}
\begin{array}{|l|}\hline
\textbf{class}\ B\ \textbf{extends}\ A\{\\
\quad \textbf{pub}\ T\ a;\ \mathit{fds}\\
\quad \mathit{mts}\\
\}\\
\textbf{class}\ C\ \textbf{extends}\ B\{\\
\quad \mathit{fds}'\\
\quad \mathit{mts}'\\
\}\\ \hline
\end{array}
$$

provided

($\rightarrow$) The field name a is not declared by the subclasses of B in CT;

($\leftarrow$) $D.a$, for any $D \leq B$ and $D \not\leq C$, does not appear in CT, c, *mts*, or *mts*$'$.

Law 38. ⟨*eliminate* **super**⟩
Consider that *CDS* is a set of two class declarations as follows.

class B **extends** $A\{$
  fds
  $T\ \ m\ \ (\bar{T}\ \ \bar{x})\ \ \{\ \ pc\ \ \}$
  mts
$\}$

class C **extends** $B\{$
  fds'
  mts'
$\}$

Then we have that

$$CT\ \ CDS,\ \ C \rhd \textbf{super}.m = pc$$

provided

(→) **super** and the private fields in *fds* do not appear in *pc*.

B.4 Type casts and tests

Law 39. ⟨*eliminate cast of expressions*⟩
If $CT, A \rhd le : B,\ e : B',\ C \leq B'$ and $B' \leq B$, then

$$CT, A \rhd le := (C)e = \{e \ \textbf{is}\ C\} le := e$$

Law 40. ⟨*introduce trivial cast in expressions*⟩
If $CT, A\ \rhd\ e : C$, then $CT, A\ \rhd\ e = (C)e$.

For simplicity, this is formalized as a law of expressions, not commands. Nevertheless, it should be considered as an abbreviation for several laws of assignments, conditionals, and method calls that deal with each possible pattern of expressions. For example, it abbreviates the following laws, all with the same antecedent as Law 40.

$$CT, A\ \rhd\ le := e.x = le := ((C)e).x$$
$$CT, A\ \rhd\ e'.m(e) = e'.m((C)e)$$

This is equally valid for left-expressions, which are a particular form of expression.

Law 41. ⟨**is** *test true*⟩
If $N \leq_{CT} M$, then $CT, N \rhd$ **self is** $M =$ **true**

B.5 Refactorings

Refactoring 2. ⟨*distribute casts*⟩

Consider the following class declarations, where $C_i, i = 1..n$ encompasses all subclasses of B in CT.

class B **extends** A {
  fds;
  mts
}
class C_i **extends** B {
  fds_i;
  mts_i
}

Then, for any command *cmd* in CT:
$cmd[(B)exp] =$ $\textbf{if}_i$ $(exp$ **is** $C_i)$ **then** $cmd[(C_i)exp]$

provided

(↔) $x :=$ **new** B does not appear in CT, *mts* or mts_i

Refactoring 3. ⟨*method elimination-abstract class*⟩

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad fds\\ \quad T\ \ m(\bar{T}\ \ \bar{x})\ \ \{\ \ cmd\ \ \}\\ \quad mts\\ \}\end{array}} =_{CT} \boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\{\\ \quad fds\\ \quad mts\\ \}\\ \end{array}}$$

provided

(↔) $x :=$ **new** C does not appear in CT, *mts*, *mts*′ or *cmd*;

(→) m is redefined in all subclasses of C in CT;

(←) m is not declared in *mts* nor in any superclass or subclass of C in CT.

Refactoring 4. $\langle$*self-encapsulate field*$\rangle$

```
class A extends B{
  T x; fdsA
  Z m (Z̄ x̄){
    cmd[le := exp1[self.x];
    self.x := exp2]); c1
  }
  mtsA
}
```

$=_{CT}$

```
class A extends B{
  T x; fdsA
  Z m(Z̄ x̄){ c1′ }
  T getX(){
    result := self.x
  }
  unit setX(T arg){
    self.x := arg
  }
  mtsA
}
```

where

$c1' = c1[T\ aux\ \bullet\ aux := \textbf{self}.getX();\ le := exp1[aux];$
$\quad \textbf{self}.setX(exp2)/le := exp1[\textbf{self}.x], \textbf{self}.x := exp2]$

$mtsA' = mtsA[T\ aux\ \bullet\ aux := \textbf{self}.getX();\ le := exp1[aux];$
$\quad \textbf{self}.setX(exp2)/le := exp1[\textbf{self}.x], \textbf{self}.x := exp2]$

provided

($\rightarrow$) *getX*, *setX* are not declared in any superclass or subclass of A in CT;

($\leftarrow$) *le.getX* and *le.setX* do not appear in $mtsA'$ or CT for any *le* such that $le \leq A$.

Refactoring 5. ⟨*pull up/push down field*⟩

```
class A extends D{          class A extends D{
  fdsA                        pub T z; fdsA
  mtsA                        mtsA
}                           }
class B extends A{          class B extends A{
  pub T x; fdsB               fdsB
  mtsB               =CT      mtsB'
}                           }
class C extends A{          class C extends A{
  pub T y; fdsC               fdsC
  mtsC                        mtsC'
}                           }
cds1                        cds1'
```

where

$mtsB' = mtsB[z/x]$

$mtsC' = mtsC[z/y]$

$cds1' = cds1[M.z, N.z/M.x, N.y]$, for any $M \leq B$ and $N \leq C$

provided

($\rightarrow$) CT contains no subclasses of B and C in which there are references to x and y;

$N.x$, for any $N \leq B$, does not appear in CT, and $N.y$, for any $N \leq C$, does not appear in CT;

$cds1$ contains only subclasses of B and C in which there are references to x and y;

($\leftarrow$) The field name z is not declared in $fdsA, fdsB, fdsC$, nor in any subclass or superclass of A in CT;

$N.z$, for any $N \leq A$, $N \nleq B$ and $N \nleq C$, does not appear in CT.

($\rightarrow$) $x(y)$ is not declared in $fdsA, fdsB(fdsC)$, nor in any subclass or superclass of $B(C)$ in CT and $cds1'$.

Appendix C

Proofs for Laws of Programming with References

In this appendix we show the proofs developed for four laws of programming from the ROOL catalog [26], but using the denotational semantics defined for BN [8], which includes pointers. These proofs do not use any refinement theory for pointers; they only increase confidence in laws that are intuitively correct in this context.

C.1 Class Elimination

Law $\langle$*class elimination*$\rangle$

$CT\ cd_1 \ = \ CT$

provided

($\leftrightarrow$) $cd_1 \neq Main$;

($\rightarrow$) cd_1 is not used in CT.

($\leftarrow$) (1) cd_1 is a distinct name; (2) Field, method and superclass types in cd_1 are declared in CT.

We now establish the theorem to be proved for the Law *class elimination*.

Theorem C.1. *Let CT and CT' be two class tables, representing, respectively, the left-hand and right-hand sides of Law classelimination, with $Own = \{\}$. Then, for any heap h_0 and store η_0 in CT, and any command S in the main method:*

$$\begin{array}{l} \forall\, \mu : [\![\ CT\ ]\!] \bullet \\ collect([\![\ \Gamma \vdash S\ ]\!]\ \mu(h_0, \eta_0)) = collect([\![\ \Gamma \vdash S\ ]\!]\ \mu(h_0', \eta_0')) \end{array}$$

where

$h_0' = h_0 \downarrow \ell \in \ locs\ \ cd_1$

$\eta_0' = \eta_0 \downarrow \ell \in \; locs \; cd_1$

Proof C.1. .By induction. Here, we consider only the case of field assignment; all other commands have similar proof.

Case $e_1.f := e_2$.
$collect([\![\Gamma \vdash e_1.f := e_2]\!] \, \mu(h_0, \eta_0))$

$=$ [by semantics of field assignment]
  $let \; \ell =[\![\Gamma \vdash e_1 : (\Gamma \mathbf{self})]\!] \, (h_0, \eta_0) \; in$
    $if \; \ell = nil \; \bot \; else$
    $let \; d =[\![\Gamma \vdash e_2 : U]\!] \, (h_0, \eta_0) \; in$
    $([h \mid \ell \mapsto [h_0 \ell \mid f \mapsto d]], \eta_0)$

$=$ [by law assumption, $\nexists \ell : locs \; cd_1 \bullet \ell \in dom h_0 \vee \ell \in dom \eta_0$, then $h_0 = h_0'$ and $\eta_0 = \eta_0'$]
  $let \; \ell =[\![\Gamma \vdash e_1 : (\Gamma \mathbf{self})]\!] \, (h_0', \eta_0') \; in$
    $if \; \ell = nil \; \bot \; else$
    $let \; d =[\![\Gamma \vdash e_2 : U]\!] \, (h_0', \eta_0') \; in$
    $([h \mid \ell \mapsto [h_0' \ell \mid f \mapsto d]], \eta_0')$

$=$ [by semantics of field assignment]
  $collect([\![\Gamma \vdash e_1.f := e_2]\!] \, \mu(h_0', \eta_0'))$

$\square$

C.2 Assumption Guard

Law $\langle$*assumption guard*$\rangle$

if (e) **then** S_1 **else** $S_2 =$ **if** (e) **then** $\{e\}S_1$ **else** $\{\neg e\}S_2$

We now establish the theorem to be proved for the Law *assumptionguard*.

Theorem C.2. *For any heap* h_0 *and store* η_0*, and well-typed commands* S_1*,* S_2 *and boolean expression* e*:*

$\forall \mu :[\![\; CT \;]\!] \bullet$
$[\![\Gamma \vdash \mathbf{if} \; (e) \; \mathbf{then} \; S_1 \; \mathbf{else} \; S_2]\!] \, \mu(h_0, \eta_0) =$
$[\![\Gamma \vdash \mathbf{if} \; (e) \; \mathbf{then} \; \{e\}S_1 \; \mathbf{else} \; \{\neg e\}S_2]\!] \, \mu(h_0, \eta_0)$

Proof C.2. .Direct proof.

$[\![\Gamma \vdash \mathbf{if} \; (e) \; \mathbf{then} \; S_1 \; \mathbf{else} \; S_2]\!] \, \mu(h_0, \eta_0)$

= [by semantics of if]
$\quad$ $let\ b = [\![\Gamma \vdash e : bool]\!]\ (h_0, \eta_0)\ in$
$\quad\quad$ $if\ (b)\ then\ [\![\Gamma \vdash S_1]\!]\ \mu(h_0, \eta_0)\ else\ [\![\Gamma \vdash S_2]\!]\ \mu(h_0, \eta_0)$

= [by characteristics of semantic if]
$\quad$ $let\ b = [\![\Gamma \vdash e : bool]\!]\ (h_0, \eta_0)\ in$
$\quad\quad$ $if\ (b)\ then\ (if\ (b)\ then\ [\![\Gamma \vdash S_1]\!]\ \mu(h_0, \eta_0))$
$\quad\quad$ $else\ (if\ (\neg b)\ then\ [\![\Gamma \vdash S_2]\!]\ \mu(h_0, \eta_0))$

= [we add an else that can never be executed]
$\quad$ $let\ b = [\![\Gamma \vdash e : bool]\!]\ (h_0, \eta_0)\ in$
$\quad\quad$ $if\ (b)\ then\ (if\ (b)\ then\ [\![\Gamma \vdash S_1]\!]\ \mu(h_0, \eta_0)\ else\ \bot)$
$\quad\quad$ $else\ (if\ (\neg b)\ then\ [\![\Gamma \vdash S_2]\!]\ \mu(h_0, \eta_0)\ else\ \bot)$

= [repeating definition of b]
$\quad$ $let\ b = [\![\Gamma \vdash e : bool]\!]\ (h_0, \eta_0)\ in$
$\quad\quad$ $if\ (b)\ then\ (let\ b = [\![\Gamma \vdash e : bool]\!]\ (h_0, \eta_0)\ in$
$\quad\quad\quad$ $if\ (b)\ then\ [\![\Gamma \vdash S_1]\!]\ \mu(h_0, \eta_0)\ else\ \bot)$
$\quad\quad$ $else\ (let\ b = [\![\Gamma \vdash e : bool]\!]\ (h_0, \eta_0)\ in$
$\quad\quad\quad$ $if\ (\neg b)\ then\ [\![\Gamma \vdash S_2]\!]\ \mu(h_0, \eta_0)\ else\ \bot)$

= [syntactic sugar]
$\quad$ $let\ b = [\![\Gamma \vdash e : bool]\!]\ (h_0, \eta_0)\ in$
$\quad\quad$ $if\ (b)\ then\ [\![\Gamma \vdash \{b\} S_1]\!]\ \mu(h_0, \eta_0)\ else\ [\![\Gamma \vdash \{\neg b\} S_2]\!]\ \mu(h_0, \eta_0)$

= [by semantics of if]
$\quad$ $[\![\Gamma \vdash \mathbf{if}\ (e)\ \mathbf{then}\ \{e\} S_1\ \mathbf{else}\ \{\neg e\} S_2]\!]\ \mu(h_0, \eta_0)$

$\square$

C.3 Change Field Type

Law ⟨*change field type*⟩

```
class C extends D{
   pub T a; fds
   mts
}
```

$=_{CT}$

```
class C extends D{
   pub T' a; fds
   mts
}
```

provided

(↔) $T \leq T'$ and every non-assignable occurrence of a in expressions of mts, CT and c is cast with T or any subtype of T declared in CT.

(←) (1) every expression assigned to a, in mts, CT and c, is of type T or any subtype of T; (2) every use of a as return value is for a corresponding formal parameter of type T or any subtype of T.

Theorem C.3. *As the main corollary of the abstraction theorem proved by Banerjee and Naumann [8], if there is a simulation for CT and CT' in confined $Own = \{C\}$, then $CT, (\Gamma \vdash S)$ is equivalent to $CT', (\Gamma \vdash' S)$, for any command S in the main method. Therefore, we only need to prove the simulation for class C. So, for any heap h_0, store η_0:*

$$\begin{array}{l}\forall\, \mu : [\![\ CT\ ]\!], \mu' : [\![\ CT'\ ]\!] \bullet \\ \exists\, R \bullet (\forall\, c : constructorsC \bullet R(\mu\ Cc)(\mu'\ Cc)) \wedge \\ \forall\, m : methsC \bullet R(h_0, \eta_0)(h'_0, \eta'_0) \Rightarrow R(\mu\ Cm(h_0, \eta_0))(\mu'\ Cm(h'_0, \eta'_0)))\end{array}$$

Proof C.3. . By the law, we conclude that heap will not change (since all expressions assigned to field a are cast with type T or have type T). Thus, we will prove that the following coupling is a simulation:

$$R_1(h, \eta)(h', \eta') = h = h' \ \wedge\ \eta = \eta'$$

$\exists\, R \bullet (\forall\, c : constructorsC \bullet R(\mu\ Cc)(\mu'\ Cc)) \wedge$
$\forall\, m : methsC \bullet R(h_0, \eta_0)(h'_0, \eta'_0) \Rightarrow R(\mu\ Cm(h_0, \eta_0))(\mu'\ Cm(h'_0, \eta'_0)))$

$=$ [$\exists$-elimination, using R_1]
$\quad \forall\, c : constructorsC \bullet R_1(\mu\ Cc)(\mu'\ Cc)) \wedge$
$\quad \forall\, m : methsC \bullet R_1(h_0, \eta_0)(h'_0, \eta'_0) \Rightarrow R_1(\mu\ Cm(h_0, \eta_0))(\mu'\ Cm(h'_0, \eta'_0))$

$=$ [using arbitrary constructor c and method m in class C]
$\quad R_1(\mu\ Cc)(\mu'\ Cc)) \wedge$
$\quad R_1(h_0, \eta_0)(h'_0, \eta'_0) \Rightarrow R_1(\mu\ Cm(h_0, \eta_0))(\mu'\ Cm(h'_0, \eta'_0))$

$=$ [from $R_1(h_0,\eta_0)(h_0',\eta_0')$, $h_0 = h_0' \wedge \eta_0 = \eta_0'$]
$R_1(\mu\ Cc)(\mu'\ Cc)) \wedge$
$R_1(h_0,\eta_0)(h_0',\eta_0') \Rightarrow R_1(\mu\ Cm(h_0,\eta_0))(\mu'\ Cm(h_0,\eta_0))$

$=$ [no method changes, thus $\mu = \mu'$]
$R_1(\mu\ Cc)(\mu\ Cc)) \wedge$
$R_1(h_0,\eta_0)(h_0',\eta_0') \Rightarrow R_1(\mu\ Cm(h_0,\eta_0))(\mu\ Cm(h_0,\eta_0))$

$=$ [application of R_1]
true

□

C.4 Move Redefined Method to Superclass

Law ⟨*move redefined method to superclass*⟩

```
class B extends A {
   ads
   T m(T̄ x̄) {S}
   mts
}
class C extends B {
   ads'
   T m(T̄ x̄) {S'}
   mts'
}
```

$=_{CT,}$

```
class B extends A {
   ads
   T m(T̄ x̄) {
      if ¬(self is C)
         then S
         else S'
   }
   mts
}
class C extends B {
   ads'
   mts'
}
```

provided

(↔) (1) **super** and private fields do not appear in S'; (2) **super**.m do not appear in *mts*$'$;

(→) S' does not contain uncast occurrences of **self**;

(←) m is not declared in *mts*$'$.

Theorem C.4. *As the main corollary of the abstraction theorem proved by Banerjee and Naumann [8], if there is a simulation for CT and CT' in confined $Own = \{B, C\}$, then $CT, (\Gamma \vdash S)$ is equivalent to $CT', (\Gamma \vdash' S)$, for any command S in the main method. Therefore, we only need to prove the simulation for classes B and C. So, for any heap h_0, store η_0:*

$\forall\, \mu :[\![\ CT\ ]\!], \mu' :[\![\ CT'\ ]\!] \bullet$
$\exists\, R \bullet (\forall\, c : \mathit{constructorsB}, C \bullet R(\mu\, Cc)(\mu'\, Cc)) \;\wedge$
$\forall\, m : \mathit{methsB}, C \bullet R(h_0, \eta_0)(h_0', \eta_0') \Rightarrow R(\mu\, Cm(h_0, \eta_0))(\mu'\, Cm(h_0', \eta_0')))$

Proof C.4. . By the law, we conclude that heap will not change (as no object structures are affected). Thus, we will prove that the following coupling is a simulation:
$R_1(h, \eta)(h', \eta') = h = h' \;\wedge\; \eta = \eta'$

$\exists\, R \bullet (\forall\, c : \mathit{constructorsC} \bullet R(\mu\, B, Cc)(\mu'\, B, Cc)) \;\wedge$
$\forall\, m : \mathit{methsB}, C \bullet R(h_0, \eta_0)(h_0', \eta_0') \Rightarrow R(\mu\, B, Cm(h_0, \eta_0))(\mu'\, B, Cm(h_0', \eta_0')))$

= [$\exists$-elimination, using R_1]
$\forall\, c : \mathit{constructorsB}, C \bullet R_1(\mu\, Cc)(\mu'\, Cc)) \;\wedge$
$\forall\, m : \mathit{methsB}, C \bullet R_1(h_0, \eta_0)(h_0', \eta_0') \Rightarrow R_1(\mu\, B, Cm(h_0, \eta_0))(\mu'\, B, Cm(h_0', \eta_0'))$

= [using arbitrary constructor c and method m in class C]
$R_1(\mu\, Cc)(\mu'\, Cc)) \;\wedge$
$R_1(h_0, \eta_0)(h_0', \eta_0') \Rightarrow R_1(\mu\, Bm(h_0, \eta_0))(\mu'\, Cm(h_0', \eta_0'))$

= [from $R_1(h_0, \eta_0)(h_0', \eta_0')$, $h_0 = h_0' \wedge \eta_0 = \eta_0'$]
$R_1(\mu\, Cc)(\mu'\, Cc)) \;\wedge$
$R_1(h_0, \eta_0)(h_0', \eta_0') \Rightarrow R_1(\mu\, Cm(h_0, \eta_0))(\mu'\, Cm(h_0, \eta_0))$

= [the method definitions in B and C are maintained, from the semantics of dynamic binding and the **if** command, thus $\mu = \mu'$]
$R_1(\mu\, Cc)(\mu\, Cc)) \;\wedge$
$R_1(h_0, \eta_0)(h_0', \eta_0') \Rightarrow R_1(\mu\, Cm(h_0, \eta_0))(\mu\, Cm(h_0, \eta_0))$

= [application of R_1]
true

$\square$

Appendix D

Strategies

All strategies developed for this thesis are cataloged in this Appendix, using the Angel [71] notation, as explained in Chapter 6. Law and Rule names are *italic*, while Angel statements and functions are **bold**. For clarification purposes, each strategy includes a figure for providing an overview of the input and output programs for each strategy in UML class diagrams.

D.1 introduceClass

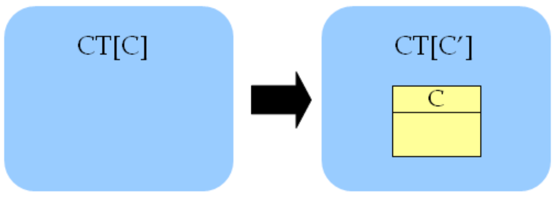


Figure D.1: introduceClass

```
Tactic  $introduceClass(C : Class)$
        (law $rename(C, C')$ | skip);
        law $classElimination(C, \leftarrow)$;
end
```

D.2 removeClass

```
Tactic  $removeClass(C : Class)$
        (law $classElimination(C, \rightarrow)$ | law $rename(C, C', program)$);
end
```

D.3 introduceSuperclass

```
Tactic  $introduceSuperclass(C : Class, < C_1, \ldots, C_n >: classList)$
```

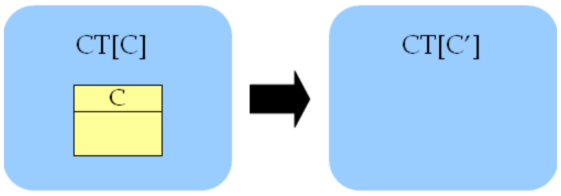


Figure D.2: removeClass

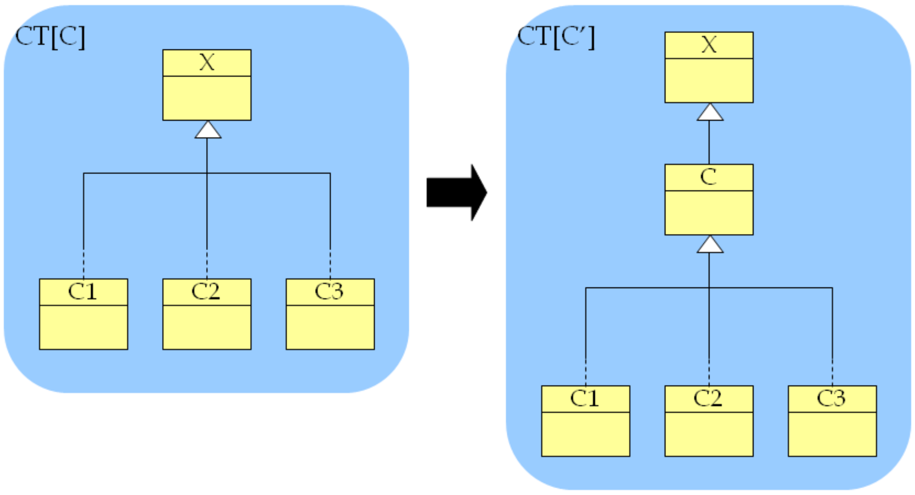


Figure D.3: introduceSuperclass

```
        (law  $rename(C, C')$  |  skip);
        law  $classElimination$(setExtends($C$, firstCommonSuper($\{C_1, \ldots, C_n\}$)), $\leftarrow$);
        law  $changeSuperfromEmptyToImmediateSuperclass$(immedSubs($cc$), $C$, $\leftarrow$);
end
```

D.4 removeSuperclass

```
Tactic  $removeSuperclass(C : Class)$
    law  $changeVisibilityPrivatePublic$(priFields(getHierarchyTopDown($C$)), $\rightarrow$);
    law  $eliminateSuper$(getHierarchyTopDown($C$), $\rightarrow$);
    law  $eliminateSuper$(immedSubs(C), $\rightarrow$);
    applies to  $exp.x, exp.m(\bar{e})$\{isExactly($exp, C$)\} do
        law  $introduceCastToExpression(C, exp, \rightarrow)$;
    Tactic  $changeDeclarationsType(C)$;
    applies to  $((C)e)$  do refactoring  $distributeCasts(\rightarrow)$;
    applies to  ($e$ is $C$)  do law  $distributeTypeTest(\rightarrow)$;
    law  $eliminateRedundantIf$(immedSubs(C), $\leftarrow$);
    (law  $moveRedefinedMethodToSuperclass$(meths($C$), $\leftarrow$)  |  skip);
    law  $methodEliminationAbstractClass$(meths($C$), $\rightarrow$);
```

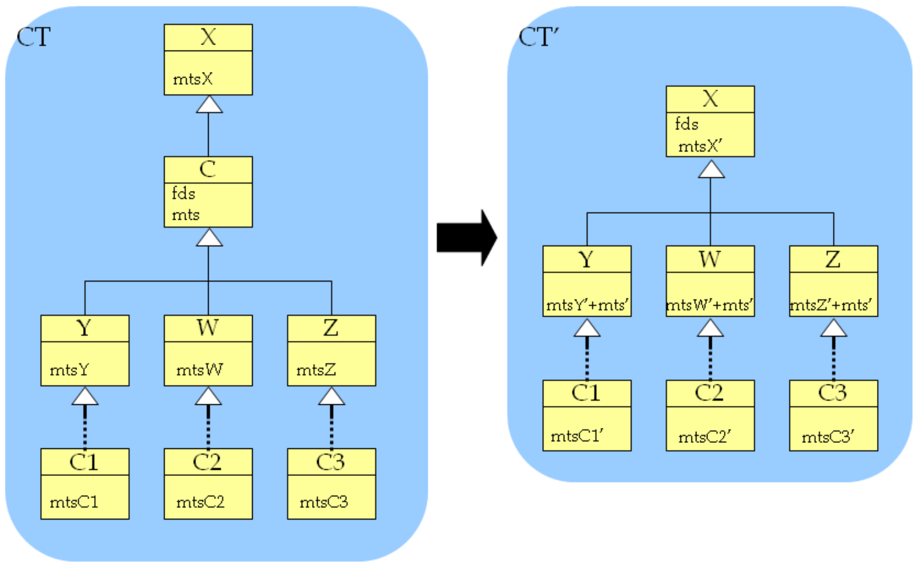


Figure D.4: removeSuperclass

```
    (law  moveFieldToSuperclass(fields(C), →)  |  skip);
    law  changeSuperfromEmptyToImmediateSuperclass(immedSubs(C),
        super(C), ←);
    law  classElimination(C, “”, →);
end
```

D.5 introduceSubclass

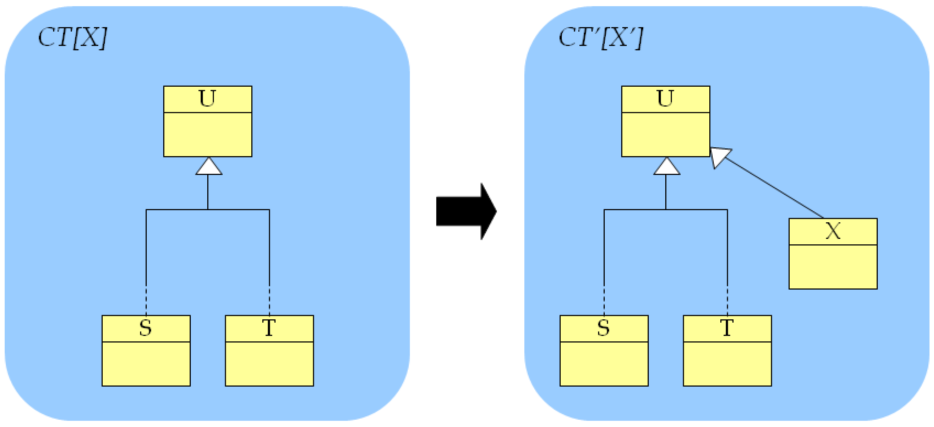


Figure D.5: introduceSubclass

```
Tactic  introduceSubclass(X, U : Class)
        (law  rename(X, X′)  |  skip);
```

law $classElimination(\textbf{setExtends}(X, U), \leftarrow)$;
law $newSuperclass(U, X, \rightarrow)$;
end

D.6 removeSubclass

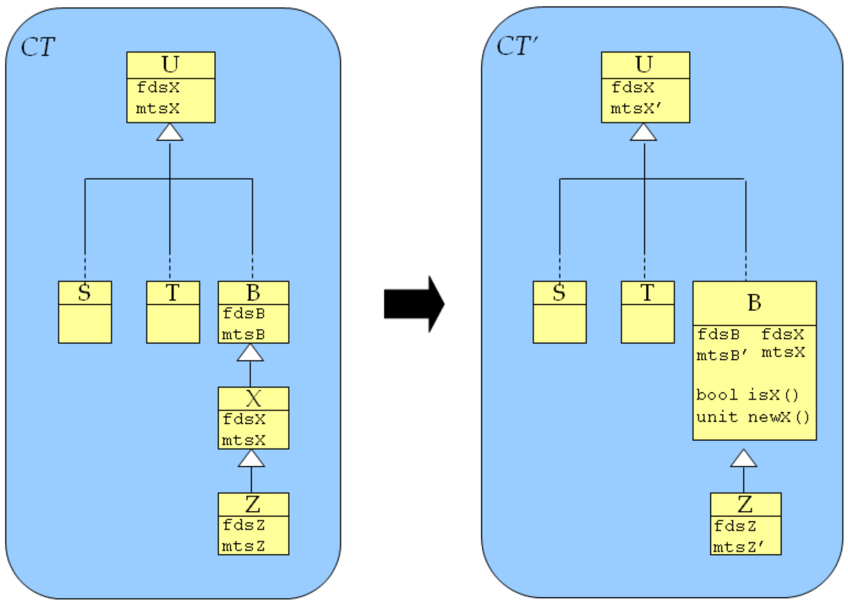


Figure D.6: removeSubclass

Tactic $removeSubclass(X : Class)$

tactic $moveUpFields(X)$;

tactic $adjustHierarchyForPullingUpMethods(X)$;
tactic $moveUpMethods(X)$;

tactic $changeDeclarationsTypetoSuper(X)$;
applies to $cmd[(X)e]$ **do law** $eliminateCastExpressions(cmd[(X)e], \rightarrow)$;
tactic $eliminateTypeTests(X,$ "**bool** $isX()\{$ **result** $:=$ **self is** X }");
tactic $eliminateNew(X)$;

law $changeSuperfromEmptyToImmediateSuperclass(\textbf{immedSubs}(X), \textbf{super}(X), \rightarrow$
);
law $classElimination(X, \rightarrow)$;
end

D.7 introduceField

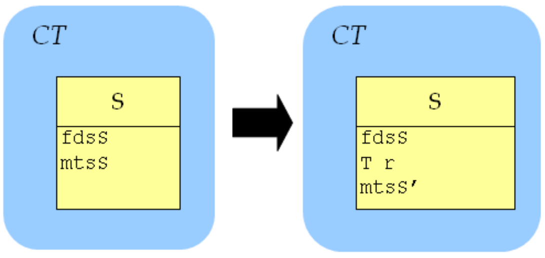


Figure D.7: introduceField

In this section we present the strategies for the considered three cases of relation definitions: empty field, clone field and composition of fields.

```
**Tactic** $introduceEmptyField(r : Field, C : Class)$
    (**law** $rename(r, r', C)$ | **skip**);
    **law** $fieldElimination(r, C, \leftarrow)$;
    **law** ***addtoEnd***(**constructor**$(C)$,"**self**$.r :=\ \varnothing$");
**end**

**Tactic** $introduceCloneField(r : Field, x : field, C : Class)$
    (**law** $rename(r, r', C)$ | **skip**);
    **law** $fieldElimination(r, C, \leftarrow)$;
    $\boxed{\texttt{class C}}$
      **law** **replace**("**self**$.x :=\ exp$",“**unpack self**; **self**$.x :=\ exp$;
        **self**$.r :=$ **self**$.x$; **pack self**”);
    $\boxed{\}}$
**end**

**Tactic** $introduceCompositionField(r, x, y : Field, S, U : Class)$
    (**law** $rename(r, r', S)$ | **skip**);
    (**law** $changeVisibilityPrivatePublic(y, U, \rightarrow)$ | **skip**);
    **law** $fieldElimination(r, S, \leftarrow)$;
    **refactoring** $selfEncapsulateField(y, U, \leftarrow)$;
    $\boxed{\texttt{class S}}$
      **applies to self**$.x.y := exp$ **do**
        **law replace**(“**self**$.x.y := exp$”,
          “**unpack self**; **self**$.x.setY(exp)$; **self**$.r := \{$**self**$.x.getY()\}$; **pack self**”);
    $\boxed{\}}$
    $\boxed{\texttt{class S}}$
      **applies to self**$.x := exp$ **do**
        **law replace**(“**self**$.x := exp$”,
          “**unpack self**; **self**$.x := exp$; $if$(**self**$.x =$ **null**) **then self**$.r := \varnothing$
```

```
                else self.$r$ := {self.$x$.$getY$()}; pack $self$”);
        $\boxed{\}}$
end
```

D.8 removeField

Analogously with the previous section, we present the strategies for the considered three cases of relation definitions: empty field, clone field and composition of fields.

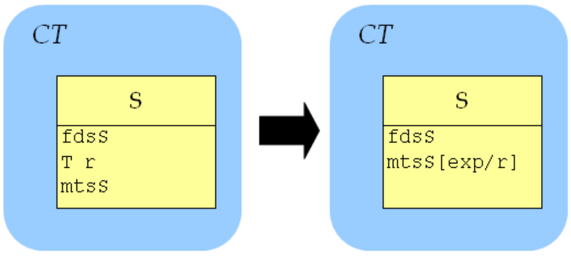


Figure D.8: removeField

```
Tactic $removeEmptyField(r : Field, S : Class)$
        (refactoring $selfEncapsulateField(r, S)$ | skip);
        law replace(”self.$r$ := $exp$”, “skip”);
        law replace(”$exp$[self.$r$]”, “$exp[\varnothing]$”);
        law $fieldElimination(r, S, \rightarrow)$;
end

Tactic $removeCloneField(S : Class, r, x : Field)$
        (refactoring $selfEncapsulateField(r, S)$ | skip);
        law replace(”self.$r$ := $exp$”, “self.$x$ := $exp$; ”);
        law replace(”$exp$[self.$r$]”, “$exp$[self.$x$]”);
        law $fieldElimination(r, S, \rightarrow)$;
end

Tactic $removeCompositionField(r, x, y : Field, S, U : Class)$
        (refactoring $selfEncapsulateField(r, S)$ | skip);
        (law $changeVisibilityPrivatePublic(y, U, \rightarrow)$ | skip);
        replace(“self.$r$ := $exp$”, “self.$x$.$y$ := $exp$; ”);
        replace(“$exp$[self.$r$]”, “$exp$[self.$x$.$y$]”);
        law $fieldElimination(r, S, \rightarrow)$;
end
```

D.9 FromOptionalToSetField

```
Tactic $fromOptionalToSetField(a : Field, C : Class)$
```

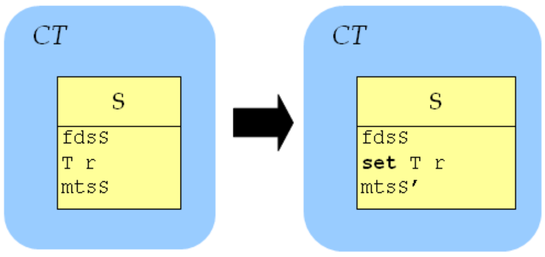


Figure D.9: fromOptionalToSetField

```
        law  $fieldElimination(a', C, \leftarrow)$;
        $\boxed{\text{class C}}$
          law replace("self.$a$ :=  $exp$",
            "if ($exp$ = null) then self.$a'$ := $\varnothing$ else self.$a'$ := {$exp$}")  |
          law replace("$cmd[exp[$self.$a]]$",
            "if (self.$a'$ = $\varnothing$) then $cmd[exp[$null/self.$a]]$
                                    else $cmd[exp[$elem(self.$a'$)/self.$a]]$")  |
          skip);
        $\boxed{\}}$
        law  $rename(a', a, C)$;
end
```

D.10 fromSetToOptionalField

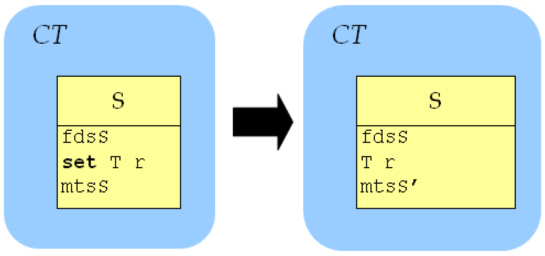


Figure D.10: fromSetToOptionalField

```
Tactic  $fromSetToOptionalField(a : Field, C : Class)$
        law  $fieldElimination(a', C, \leftarrow)$;
        $\boxed{\text{class C}}$
          law replace("self.$a$ :=  $exp$",
            "if ($exp = \varnothing$) then self.$a'$ := null else self.$a'$ := elem($exp$)")  |
          law replace("$cmd[exp[$self.$a]]$",
            "if (self.$a'$ = null) then $cmd[exp[\varnothing/$self.$a]]$
                                    else $cmd[exp[\{$self.$a'\}/$self.$a]]$")  |
        skip);
    $\boxed{\}}$
    law  rename(a', a, C);
end
```

D.11 fromSingleToSetField

```
Tactic fromSingleToSetField(a : Field, C : Class)
    law  fieldElimination(a', C, ←);
    $\boxed{\texttt{class C}}$
        law  replace("self.a := exp", "self.a' := {exp}")  |
        law  replace("cmd[exp[self.a]]",
            "cmd[exp[elem(self.a')/self.a]])"  |
        skip);
    $\boxed{\}}$
    law  rename(a', a, C);
```

D.12 fromSetToSingleField

```
Tactic fromSetToSingleField(a : Field, C : Class)
    law  fieldElimination(a', C, ←);
    $\boxed{\texttt{class C}}$
        law  replace("self.a := exp",
            "self.a' := elem(exp)")  |
        law  replace("cmd[exp[self.a]]",
            "cmd[exp[{self.a'}/self.a]])"  |
        skip);
    $\boxed{\}}$
    law  rename(a', a, C);
end
```

D.13 splitField

```
Tactic splitField(a, x, y : Field, S, U : Class)
    (law  rename(x, x', S)  |  skip);
    (law  rename(U, U')  |  skip);
    law  classElimination(addField(U, y), ←);
    law  changeVisibilityPrivatePublic(S, a, ←);
    law  changeVisibilityPrivatePublic(U, y, →);
    law  fieldElimination(S, x, ←);
    law  addtoEnd(constructor(S), "self.x := new U");
    $\boxed{\texttt{class S}}$
        law  replace("self.a := exp", "unpack self; self.a := exp;
```

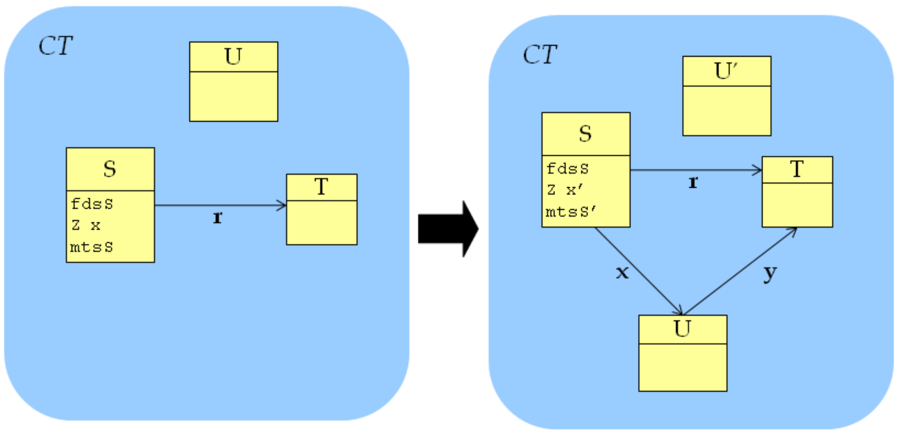


Figure D.11: splitField

```
        elem(self.$x$).$y$ := self.$a$; pack self");
    $\boxed{\}}$
end
```

D.14 removeIndirectReference

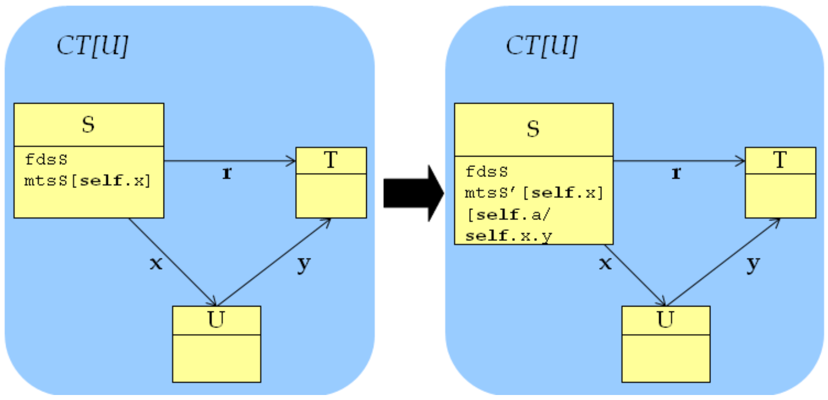


Figure D.12: removeIndirectReference

```
Tactic $removeIndirectReference(a, x, y : Field, S, U : Class)$
    $\boxed{\texttt{class S}}$
        law replace("$cmd[exp[$self$.x.y]]$", "$cmd[exp[$self$.a]]$")  |
        skip);
    $\boxed{\}}$
    (law $fieldElimination(x, S, \rightarrow)$  |  skip);
    (law $classElimination(U, ``", \rightarrow)$  |  skip);
end
```

D.15 Auxiliary Rules

```
Tactic $moveUpFields(X:\ Class)$
    (law $moveFieldToSuperClass(\textbf{fields}(X), X, \textbf{super}(X), \rightarrow)$ |
      refactoring $pullUpPushDownField(\textbf{fields}(X), \textbf{super}(X), \rightarrow)$;
end

Tactic $adjustHierarchyForPullingUpMethods(C:Class)$
    law $changeVisibilityPrivatePublic(\textbf{getHierarchyTopDown}(C),$
        $\textbf{priFields}(\textbf{getHierarchyTopDown}(C)), \rightarrow)$;
    law $eliminateSuper(\textbf{getHierarchyTopDown}(C), \rightarrow)$;
    applies to $exp.x, exp.m(\bar{e})\{\textbf{isExactly}(exp, C)\}$ do
        law $introduceCastToExpression(C, \rightarrow)$;
end

Tactic $moveUpMethods(X:\ Class)$
    law $moveRedefinedMethodToSuperclass(\textbf{redefinedMeths}(X), \textbf{super}(X), \rightarrow)$;
    (law $MethodElimination(\textbf{immedSubs}(\textbf{super}(X)),$
      $\textbf{nonRedefinedMeths}(X), \leftarrow)$ | skip);
    law $moveOriginalMethodToSuperclass(\textbf{meths}(X), \rightarrow)$;
end

Tactic $changeDeclarationsTypetoSuper(C:Class)$
    applies to $C\ \ x$ do law $changeFieldType(x, \textbf{super}(C), \rightarrow)$;
    applies to $C\ \ x := e$ in do law $changeVariableType(x, \textbf{super}(C), \rightarrow)$;
    applies to $T\ \ meth(C\ \ x, pars)$ do law $changeParameterType(x,$
      $\textbf{super}(C), \rightarrow)$;
    applies to $C\ \ meth(pars)$ do law $changeReturnType(x, \textbf{super}(C), \rightarrow)$;
end

Tactic $eliminateTypeTests(X:Class, isMethod:Method)$
    law $methodElimination(isMethod, \textbf{super}(X), \leftarrow)$;
    applies to $cmd[x$ is $X]$ do
      law $varBlockValue(cmd[x$ is $X], test, \textbf{true}, \rightarrow)$;
      law $varBlockResult(\textbf{bool}\ \ test := \ x$ is $X, \textbf{result}, \rightarrow)$;
      law $pcomEliminationResult(\textbf{result} := \ x$ is $X, \leftarrow)$;
      law $ifIdenticalGuardedCommands(test := (\textbf{result} := \ x$ is $X)$,
          "$x = \textbf{null}$", "$test :=$ false", "$test := \ (\textbf{result} := \ x$ is $X)$", $\leftarrow)$;
      law $methodCallElimination(test := \ (\textbf{result} := \ x$ is $X)$,
          "$test := \ x.isX()$", $\leftarrow)$;
      law $varBlockValue(\textbf{bool}\ \ b, \leftarrow)$;
    law $fieldElimination($"string $type$"$, \textbf{super}(X), \leftarrow)$;
    applies to $cmd[x$ is $X]$ do
     (law $methodElimination(\textbf{subclasses}(X), \textbf{constr}, \leftarrow)$ | skip);
     law addtoEnd($\textbf{constructor}(cc)$, "self.$type$ = " + $\textbf{name}(cc)$);
    $disjunction :=$ createDisjunction();
    law addToDisj($subclasses(X)$, $disjunction$, "self is " + $\textbf{name}(cc)$);
    **applies to** $isMethod$ **do**
      **law replace**(''self is $X$", ''self.$type$ ="+ **name**($X$) +'' ∨ "
         + $disjunction$);
end

Tactic $eliminateNew(X : Class)$
  applies to "$x :=$ $new$ " + name($X$) do
    law replace("$x :=$ $new$ " + name($X$), "$x :=$ $new'$"
      + name($X$) + "; $x.newX()$");
  law $moveOriginalMethodToSuperclass$($X$, method("$newX$"), $\rightarrow$);
  law $newSuperclass$($X$, super($X$), $rightarrow$);
end
```

Appendix E

Strategy Proofs

In this appendix, the other developed proofs for the main lemmas of Theorem 7.1 are described. These proofs follow the guidelines presented in Chapter 7.

E.1 introduceClass

In this strategy, a specific class is introduced, based on a correspondent signature added to the model. Let OM, OM' be any two object models and P, P' two programs as follows:

$$\begin{array}{ccc} OM & & OM' \\ \boxed{\begin{array}{l} ps \\ \end{array}} & \Longrightarrow & \boxed{\begin{array}{l} ps \\ \textbf{sig}\ S\ \{\} \end{array}} \end{array}$$

$$\begin{array}{ccc} P & & P' \\ \boxed{\begin{array}{l} CT \\ \textbf{class}\ S\{ \\ \quad ads; \quad mts \\ \} \\ \end{array}} & \Longrightarrow & \boxed{\begin{array}{l} CT' \\ \textbf{class}\ S'\{ \\ \quad ads; \quad mts \\ \} \\ \textbf{class}\ S\{\ \} \end{array}} \end{array}$$

where:
$\quad CT' = CT[S'/S]$

Syntactic conformance

Proof. The goal is to prove validity of the predicate, based on the premise predicates, mainly $syntConformance(OM, P)$.

$syntConformance(OM', P')$
$=$ [definition]
$\quad \forall\, s : sigs(OM') \bullet sigMapping(s, P') \wedge$
$\quad \forall\, r : rels(OM') \bullet relationMapping(r, P') \wedge$

$abstractConstraint(OM', P')$
= [from definition of OM, OM', $sigs(OM') = sigs(OM) \cup \{sigS\}$ and $rels(OM') = rels(OM)$]
$\forall s : sigs(OM) \cup \{sigS\} \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM) \bullet relMapping(r, P') \wedge$
$abstractConstraint(OM', P')$
= [From the definition of OM', P, P',$\forall r : rels(OM') \bullet relMapping(r, P) \Leftrightarrow relMapping(r, P')$]
$\forall s : sigs(OM) \cup \{sigS\} \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM) \bullet relMapping(r, P) \wedge$
$abstractConstraint(OM', P')$
= [From the definition of OM, OM', P, P', $abstractConstraint(OM', P') = abstractConstraint(OM, P)$, which is valid from the premise]
$\forall s : sigs(OM) \cup \{sigS\} \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM) \bullet relMapping(r, P)$
= [From the definition of $syntConformance$, and premise,$\forall r : rels(OM) \bullet relMapping(r, P)$ is valid]
$\forall s : sigs(OM) \cup \{sigS\} \bullet sigMapping(s, P')$
= [set theory]
$\forall s : sigs(OM) \bullet sigMapping(s, P') \wedge sigMapping(s, P')$
= [from $premise(OM, OM', P)$, $\forall s : sigs(S) \bullet s \neq S \Rightarrow sigMapping(s, P') \Leftrightarrow sigMapping(s, P)$, which is valid]
$sigMapping(s, P')$
= [from the definition of P',$sigMapping(s, P')$ is valid]
$true$

□

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement. In this case, `S` $\in Own$.

1. No method interface is changed, thus from $premise(OM, OM', P)$ there are no methods in *Own* with *Rep* return types;
2. No inherited methods are added, thus from $premise(OM, OM', P)$ no inherited methods have *Rep* parameters;
3. Same as above, thus from $premise(OM, OM', P)$, *Rep* classes do not inherit methods from non-*Rep* classes;
4. No public fields are affected, thus from $premise(OM, OM', P)$ no `e.f` is seen, `e` $\leq Own$, unless `e` is **self**;
5. No **new** is changed, thus from $premise(OM, OM', P)$ *Rep* instance is created outside *Own* classes;
6. No method call is affected, thus from $premise(OM, OM', P)$, `e.m` calls within *Own* or *Rep* do not have *Rep* parameters.

□

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. If `S` is already declared, we rename it to `S'`, which is a straightforward equivalence;
2. Law 3 *Class Elimination* (R-L), whose provisos are valid:
   (a) The name of class `U` is distinct from those of all classes declared in CT;
   (b) The superclass appearing in `U` is either `Object` or declared in CT; `U` has no superclass;
   (c) The field and method names declared by `U` are not declared by its superclasses in CT, except in the case of method redefinitions; `U` has no superclass.

□

Semantic conformance

First we prove a few auxiliary lemmas as they are used in the proof.

Lemma E.1. The semantics of OM' can be defined in terms of $semantics(OM)$ as follows.
$semantics(OM') = \{i \oplus (\textbf{sig}\ S).name \mapsto \varnothing \mid i \in semantics(OM)\}$

Proof. We start from the definition of $semantics(OM')$.
$semantics(OM')$
$=$ [definition]
$\{i : Interpretation \mid satisfyImpInvs(OM', i) \wedge\ satisfyExpInvs(OM', i)\}$
$=$ [when introducing r, no **extends** clause is affected, thus $satisfyImpInvs(OM, i) = satisfyImpInvs(OM', i)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge\ satisfyExpInvs(OM', i)\}$
$=$ [from definition of $satisfyExpInvs(OM', i)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge \forall f : factInvs(OM') \bullet$
$satisfyFormula(f, i)\}$
$=$ [from definitions of OM, OM',$factInvs(OM') = factInvs(OM)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge$
$\forall f : factInvs(OM) \bullet satisfyFormula(f, i)\}$
$=$ [from definition of $semantics(OM)$]
$\{i : Interpretation \mid i \in semantics(OM)\}$
$=$ [From Lemma 7.5, where $v = \varnothing$]
$\{i \oplus (\textbf{sig}\ S).name \mapsto \varnothing \mid i \in semantics(OM)\}$

□

Lemma E.2. For P and P':
$heaps(P', filter) = \{h \oplus (\textbf{class}\ S).name \mapsto \varnothing \mid h \in heaps(P, filter)\}$
Proof. The changed commands do not add or remove heaps; field `r` is private, so it was only accessed in *mts*.

□

Main proof. Lemma 7.4 is then proved, using the result of the auxiliary lemmas and premise.

$semanticConformance(OM', P')$
$=$ [definition]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : semantics(OM') \bullet$
$\forall s : sigs(OM') \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\forall r : rels(OM') \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$=$ [from definitions of OM,OM',$rels(OM') = rels(OM)$,$sigs(OM') = sigs(OM) \cup \{sig\ S\}$]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : semantics(OM') \bullet$
$\forall s : sigs(OM) \cup \{sig\ S\} \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$=$ [set theory]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : semantics(OM') \bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$i.mapSig(S) = h.mapClass(S) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$=$ [Lemma E.1 replaces $semantics(OM')$]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : \{i \oplus (\textbf{sig}\ S).name \mapsto \varnothing \mid i \in semantics(OM)\} \bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$i.mapSig(S) = h.mapClass(S) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$=$ [Lemma E.2 replaces $heaps(P', filter)$]
$\forall h : \{h \oplus (\textbf{class}\ S).name \mapsto \varnothing \mid h \in heaps(P, filter)\} \bullet$
$\exists\ i : i : \{i \oplus (\textbf{sig}\ S).name \mapsto \varnothing \mid i \in semantics(OM)\} \bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$i.mapSig(S) = h.mapClass(S) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$=$ [From predicate calculus, choosing an arbitrary $h1$]
$\exists\ i : \{i \oplus (\textbf{sig}\ S).name \mapsto \varnothing \mid i \in semantics(OM)\} \bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$i.mapSig(S) = h1.mapClass(S) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = mapField(h1)(r.name)$
$=$ [For the existential quantification we choose the specific interpretation $i1$, which repeats the mappings for signatures and relations of $h1$, and $i1.mapSig((\textbf{sig}\ S).name) = \varnothing$]

$\quad \forall\, s : sigs(OM) \bullet \text{i1}.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\quad i1.mapSig(S) = h1.mapClass(S) \wedge$
$\quad \forall\, r : rels(OM) \bullet \text{i1}.mapRel(r.name) = mapField(h1)(r.name)$
$=$ [from $premise(OM,\ OM',\ P)$]
$\quad i1.mapSig(S) = h1.mapClass(S)$
$=$ [from definition of $i1$ and $h1$, $i1.mapSig(S) = h1.mapClass(S) = \varnothing$]
$\quad true$

□

E.2 introduceSuperclass

In this strategy, a superclass is introduced for two other classes. Also, an invariant stating that the newly-introduced class is abstract must be satisfied.

Let OM, OM' be any two object models and P, P' two programs as follows:

OM

```
ps
sig S { rsS }
sig T { rsT }
fact F { forms }
```

$\Longrightarrow$

OM'

```
ps
sig U {}
sig S extends U{ rsS }
sig T extends U{ rsT }
fact F {
  forms
  U = S + T
}
```

P

```
CT
class U{...}
class SC{..}
class S1 extends SC{..}
class T1 extends SC{..}
class S extends X{..}
class T extends Y{..}
```

$\Longrightarrow$

P'

```
CT'
class U'{...}
class SC{..}
class U extends SC{ }
class S1 extends U{..}
class T1 extends U{..}
class S extends X{..}
class T extends Y{..}
```

where:
$\quad CT' = CT[U'/U]$
$\quad X \leq S1$ and $Y \leq T1$.

Syntactic conformance

Proof. The goal is to prove validity of the predicate, based on the premise predicates, mainly $syntConformance(OM, P)$.

$syntConformance(OM', P')$
$=$ [definition]
$\quad \forall s : sigs(OM') \bullet sigMapping(s, P') \wedge$
$\quad \forall r : rels(OM') \bullet relationMapping(r, P') \wedge$
$\quad abstractConstraint(OM', P')$
$=$ [from definitions of OM, OM',$sigs(OM') = sigs(OM) \cup \{\textbf{sig}\ U\}$ and $rels(OM') = rels(OM)$]
$\quad \forall s : sigs(OM) \cup \{\textbf{sig}\ U\} \bullet sigMapping(s, P') \wedge$
$\quad \forall r : rels(OM) \bullet relMapping(r, P') \wedge$
$\quad abstractConstraint(OM', P')$
$=$ [From definitions of $OM', P, P', \forall r : rels(OM') \bullet relMapping(r, P) = relMapping(r, P')$]
$\quad \forall s : sigs(OM) \cup \{\textbf{sig}\ U\} \bullet sigMapping(s, P') \wedge$
$\quad \forall r : rels(OM) \bullet relMapping(r, P) \wedge$
$\quad abstractConstraint(OM', P')$
$=$ [From definitions of OM', P', $abstractConstraint(OM', P') = abstractConstraint\ (OM, P)$, as every intermediate program class is still abstract (no command is added)]
$\quad \forall s : sigs(OM) \cup \{\textbf{sig}\ U\} \bullet sigMapping(s, P') \wedge$
$\quad \forall r : rels(OM) \bullet relMapping(r, P)$
$=$ [From definition of *syntConformance*, and premise,$\forall r : rels(OM) \bullet relMapping(r, P)$ is valid]
$\quad \forall s : sigs(OM) \cup \{\textbf{sig}\ U\} \bullet sigMapping(s, P')$
$=$ [set theory]
$\quad \forall s : sigs(OM) \bullet sigMapping(s, P') \ \wedge\ sigMapping((\textbf{sig}\ U), P')$
$=$ [from $premise(OM, OM', P)$,$\forall s : sigs(S) \bullet s \neq U \Rightarrow sigMapping(s, P') \Leftrightarrow sigMapping(s, P)$, which is valid]
$\quad sigMapping(\textbf{sig}\ U, P')$
$=$ [definition of *sigMapping*]
$\quad \exists cl : classes(P) \bullet (\textbf{sig}\ U).name = cl.name \wedge$
$\quad\quad list2set((\textbf{sig}\ U).extends) \subseteq list2set(cl.extends)$
$=$ [predicate calculus, choosing class U]
$\quad (\textbf{sig}\ U).name = (\textbf{class}\ U\ \textbf{extends}\ SC).name \wedge$
$\quad list2set((\textbf{sig}\ U).extends) \subseteq list2set((\textbf{class}\ U\ \textbf{extends}\ SC).extends)$
$=$ [$(\textbf{sig}\ U).name = (\textbf{class}\ U\ \textbf{extends}\ SC).name$ and $list2set((\textbf{sig}\ U).extends) = \{object\}$]
$\quad true$ □

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement. In this case, `U` $\in$ *Own*.

1. No method interface is changed, thus from $premise(OM,\ OM',\ P)$ there are no methods in *Own* with *Rep* return types;

2. No inherited methods are added, thus from $premise(OM,\ OM',\ P)$ no inherited methods have *Rep* parameters;

3. Same as above, thus from $premise(OM, OM', P)$, *Rep* classes do not inherit methods from non-*Rep* classes;

4. No public fields are affected, thus from $premise(OM, OM', P)$ no `e.f` is seen, `e` $\leq$ *Own*, unless `e` is **self**;

5. No **new** is changed, thus from $premise(OM, OM', P)$ *Rep* instance is created outside *Own* classes;

6. No method call is affected, thus from $premise(OM, OM', P)$, `e.m` calls within *Own* or *Rep* do not have *Rep* parameters.

□

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. If `U` is already declared, we rename it to `U'`, which is a straightforward equivalence;

2. Law 3 *Class Elimination* (R-L) is applied, whose provisos are valid:

   (a) The name of class `U` is distinct from those of all classes declared in CT;

   (b) The superclass appearing in `U` is either `Object` or declared in CT; `U`'s superclass is defined as the first common superclass;

   (c) The field and method names declared by `U` are not declared by its superclasses in CT, except in the case of method redefinitions; `U` is empty.

3. For each immediate subclass of `cc`, the first common superclass for the given class list, Law 26 *Change superclass: from empty to immediate* (R-L), with no provisos.

□

Semantic conformance

First we prove a few auxiliary lemmas as they are used in the proof.

Lemma E.3. The semantics of OM' can be defined in terms of $semantics(OM)$ as follows.

$semantics(OM') = \{i \oplus (\textbf{sig}\ U).name \mapsto i.mapSig(\textbf{sig}\ S) \cup i.mapSig(\textbf{sig}\ T) \mid i \in semantics(OM)\}$

Proof. We start from the definition of $semantics(OM')$.
$semantics(OM')$
$=$ [definition]
$\{i : Interpretation \mid satisfyImpInvs(OM', i) \wedge satisfyExpInvs(OM', i)\}$

$= $ [from definition of $\mathit{satisfyImpInvs}(OM', i)$]
$\{i : \mathit{Interpretation} \mid$
$(\forall\, s : \mathit{sigs}(OM') \bullet i.\mathit{mapSig}(s.\mathit{name}) \subseteq i.\mathit{mapSig}((\mathit{super}(s)).\mathit{name})) \wedge$
$\mathit{satisfyExpInvs}(OM', i)\}$
$= $ [from definitions of OM, OM',$\mathit{sigs}(OM') = \mathit{sigs}(OM) \cup (\textbf{sig}\ U)$]
$\{i : \mathit{Interpretation} \mid$
$(\forall\, s : \mathit{sigs}(OM) \cup (\textbf{sig}\ U) \bullet i.\mathit{mapSig}(s.\mathit{name}) \subseteq i.\mathit{mapSig}((\mathit{super}(s)).\mathit{name})) \wedge$
$\mathit{satisfyExpInvs}(OM', i)\}$
$= $ [set theory]
$\{i : \mathit{Interpretation} \mid$
$(\forall\, s : \mathit{sigs}(OM) \bullet i.\mathit{mapSig}(s.\mathit{name}) \subseteq i.\mathit{mapSig}((\mathit{super}(s)).\mathit{name}) \wedge$
$i.\mathit{mapSig}((\textbf{sig}\ U).\mathit{name}) \subseteq i.\mathit{mapSig}((\mathit{super}(\textbf{sig}\ U)).\mathit{name}))$
$\wedge\ \mathit{satisfyExpInvs}(OM', i)\}$
$= $ [from definition of $\mathit{satisfyImpInvs}(OM)$]
$\{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i) \wedge\ \mathit{satisfyExpInvs}(OM', i)\}$
$= $ [from definition of $\mathit{satisfyExpInvs}(OM', i)$]
$\{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i) \wedge \forall\, f : \mathit{factInvs}(OM') \bullet$
$\mathit{satisfyFormula}(f, i)\}$
$= $ [from definitions of OM, OM',$\mathit{factInvs}(OM') = \mathit{factInvs}(OM) \cup (U = S + T)$]
$\{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i) \wedge$
$\forall\, f : \mathit{factInvs}(OM) \cup \{U = S + T\} \bullet \mathit{satisfyFormula}(f, i)\}$
$= $ [set theory]
$\{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i) \wedge$
$\forall\, f : \mathit{factInvs}(OM) \bullet \mathit{satisfyFormula}(f, i) \wedge$
$\mathit{satisfyFormula}(\{U - S + T\}, i)\}$
$= $ [from definition of $\mathit{semantics}(OM)$]
$\{i : \mathit{Interpretation} \mid i \in \mathit{semantics}(OM)\ \wedge\ \mathit{satisfyFormula}(\{U = S + T\}, i)\}$
$= $ [From Lemma 7.5, where $v = i.\mathit{mapSig}(S)\ \cup\ i.\mathit{mapSig}(T)$]
$\{i \oplus (\textbf{sig}\ U).\mathit{name} \mapsto i.\mathit{mapSig}(\textbf{sig}\ S) \cup i.\mathit{mapSig}(\textbf{sig}\ T)\ \mid\ i \in \mathit{semantics}(OM)\}$

Lemma E.4. For OM' and P':
$\forall\, h : \mathit{heaps}(P', \mathit{filter}) \bullet$
$h.\mathit{mapClass}(U) = h.\mathit{mapClass}(S)\ \cup\ h.\mathit{mapClass}(T)$

Proof. From definition of P'
$\forall\, h : \mathit{heaps}(P', \mathit{filter}) \bullet$
$h.\mathit{mapClass}(U) = \bigcup_{U \in \mathit{hierarchy}(U,S) \cup \mathit{hierarchy}(U,T)} h.\mathit{mapClass}(U)$
$= $ [From $\mathit{abstractConstraint}(OM', P')$, the only concrete subclasses of U are S and T]
$\forall\, h : \mathit{heaps}(P', \mathit{filter}) \bullet$
$h.\mathit{mapClass}(U) = h.\mathit{mapClass}(S)\ \cup\ h.\mathit{mapClass}(T)$

Main proof. Lemma 7.4 is then proved, using the result of the auxiliary lemmas and premise.

$semanticConformance(OM', P')$
= [definition]
$\quad \forall h : heaps(P', filter) \bullet$
$\quad\quad \exists\ i : semantics(OM') \bullet$
$\quad\quad\quad \forall s : sigs(OM') \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad \forall r : rels(OM') \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
= [from definitions of OM, OM', $rels(OM') = rels(OM)$, $sigs(OM') = sigs(OM) \cup \{sig\ U\}$]
$\quad \forall h : heaps(P', filter) \bullet$
$\quad\quad \exists\ i : semantics(OM') \bullet$
$\quad\quad\quad \forall s : sigs(OM) \cup \{sig\ \ U\} \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad \forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
= [set theory]
$\quad \forall h : heaps(P', filter) \bullet$
$\quad\quad \exists\ i : semantics(OM') \bullet$
$\quad\quad\quad \forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad i.mapSig(U) = h.mapClass(U) \wedge$
$\quad\quad\quad \forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
= [Lemma E.3 replaces $semantics(OM')$]
$\quad \forall h : heaps(P', filter) \bullet$
$\quad\quad \exists\ i : \{i \oplus (\textbf{sig}\ U).name \mapsto i.mapSig(S) \cup i.mapSig(T) \mid i \in semantics(OM)\} \bullet$
$\quad\quad\quad \forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad i.mapSig(U) = h.mapClass(U) \wedge$
$\quad\quad\quad \forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
= [Lemma E.4]
$\quad \forall h : \{h \oplus (\textbf{sig}\ U).name \mapsto h.mapClass(S) \cup h.mapClass(T) \mid h \in heaps(P, filter)\} \bullet$
$\quad\quad \exists\ i : \{i \oplus (\textbf{sig}\ U).name \mapsto i.mapSig(S) \cup i.mapSig(T) \mid i \in semantics(OM)\} \bullet$
$\quad\quad\quad \forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad i.mapSig(U) = h.mapClass(U) \wedge$
$\quad\quad\quad \forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
= [From predicate calculus, choosing an arbitrary $h1$]
$\quad \exists\ i : \{i \oplus (\textbf{sig}\ U).name \mapsto i.mapSig(S) \cup i.mapSig(T) \mid i \in semantics(OM)\} \bullet$
$\quad\quad \forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\quad\quad i.mapSig(U) = h1.mapClass(U) \wedge$
$\quad\quad \forall r : rels(OM) \bullet\ i.mapRel(r.name) = mapField(h1)(r.name)$
= [From predicate calculus, we choose the specific interpretation $i1$, which repeats the mappings for signatures and relations of $h1$ and the mapping for U is according to the definition]
$\quad \forall s : sigs(OM) \bullet \text{i1}.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\quad i1.mapSig(U) = h1.mapClass(U) \wedge$
$\quad \forall r : rels(OM) \bullet \text{i1}.mapRel(r.name) = mapField(h1)(r.name)$
= [from $premise(OM,\ OM',\ P)$]
$\quad i1.mapSig(U) = h1.mapClass(U)$
= [from definition of $i1$ and $h1$, $i1.mapSig(U) = h1.mapClass(U) = h1.mapClass(S) \cup h1.mapClass(T)$]
$\quad true$

$\square$

E.3 removeSuperclass

In contrast to the previously proved strategy, a common superclass is removed from the program, along with the invariant.

Let OM, OM' be any two object models and P, P' two programs as follows:

OM

```
ps
sig U {}
sig S extends U{ rs }
sig T extends U{ rs' }
fact F {
  forms
  U = S + T
}
```

$\Longrightarrow$

OM'

```
ps
sig S { rs }
sig T { rs' }
fact F { forms }
```

P

```
CT
class SC{
  adsSC;  mtsSC}
class U extends SC{
  adsU;  mtsU}
class S1 extends U{
  .. mtsS1}
class T1 extends U{
  .. mtsT1}
class S extends X{
  .. mtsS}
class T extends Y{
  .. mtsT}
```

$\Longrightarrow$

P'

```
CT'
class SC{
  adsSC, adsU;  mtsSC'}
class S1 extends U{
  .. mtsS1', mtsU'}
class T1 extends U{
  .. mtsT1', mtsU'}
class S extends X{
  .. mtsS'}
class T extends Y{
  .. mtsT'}
```

where:

$X \leq S1$ and $Y \leq T1$;

$CT'', mts'' = CT, mts[exp \textbf{ is } U/(exp \textbf{ is } S \vee exp \textbf{ is } T)]$;

$CT', mts' = CT'', mts''[cmd[(U)exp]/\textbf{if}(exp \textbf{ is } S)cmd[(S)exp] \textbf{ else } cmd[(T)exp]]$.

Syntactic conformance

Proof. The goal is to prove validity of the predicate, based on the premise predicates, mainly $syntConformance(OM, P)$.

$syntConformance(OM', P')$

$=$ [definition]

$\forall s : sigs(OM') \bullet sigMapping(s, P') \wedge$

$\forall r : rels(OM') \bullet relationMapping(r, P') \wedge$

$abstractConstraint(OM', P')$

= [from definition of OM, OM', $sigs(OM') = sigs(OM) - \{\textbf{sig}\ U\}$ and $rels(OM') = rels(OM)$]
$\quad \forall\, s : sigs(OM) - \{\textbf{sig}\ U\} \bullet sigMapping(s, P') \wedge$
$\quad \forall\, r : rels(OM) \bullet relMapping(r, P') \wedge$
$\quad abstractConstraint(OM', P')$
= [From definitions of OM', P, P',$\forall\, r : rels(OM') \bullet\ relMapping(r, P) \Leftrightarrow relMapping(r, P')$]
$\quad \forall\, s : sigs(OM) - \{\textbf{sig}\ U\} \bullet sigMapping(s, P') \wedge$
$\quad \forall\, r : rels(OM) \bullet relMapping(r, P) \wedge$
$\quad abstractConstraint(OM', P')$
= [From definitions of OM', P' and premise, $abstractConstraint(OM', P')$ is valid, as no **new** command is added for classes not in the model]
$\quad \forall\, s : sigs(OM) - \{\textbf{sig}\ U\} \bullet sigMapping(s, P') \wedge$
$\quad \forall\, r : rels(OM) \bullet relMapping(r, P)$
= [from definition of *syntConformance*, and premise,$\forall\, r : relations(OM) \bullet relMapping(r, P)$ is valid]
$\quad \forall\, s : sigs(OM) - \{\textbf{sig}\ U\} \bullet sigMapping(s, P') \wedge$
= [set theory]
$\quad \forall\, s : sigs(OM) \bullet (s \neq \textbf{sig}\ U) \Rightarrow sigMapping(s, P')$
= [predicate calculus, choosing arbitrary $s1$]
$\quad (\textbf{sig}\ U \neq s1) \Rightarrow sigMapping(s1, P')$
= [Assuming $(\textbf{sig}\ U \neq s1)$ as a premise]
$\quad sigMapping(s1, P')$
= [As $(\textbf{sig}\ U \neq s1)$, and from definitions P, P' no other class is removed but U, from $premise(OM,\ OM',\ P)$ $sigMapping(s1, P')$ is valid]
$\quad true$

□

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement.

1. All methods that were pushed down to subclasses of `U` were inherited by those subclasses before, thus from $premise(OM,\ OM',\ P)$ there are no methods in *Own* with *Rep* return types;

2. No inherited methods are added, thus from $premise(OM,\ OM',\ P)$ no inherited methods have *Rep* parameters;

3. If methods are inherited and previous subclasses of `U` are *Rep*, thus from $premise(OM,\ OM',\ P)$, *Rep* classes do not inherit methods from non-*Rep* classes;

4. No public fields of *Own* classes are used outside their declaring module, thus from $premise(OM,\ OM',\ P)$ no `e.f` is seen, unless `e` is **self**;

5. No **new** is changed, thus from $premise(OM,\ OM',\ P)$ *Rep* instance is created outside *Own* classes;

6. No method call is affected, thus from *premise*(OM, OM', P), `e.m` calls within *Own* or *Rep* do not have *Rep* parameters.

$\square$

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. For each field in superclasses of U, Law 27 *change visibility: from pri to pub* (L-R), with no precondition;
2. For each superclass, in a top-down list, replace every method call using **super** with Law 38 *eliminate super* (L-R), whose provisos are valid:
   (a) **super** and private fields does not appear in the inline code, as the iteration is top-down.
3. Within each immediate subclass, replace every method call using **super** with Law 38 *eliminate super* (L-R), whose provisos are valid:
   (a) **super** and private fields does not appear in the inline code, as all superclasses have been changes in the previous step.
4. Each expression whose type is `U` is cast with this type, using Law 40 *introduce trivial cast in expressions* (L-R), with no provisos.
5. Declarations are changed; for instance, Law 29 *change field type* (L-R), whose precondition is valid:
   (a) $U \leq \textbf{super}(U)$, and all U expressions are cast, from the previous step.
6. Refactoring 2 *distribute cast* is applied, precondition is fulfilled (class U has no instances).
7. Law 5 *distribute test* is applied, precondition is fulfilled (class U has no instances).
8. Law 6 *redundant if* (R-L) is applied to all `m` command bodies.
9. Law 35 *move redefined method to superclass* (L-R) is applied to all methods, whose body was transformed into an if statement; the provisos are valid:
   (a) **super** and private fields do not appear into U's methods, from previous steps;
   (b) **super**.m() does not appear in the subclasses that will receive the method declaration;
   (c) If the subclass already declares `m` then the law fails, then nothing happens (strategy skip).

10. Refactoring 3 *method elimination: abstract class* is applied, with the following provisos:
    (a) `U` is an abstract class;
    (b) Method calls to each method are made from other types than `U`, since `U` was removed from all declarations.
11. For each field, Law 37 *move field to superclass* (L-R) is applied, with the following provisos:
    (a) If the field is declared in subclasses of **super**`(U)`, then the application fails (strategy skip).
12. For each immediate subclass of `cc`, the first common superclass for the given class list, Law 26 *change superclass: from an empty class to immediate superclass* (R-L), with no provisos.
13. Law 3 *class elimination* (R-L) is applied, whose provisos are valid:
    (a) The name of class `U` is distinct from those of all classes declared in CT;
    (b) The superclass appearing in `U` is either `Object` or declared in CT; `U`'s superclass is defined as the first common superclass;
    (c) The field and method names declared by `U` are not declared by its superclasses in CT, except in the case of method redefinitions; `U` is empty.

□

Semantic conformance

First we prove a few auxiliary lemmas as they are used in the proof.

Lemma E.5. The semantics of OM' can be defined in terms of $semantics(OM)$ as follows.

$$\begin{aligned}&semantics(OM') = i : \{i : interpretation \\ &\quad \mid i \in semantics(OM) \wedge i \rhd (\mathbf{sig}\ U).name\}\end{aligned}$$

Proof. We start from the definition of $semantics(OM')$.
$semantics(OM')$
= [definition]
$\{i : Interpretation \mid satisfyImpInvs(OM', i) \wedge\ satisfyExpInvs(OM', i)\}$
= [from definition of $satisfyImpInvs(OM', i)$]
$\{i : Interpretation \mid$
$(\forall\, s : sigs(OM') \bullet i.mapSig(s.name) \subseteq i.mapSig((super(s)).name)) \wedge$
$satisfyExpInvs(OM', i)\}$
= [from definitions of OM, OM',$sigs(OM') = sigs(OM) - (\mathbf{sig}\ U)$]

$\{i : Interpretation \mid$
$(\forall s : sigs(OM) - (\textbf{sig } U) \bullet$
$i.mapSig(s.name) \subseteq i.mapSig((super(s)).name)) \wedge$
$satisfyExpInvs(OM', i)\}$
= [set theory]
$\{i : Interpretation \mid$
$(\forall s : sigs(OM) \bullet$
$s \neq (\textbf{sig } U) \ \wedge \ i.mapSig(s.name) \subseteq i.mapSig((super(s)).name))$
$\wedge \ satisfyExpInvs(OM', i)\}$
= [from definition of $satisfyImpInvs(OM)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge \ satisfyExpInvs(OM', i)\}$
= [from definition of $satisfyExpInvs(OM', i)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge \forall f : factInvs(OM') \bullet$
$satisfyFormula(f, i)\}$
= [from definitions of OM, OM', $factInvs(OM') = factInvs(OM) - (U = S + T)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \ \wedge$
$\forall f : factInvs(OM) - \{r = x.y\} \bullet satisfyFormula(f, i)\}$
= [set theory]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \ \wedge$
$\forall f : factInvs(OM) \bullet (f \neq \{U = S + T\}) \Rightarrow satisfyFormula(f, i)\}$
= [from Lemma 7.8, i is part of the semantics of OM]
$\{i : Interpretation \mid i \in semantics(OM) \wedge i \trianglerighteq (\textbf{sig } U).name\}$

Lemma E.6. For P and P':
$heaps(P', filter) = \{h : Heap \mid h \in heaps(P, filter) \wedge h \trianglerighteq (\textbf{sig } U).name\}$

Proof. The changed commands do not add or remove heaps; field `r` is private, so it was only accessed in *mts*.

Main proof. Lemma 7.4 is then proved, using the result of the auxiliary lemmas and premise.

$semanticConformance(OM', P')$
= [definition]
$\forall h : heaps(P', filter) \bullet$
$\exists \ i : semantics(OM') \bullet$
$\forall s : sigs(OM') \bullet \ i.mapSig(s.name) = h.mapClass(s.name) \ \wedge$
$\forall r : rels(OM') \bullet \ i.mapRel(r.name) = h.mapField(r.name)$
= [from $premise(OM, \ OM', \ P)$, $rels(OM') = rels(OM)$, $sigs(OM') = sigs(OM) - \{\textbf{sig } U\}$]
$\forall h : heaps(P', filter) \bullet$
$\exists \ i : semantics(OM') \bullet$
$\forall s : sigs(OM) - \{\textbf{sig } U\} \bullet \ i.mapSig(s.name) = h.mapClass(s.name) \ \wedge$
$\forall r : rels(OM) \bullet \ i.mapRel(r.name) = h.mapField(r.name)$
= [set theory]
$\forall h : heaps(P', filter) \bullet$

$\quad\quad \exists\ i : semantics(OM')\bullet$
$\quad\quad\quad \forall\, s : sigs(OM)\bullet\ (\mathbf{sig}\ U) \neq s \Rightarrow i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad \forall\, r : rels(OM)\bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [Lemma E.5 replaces $semantics(OM')$]

$\quad \forall\, h : heaps(P', filter)\bullet$
$\quad\quad \exists\ i : \{i : Interpretation \mid i \in semantics(OM) \wedge i \rhd (\mathbf{sig}\ U).name\}\bullet$
$\quad\quad\quad \forall\, s : sigs(OM)\bullet\ (\mathbf{sig}\ U) \neq s \Rightarrow i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad \forall\, r : rels(OM)\bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [Lemma E.6 replaces $heaps(P', filter)$]

$\quad \forall\, h : \{h : Heap \mid h \in heaps(P, filter) \wedge h \rhd (\mathbf{sig}\ U).name\}\bullet$
$\quad\quad \exists\ i : \{i : Interpretation \mid i \in semantics(OM) \wedge \rhd(\mathbf{sig}\ U).name\}\bullet$
$\quad\quad\quad \forall\, s : sigs(OM)\bullet\ (\mathbf{sig}\ U) \neq s \Rightarrow i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad \forall\, r : rels(OM)\bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [From predicate calculus, choosing an arbitrary $h1$]

$\quad \exists\ i : \{i : Interpretation \mid i \in semantics(OM) \wedge i \rhd (\mathbf{sig}\ U).name\}\bullet$
$\quad\quad \forall\, s : sigs(OM)\bullet\ (\mathbf{sig}\ U) \neq s \Rightarrow i.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\quad\quad \forall\, r : rels(OM)\bullet\ i.mapRel(r.name) = mapField(h1)(r.name)$

= [For existential quantification, we choose interpretation $i1$ which repeats the mappings for model names in $h1$, removing the mapping from U signature]

$\quad \forall\, s : sigs(OM)\bullet\ (\mathbf{sig}\ U) \neq s \Rightarrow i1.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\quad \forall\, r : rels(OM)\bullet\ i1.mapRel(r.name) = mapField(h1)(r.name)$

= [from $premise(OM, OM', P)$]

$\quad \forall\, s : sigs(OM)\bullet\ (\mathbf{sig}\ U) \neq s \Rightarrow i1.mapSig(s.name) = h1.mapClass(s.name)$

= [from definition of $i1$ and $h1$, every original model signature and class is mapped to the same values, except U]

$\quad true$

$\square$

E.4 introduceSubclass

In this strategy, a subclass is introduced, making the superclass abstract. Let OM, OM' be any two object models and P, P' two programs as follows:

OM		OM'

ps
sig U { rsU }
sig S **extends** U{ rsS }
sig T **extends** U{ rsT }
fact F { *forms* }
	$\Longrightarrow$	*ps*
		sig U { rsU }
		sig S **extends** U{ rsS }
		sig T **extends** U{ rsT }
		sig X **extends** U{}
		fact F {
		  forms
		  $X = U - S - T$
		}

$$\begin{array}{ccc} P & & P' \\ \boxed{\begin{array}{l} CT \\ \textbf{class } X\{..\} \\ \textbf{class } U\{ \\ \quad ..\ mtsU\} \\ \textbf{class } S \textbf{ extends } Y\{ \\ \quad ..\ mtsS\} \\ \textbf{class } T \textbf{ extends } W\{ \\ \quad ..\ mtsT\} \\ \\ \end{array}} & \Longrightarrow & \boxed{\begin{array}{l} CT' \\ \textbf{class } X'\{..\} \\ \textbf{class } U\{ \\ \quad ..\ mtsU'\} \\ \textbf{class } S \textbf{ extends } Y\{ \\ \quad ..\ mtsS'\} \\ \textbf{class } T \textbf{ extends } W\{ \\ \quad ..\ mtsT'\} \\ \textbf{class } X \textbf{ extends } U\{\ \} \end{array}} \end{array}$$

where:
$Y \leq U$ and $W \leq U$;
$CT'', mts'' = CT, mts[X'/X]$;
$CT', mts' = CT'', mts''[\mathbf{new} X/\mathbf{new} U]$;

Syntactic conformance

Proof. The goal is to prove validity of the predicate, based on the premise predicates, mainly $syntConformance(OM, P)$.

$syntConformance(OM', P')$
$=$ [definition]
$\forall s : sigs(OM') \bullet sigMapping(s, P') \land$
$\forall r : rels(OM') \bullet relMapping(r, P') \land$
$abstractConstraint(OM', P')$
$=$ [from definitions of OM, OM',$sigs(OM') = sigs(OM) \cup \{\mathbf{sig}\ X\ \mathbf{extends}\ U\}$ and $rels(OM') = rels(OM)$]
$\forall s : sigs(OM) \cup \{\mathbf{sig}\ X\ \mathbf{extends}\ U\} \bullet sigMapping(s, P') \land$
$\forall r : rels(OM) \bullet relMapping(r, P') \land$
$abstractConstraint(OM', P')$
$=$ [From definitions of $OM', P, P', \forall r : rels(OM') \bullet relMapping(r, P) = relMapping(r, P')$]
$\forall s : sigs(OM) \cup \{\mathbf{sig}\ X\ \mathbf{extends}\ U\} \bullet sigMapping(s, P') \land$
$\forall r : rels(OM) \bullet relMapping(r, P) \land$
$abstractConstraint(OM', P')$
$=$ [From definitions of OM', P', $abstractConstraint(OM', P') = abstractConstraint(OM, P)$, as every intermediate program class is still abstract, and X and U are in the model]
$\forall s : sigs(OM) \cup \{\mathbf{sig}\ X\ \mathbf{extends}\ U\} \bullet sigMapping(s, P') \land$
$\forall r : rels(OM) \bullet relMapping(r, P)$
$=$ [From definition of $syntConformance$, and premise,$\forall r : rels(OM) \bullet relMapping(r, P)$ is valid]
$\forall s : sigs(OM) \cup \{\mathbf{sig}\ X\ \mathbf{extends}\ U\} \bullet sigMapping(s, P')$
$=$ [set theory]
$\forall s : sigs(OM) \bullet sigMapping(s, P') \ \land\ sigMapping((\mathbf{sig}\ X), P')$
$=$ [from $premise(OM, OM', P)$,$\forall s : sigs(S) \bullet s \neq X \Rightarrow sigMapping(s, P') \Leftrightarrow sigMapping$

$(s,\ P)$, which is valid]

$\quad sigMapping((\textbf{sig}\ X), P')$

$=$ [definition of *sigMapping*]

$\quad \exists\, cl : classes(P) \bullet (\textbf{sig}\ X\ \textbf{extends}\ U).name = cl.name\ \wedge$

$\qquad list2set((\textbf{sig}\ X\ \textbf{extends}\ U).extends) \subseteq list2set(cl.extends)$

$=$ [predicate calculus, choosing class X]

$\quad (\textbf{sig}\ X\ \textbf{extends}\ U).name = (\textbf{class}\ X\ \textbf{extends}\ U).name\ \wedge$

$\qquad list2set((\textbf{sig}\ X\ \textbf{extends}\ U).extends) \subseteq list2set((\textbf{class}\ X\ \textbf{extends}\ U).extends)$

$=$ [$(\textbf{sig}\ X\ \textbf{extends}\ U).name = (\textbf{class}\ X\ \textbf{extends}\ U).name$]

$\quad list2set((\textbf{sig}\ X\ \textbf{extends}\ U).extends) \subseteq list2set((\textbf{class}\ X\ \textbf{extends}\ U).extends)$

$=$ [As $first((\textbf{sig}\ X\ \textbf{extends}\ U).extends) = U \in list2set((\textbf{class}\ X\ \textbf{extends}\ U).extends)$]

$\quad true$

□

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement. In this case, `X` $\notin$ *Rep*.

1. No method interface is changed, thus from $premise(OM,\ OM',\ P)$ there are no methods in *Own* with *Rep* return types;

2. No inherited methods are added, thus from $premise(OM,\ OM',\ P)$ no inherited methods have *Rep* parameters;

3. Same as above, thus from $premise(OM,\ OM',\ P)$, *Rep* classes do not inherit methods from non-*Rep* classes;

4. No public fields of *Own* classes are used outside their declaring module, thus from $premise(OM,\ OM',\ P)$ no `e.f` is seen, unless `e` is **self**;

5. No **new** is changed, thus from $premise(OM,\ OM',\ P)$ *Rep* instance is created outside *Own* classes;

6. No method call is affected, thus from $premise(OM,\ OM',\ P)$, `e.m` calls within *Own* or *Rep* do not have *Rep* parameters.

□

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. If `X` is already declared, we rename it to `X'`, which is a straightforward equivalence;

2. Law 3 *Class Elimination* (R-L) is applied, whose provisos are valid:

   (a) The name of class `X` is distinct from those of all classes declared in CT;

(b) The superclass of `X` is `U`, which is correctly declared, from the syntactic conformance;

(c) The field and method names declared by `X` are not declared by its superclasses in CT; `X` is empty.

3. Law 4 *new superclass* (R-L) is applied, whose precondition is valid:

   (a) Variables of type $T \leq U$ are not involved in tests with `X`, as this class has just been included.

$\square$

Semantic conformance

First we prove a few auxiliary lemmas as they are used in the proof.

Lemma E.7. The semantics of OM' can be defined in terms of $semantics(OM)$ as follows.

$$\begin{array}{l} semantics(OM') = \{i \oplus (\textbf{sig}\ X).name \mapsto i.mapSig(U) - (i.mapSig(S) \cup i.mapSig(T)) \\ \quad \mid i \in semantics(OM)\} \end{array}$$

Proof. We start from the definition of $semantics(OM')$.

$$\begin{array}{l}
semantics(OM') \\
= [\text{definition}] \\
\quad \{i : Interpretation \mid satisfyImpInvs(OM', i) \wedge\ satisfyExpInvs(OM', i)\} \\
= [\text{from definition of } satisfyImpInvs(OM', i)] \\
\quad \{i : Interpretation \mid \\
\quad\ (\forall\, s : sigs(OM') \bullet i.mapSig(s.name) \subseteq i.mapSig((super(s)).name)) \wedge \\
\quad\ satisfyExpInvs(OM', i)\} \\
= [\text{from definitions of } OM, OM', sigs(OM') = sigs(OM) \cup (\textbf{sig}\ X)] \\
\quad \{i : Interpretation \mid \\
\quad\ (\forall\, s : sigs(OM) \cup (\textbf{sig}\ X) \bullet \\
\quad\quad i.mapSig(s.name) \subseteq i.mapSig((super(s)).name)) \wedge \\
\quad\ satisfyExpInvs(OM', i)\} \\
= [\text{set theory}] \\
\quad \{i : Interpretation \mid \\
\quad\ (\forall\, s : sigs(OM) \bullet i.mapSig(s.name) \subseteq i.mapSig((super(s)).name) \wedge \\
\quad\ i.mapSig((\textbf{sig}\ X).name) \subseteq i.mapSig(super(\textbf{sig}\ X))) \\
\quad \wedge\ satisfyExpInvs(OM', i)\} \\
= [\text{from definition of } OM',\ list2set((\textbf{sig}\ X).extends) \subseteq list2set((\textbf{sig}\ U).extends)] \\
\quad \{i : Interpretation \mid \\
\quad (\forall\, s : sigs(OM) \bullet i.mapSig(s.name) \subseteq i.mapSig((super(s)).name)) \\
\quad \wedge\ satisfyExpInvs(OM', i)\} \\
= [\text{from definition of } satisfyImpInvs(OM)]
\end{array}$$

$\{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i) \wedge \mathit{satisfyExpInvs}(OM', i)\}$
$=$ [from definition of $\mathit{satisfyExpInvs}(OM', i)$]
$\{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i) \wedge \forall f : \mathit{factInvs}(OM') \bullet$
$\mathit{satisfyFormula}(f, i)\}$
$=$ [from definitions of OM, OM',$\mathit{factInvs}(OM') = \mathit{factInvs}(OM) \cup (X = U - S - T)$]
$\{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i) \wedge$
$\forall f : \mathit{factInvs}(OM) \cup \{X = U - S - T\} \bullet \mathit{satisfyFormula}(f, i)\}$
$=$ [set theory]
$\{i : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i) \wedge$
$\forall f : \mathit{factInvs}(OM) \bullet \mathit{satisfyFormula}(f, i) \wedge$
$\mathit{satisfyFormula}(\{X = U - S - T\}, i)\}$
$=$ [from definition of $\mathit{semantics}(OM)$]
$\{i : \mathit{Interpretation} \mid i \in \mathit{semantics}(OM) \wedge \mathit{satisfyFormula}(\{X = U - S - T\}, i)\}$
$=$ [From Lemma 7.5, where $v = i.\mathit{mapSig}(U) - (i.\mathit{mapSig}(S) \cup i.\mathit{mapSig}(T)$]
$\{i\ \{i \oplus (\textbf{sig}\ X).\mathit{name} \mapsto i.\mathit{mapSig}(U) - (i.\mathit{mapSig}(S) \cup i.\mathit{mapSig}(T)) \mid i \in \mathit{semantics}(OM)\}$

Lemma E.8. :For OM' and P':
$\mathit{heaps}(P', \mathit{filter}) =$
$\{h \oplus (\textbf{class}\ U).\mathit{name} \mapsto h.\mathit{mapClass}(U) \oplus (\textbf{class}\ X).\mathit{name} \mapsto h.\mathit{mapClass}((\textbf{class}\ U).\mathit{name}) - (h.\mathit{mapClass}((\textbf{class}\ X).\mathit{name}) \cup h.\mathit{mapClass}((\textbf{class}\ T).\mathit{name})) \mid h \in \mathit{heaps}(P, \mathit{filter})\}$

Proof. From definition of P'
$\mathit{heaps}(P', \mathit{filter}) =$
$\{h' \mid \forall s : \mathit{sigs}(OM') - U - X \bullet h'.\mathit{mapClass}(s) = h.\mathit{mapClass}(s) \mid h \in \mathit{heaps}(P, \mathit{filter})\}$
$=$ [From the modeling law and property of inheritance]
$\{h \oplus (\textbf{class}\ U).\mathit{name} \mapsto h.\mathit{mapClass}((\textbf{class}\ S).\mathit{name}) \cup h.\mathit{mapClass}((\textbf{class}\ T).\mathit{name}) \cup h.\mathit{mapClass}(X) \oplus (\textbf{class}\ X).\mathit{name} \mapsto h.\mathit{mapClass}(U) - (h.\mathit{mapClass}(X) \cup h.\mathit{mapClass}(T)) \mid h \in \mathit{heaps}(P, \mathit{filter})\}$
$=$ [set theory: $S \cup T \cup X = S \cup T \cup (U - S - T) = U$]
$\{h \oplus (\textbf{class}\ U).\mathit{name} \mapsto h.\mathit{mapClass}((\textbf{class}\ U).\mathit{name}) \cup \oplus(\textbf{class}\ X).\mathit{name} \mapsto h.\mathit{mapClass}(U) - (h.\mathit{mapClass}(X) \cup h.\mathit{mapClass}(T)) \mid h \in \mathit{heaps}(P, \mathit{filter})\}$

Main proof. Lemma 7.4 is then proved, using the result of the auxiliary lemmas and premise.

$\mathit{semanticConformance}(OM', P')$
$=$ [definition]
$\forall h : \mathit{heaps}(P', \mathit{filter}) \bullet$
$\exists\ i : \mathit{semantics}(OM') \bullet$
$\forall s : \mathit{sigs}(OM') \bullet\ i.\mathit{mapSig}(s.\mathit{name}) = h.\mathit{mapClass}(s.\mathit{name}) \wedge$
$\forall r : \mathit{rels}(OM') \bullet\ i.\mathit{mapRel}(r.\mathit{name}) = h.\mathit{mapField}(r.\mathit{name})$
$=$ [from definitions of OM,OM',$\mathit{rels}(OM') = \mathit{rels}(OM)$,
$\mathit{sigs}(OM') = \mathit{sigs}(OM) \cup \{\textbf{sig}\ X\ \textbf{extends}\ U\}$]
$\forall h : \mathit{heaps}(P', \mathit{filter}) \bullet$

$\exists\ i : semantics(OM')\bullet$
$\forall s : sigs(OM) \cup \{\textbf{sig}\ X\ \textbf{extends}\ U\} \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [set theory]

$\forall h : heaps(P', filter)\bullet$
$\exists\ i : semantics(OM')\bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$i.mapSig(X) = h.mapClass(X) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [Lemma E.7 replaces $semantics(OM')$]

$\forall h : heaps(P', filter)\bullet$
$\exists\ i : \{i \oplus (\textbf{sig}\ U).name \mapsto i.mapSig(U) - (i.mapSig(X)\ \cup\ i.mapSig(T))\ \mid\ i \in semantics(OM)\}\bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$i.mapSig(X) = h.mapClass(X) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [Lemma E.8 replaces $heaps(P', filter)$]

$\forall h : \{h \oplus (\textbf{class}\ U).name \mapsto h.mapClass(U) \oplus (\textbf{class}\ X).name \mapsto h.mapClass(U) - (h.mapClass(X)\ \cup\ h.mapClass(T))\ \mid\ h \in heaps(P, filter)\}\bullet$
$\exists\ i : \{i\ \oplus\ (\textbf{class}\ X).name \mapsto i.mapSig(U) - (i.mapSig(S)\ \cup\ i.mapSig(T))\ \mid\ i \in semantics(OM)\}\bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$i.mapSig(X) = h.mapClass(X) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [From predicate calculus, choosing an arbitrary $h1$ following the quantification properties]

$\exists\ i : \{i\ \oplus\ (\textbf{class}\ X).name \mapsto i.mapSig(U) - (i.mapSig(S)\ \cup\ i.mapSig(T))\ \mid\ i \in semantics(OM)\}\bullet$
$\forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$i.mapSig(X) = h1.mapClass(X) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = mapField(h1)(r.name)$

= [Existential quantification, choosing $i1$ following the quantification properties]

$\forall s : sigs(OM) \bullet\ i1.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$i1.mapSig(X) = h1.mapClass(X) \wedge$
$\forall r : rels(OM) \bullet\ i1.mapRel(r.name) = mapField(h1)(r.name)$

= [From premise, no changes in relations, and all sets but X maintain the mappings]

$i1.mapSig(X) = h1.mapClass(X)$

= [From quantification properties,$i1.mapSig(X) = h1.mapClass(X) = h1.mapClass(U) - (h1.mapClass(S) \cup h1.mapClass(T))$]

$true$

□

E.5 removeSubclass

A subclass is removed, along with adjustments to its hierarchy, mainly its possible field and methods.

Let OM, OM' be any two object models and P, P' two programs as follows:

```
OM                                   OM'
ps                                   ps
sig U { rsU }                        sig U { rsU }
sig S extends U{ rsS }               sig S extends U{ rsS }
sig T extends U{ rsT }        ⟹      sig T extends U{ rsT }
sig X extends U{}                    fact F { forms }
fact F {
  forms
  X = U − S − T
}
```

```
P                                    P'
                                     CT'
CT                                   class U [extends]{..mtsU'}
class U [extends]{..mtsU}            class S [extends]{..mtsS'}
class S [extends]{..mtsS}            class T [extends]{..mtsT'}
class T [extends]{..mtsT}            class Y [extends]{
class Y [extends]{                     adsY; adsX;
  adsY;                                pub String type;
  mtsY}                                mtsY'; mtsX;
class X extends Y{                     bool isX(){
  adsX;                                  result := (self.type =
  mtsX;                       ⟹      ”X” ∨ self.type = ”W”)
  constr {c}                           }
}                                      void newX(){
class W extends X{                       c; self.type := ”X”
  ..mtsW'}                             }
  constr {b}                         }
}                                    class W extends Y{
                                       ..mtsW';
                                       constr {b;  self.type :=
                                     ”W” }
                                     }
```

where:

$S, T, Y \leq U$;

$CT''''', A' = CT, A[pub\ Type\ f/pri\ Type\ f], A \in hierarchy(X, object)$;

$mt' = mt[body(m)/super.m()], mt \in meths(hierarchy(X, object))$;

$mtY' = mtY[\textbf{if}\ \neg(\textbf{self is}\ X)\ \textbf{then}\ b\ \textbf{else}\ b']$,

$mtY \in name(mtsX \cap mtsY), b = body(Y.m), b' = body(X.m)$

$CT'''' = CT'''''[Y\ id/X\ id]$;

$CT''' = CT''''[\{exp.isX()\}cmd[exp]/cmd[(X)exp]]$;
$CT'' = CT'''[exp.isX()/exp$ **is** $X]$;
$CT' = CT''[$**unpack** $x;\ x :=$ **new** $Y;\ x.newX();$ **pack** $x\ /\ x :=$ **new** $X]$;

Syntactic conformance

Proof. The goal is to prove validity of the predicate, based on the premise predicates, mainly $syntConformance(OM, P)$.

$syntConformance(OM', P')$
= [definition]
$\quad \forall s : sigs(OM') \bullet sigMapping(s, P') \wedge$
$\quad \forall r : rels(OM') \bullet relationMapping(r, P') \wedge$
$\quad abstractConstraint(OM', P')$
= [from definition of OM, OM', $sigs(OM') = sigs(OM) - \{$**sig** $X\}$ and $rels(OM') = rels(OM)$]
$\quad \forall s : sigs(OM) - \{$**sig** $X\} \bullet sigMapping(s, P') \wedge$
$\quad \forall r : rels(OM) \bullet relMapping(r, P') \wedge$
$\quad abstractConstraint(OM', P')$
= [From definitions of OM', P, P',$\forall r : rels(OM') \bullet\ relMapping(r, P) \Leftrightarrow relMapping(r, P')$]
$\quad \forall s : sigs(OM) - \{$**sig** $X\} \bullet sigMapping(s, P') \wedge$
$\quad \forall r : rels(OM) \bullet relMapping(r, P) \wedge$
$\quad abstractConstraint(OM', P')$
= [From definitions of OM', P' and premise, $abstractConstraint(OM', P')$ is valid, as the only **new** command added for classes is **new** Y, which has no subtype in the model]
$\quad \forall s : sigs(OM) - \{$**sig** $X\} \bullet sigMapping(s, P') \wedge$
$\quad \forall r : rels(OM) \bullet relMapping(r, P)$
= [from definition of $syntConformance$, and premise,$\forall r : relations(OM) \bullet relMapping(r, P)$ is valid]
$\quad \forall s : sigs(OM) - \{$**sig** $X\} \bullet sigMapping(s, P') \wedge$
= [set theory]
$\quad \forall s : sigs(OM) \bullet ($**sig** $X \neq s) \Rightarrow sigMapping(s, P')$
= [predicate calculus, choosing arbitrary $s1$]
$\quad ($**sig** $X \neq s1) \Rightarrow sigMapping(s1, P')$
= [Assuming (**sig** $X \neq s1$) as a premise]
$\quad sigMapping(s1, P')$
= [Definition of $sigMapping$]
$\quad \exists cl : classes(P') \bullet s1.name = cl.name \wedge$
$\qquad list2set(s1.extends) \subseteq list2set(cl.extends)$
= [From P, P' no other class is removed but X, from $premise(OM,\ OM',\ P)$ and additional premise, there is a class with the same name and the its superclasses are unchanged]
$\quad true$

□

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement. In this case, `S`,`U` $\in$ *Own* and `U`,`T` $\in$ *Rep*.

1. Class `Y` is the only one to receive new methods. If `Y` $\in$ *Own*, then from $premise(OM, OM', P)$ methods previously in `X` could never have *Rep* return types;
2. No inherited methods are added, thus from $premise(OM, OM', P)$ no inherited methods have *Rep* parameters;
3. Same as above, thus from $premise(OM, OM', P)$, *Rep* classes do not inherit methods from non-*Rep* classes;
4. No public fields of *Own* classes are used outside their declaring module, thus from $premise(OM, OM', P)$ no `e.f` is seen, unless `e` is **self**;
5. From premise, if `x:=`**new** `X` was outside *Own* classes, `X` $\notin$ *Rep*. Assuming the command is outside *Own*, it is impossible to have `X` $\notin$ *Rep* and `Y` $\in$ *Rep*, since all subclasses of *Rep* classes are also included. Therefore, property is maintained;
6. From premise, `e.m(..)` within *Own* or *Rep* does not have *Rep* parameters; no changes in parameters or *Rep* are made, so property is maintained.

$\square$

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. For each field in superclasses of `X`, Law 27 *change visibility: from pri to pub* (L-R), with no precondition;
2. For each superclass of `X`, in a top-down list, replace every method call using **super** with Law 38 *eliminate super* (L-R), whose provisos are valid:
   (a) **super** and private fields does not appear in the inline code, as the iteration is top-down.
3. Each expression whose type is `X` is cast appropriately, using Law 40 *introduce trivial cast in expressions* (L-R), with no provisos;
4. For each field in `X`, two options:
   - Law 37 *move field to superclass* (L-R) is applied, with the following provisos:
     (a) The field is not declared in subclasses of `Y`; otherwise, application fails.
   - If it fails, Refactoring 5 *pull up field* (L-R) with the following provisos:

   (a) simplified precondition: The field name is not declared in `Y` or superclasses, as shadowing is not allowed.

5. For each method redefinition in `X`, Law 35 *move redefined method to superclass* (L-R), with the following provisos:

   (a) **super** and private fields do not appear in `X` methods;

   (b) All ocurrences of **self** are cast with `X`.

6. For each original method in `X`, apply following laws:

   - Declare the method in all subclasses of `Y`, if it is not previously declared, with Law 34 *method elimination*;
   - Apply Law 36 *move original method to superclass*, with the following provisos:

     (a) **super** and private fields do not appear in `X` methods;

     (b) The method is declared in the subclasses of `Y` with the same parameters, and all occurrences of **self** are cast with `X`.

7. Each declaration typed as `X` is changed to type `Y`, with the following laws: Law 29 *change field type* (L-R), Law 30 *change variable type* (L-R), Law 31 *change parameter type* (L-R) and Law 32 *change return type* (L-R). Their provisos are pretty similar, for instance, *change field type* provisos are valid:

   (a) $X \leq \textbf{super}(X)$, and all C expressions are cast, from the previous step.

8. Each cast to `X` is eliminated with Law 39 *eliminate cast of expressions* (L-R), with no provisos.

9. Rule *eliminateTypeTests*

   - Method `isX` is introduced with Law 34 *method elimination* (R-L), with precondition:

     (a) `isX` is not declared in **super**`(X)` nor in any superclass or subclass of **super**`(X)` in CT.

   - In each type test, include a boolean local variable with Law 22 *var block-value* (L-R), with provisos:

     (a) the variable is fresh;

     (b) `x` **is**`X` is not in the left-hand side of assignments or target of method call.

   - In each type test, include a result local variable for receiving the test's boolean value, with 23 *var block-res* (L-R), with provisos:

     (a) the variable is fresh;

     (b) the boolean variable is not in the right-hand side of assignments, parameter or target of method call.

   - In each type test, encapsulate the assignment from the test into a parammeterized command, with 24 *pcom elimination-res* (R-L), with provisos:

   (a) the new variable is fresh;
   (b) the boolean variable is not in the left-hand side of assignments or target of method call.

- In each type test, encapsulate the assignment from the test into a parameterized command, with 24 *pcom elimination-res* (R-L), with provisos:
   (a) the new variable is fresh;
   (b) the boolean variable is not in the left-hand side of assignments or target of method call.
- In each type test parameterized, include an **if** statements with 25 *if identical guarded commands* (R-L), with no precondition.
- in each type test parameterized command, replace the command by a method call to `isX` with Law 33 *method call elimination* (R-L), with provisos:
   (a) `isX` is not redefined and the command does not contain references to super;
   (b) no fields are used in `isX`.
- The `type` field is introduced, with Law 28 *field elimination* (R-L), with provisos:
   (a) `type` is not declared in `Y` or in its super or subclasses in CT.
- For each subclass of **super**`(X)`, a constructor is added, if not previously declared, with Law 34 *method elimination*;
- These subclasses have initialization added to their constructors **self**`.type:=..`, with simulation. In this case, the coupling invariant is *true*:
   - Constructors in these subclasses force changes to field `type`, establishing the invariant;
   - No changes in methods, which also maintains the invariant.
- `isX` in `Y` is changed by replacing the type test with a disjunction test on the `type` field. Simulation is used, with coupling invariant **self is** `X`$=\bigvee$`(`**self**`.type = "T", T` $\leq$ `X)`:
   - The constructor in `Y` is not changes, establishing the invariant;
   - Only method `isX` is changed, replacing **self is** `X` by the indicated disjunction, maintaining the invariant.

10. Rule *eliminateNew*

- After applying the **new** command syntactic sugar to all `X` instantiations, `newX` is moved to the superclass with Law 36 *move original method to superclass* (L-R), with valid provisos:
   (a) **super** and private fields do not appear in the method;
   (b) `newX` is not declared in any superclass of `Y` in CT;
   (c) `newX` is not declared in `Y`;
   (d) `newX` does not contain uncast occurrences of **self**.

- Instantiations of `X` are replaced by **new**`Y` with Law 4 *new superclass* (L-R), with valid provisos:
  - (a) `X` is not used in type casts or type tests in CT for expressions of type `Y`;
  - (b) **new**`X` is assigned only to fields or variables of type `Y` or any supertype of `A`.

11. For each immediate subclass of `X`, Law 26 *change superclass: from an empty class to immediate superclass* (R-L), with no provisos, for changing their superclasses from `X` to `Y`.

12. Law 3 *class elimination* (L-R) is applied, whose provisos are valid:

    (a) `X` is not referred to in CT.

□

Semantic conformance

First we prove a few auxiliary lemmas as they are used in the proof.

Lemma E.9. The semantics of OM' can be defined in terms of $semantics(OM)$ as follows.

$$semantics(OM') = i : \{i : Interpretation \mid i \in semantics(OM) \land i \rhd (\textbf{sig}\ X).name\}$$

Proof. We start from the definition of $semantics(OM')$.
$semantics(OM')$
$=$ [definition]
$\{i : Interpretation \mid satisfyImpInvs(OM', i) \land\ satisfyExpInvs(OM', i)\}$
$=$ [from definition of $satisfyImpInvs(OM', i)$]
$\{i : Interpretation \mid$
$(\forall s : sigs(OM') \bullet i.mapSig(s.name) \subseteq i.mapSig((super(s)).name)) \land$
$satisfyExpInvs(OM', i)\}$
$=$ [from definitions of OM, OM',$sigs(OM') = sigs(OM) - (\textbf{sig}\ X)$]
$\{i : Interpretation \mid$
$(\forall s : sigs(OM) - (\textbf{sig}\ X) \bullet$
$i.mapSig(s.name) \subseteq i.mapSig((super(s)).name)) \land$
$satisfyExpInvs(OM', i)\}$
$=$ [set theory]
$\{i : Interpretation \mid$
$(\forall s : sigs(OM) \bullet$
$s \neq (\textbf{sig}\ X)\ \land\ i.mapSig(s.name) \subseteq i.mapSig((super(s)).name))$
$\land\ satisfyExpInvs(OM', i)\}$
$=$ [without signature `X`, the implicit invariant is maintained for all other signatures]
$\{i : Interpretation \mid$
$(\forall s : sigs(OM) \bullet$

$i.mapSig(s.name) \subseteq i.mapSig((super(s)).name))$
$\wedge\ satisfyExpInvs(OM', i)\}$
$=$ [from definition of $satisfyImpInvs(OM)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge\ satisfyExpInvs(OM', i)\}$
$=$ [from definition of $satisfyExpInvs(OM', i)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge \forall f : factInvs(OM') \bullet$
$satisfyFormula(f, i)\}$
$=$ [from definitions of OM, OM', $factInvs(OM') = factInvs(OM) - (X = U - S - T)$]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge$
$\forall f : factInvs(OM) - \{r = x.y\} \bullet satisfyFormula(f, i)\}$
$=$ [set theory]
$\{i : Interpretation \mid satisfyImpInvs(OM, i) \wedge$
$\forall f : factInvs(OM) \bullet (f \neq \{X = U - S - T\}) \Rightarrow satisfyFormula(f, i)\}$
$=$ [from Lemma 7.8, i is part of the semantics of OM]
$\{i : Interpretation \mid i \in semantics(OM) \wedge i \trianglerighteq (\mathbf{sig}\ X).name\}$

Lemma E.10. For P and P':
$heaps(P', filter) = \{h : Heap \mid h \in heaps(P, filter) \wedge h \trianglerighteq X.name\}$

Proof. The changed commands (tests and casts) do not add or remove heaps; all X instances that are now Y instances are still mapped by U in the heap, and the other classes in the hierarchy are abstract.

Main proof. Lemma 7.4 is then proved, using the result of the auxiliary lemmas and premise.

$=$ [definition]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : semantics(OM') \bullet$
$\forall s : sigs(OM') \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\forall r : rels(OM') \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$=$ [from $premise(OM,\ OM',\ P)$, $rels(OM') = rels(OM)$,$sigs(OM') = sigs(OM) - \{\mathbf{sig}\ X\}$]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : semantics(OM') \bullet$
$\forall s : sigs(OM) - \{\mathbf{sig}\ X\} \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$=$ [set theory]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : semantics(OM') \bullet$
$\forall s : sigs(OM) \bullet\ (\mathbf{sig}\ X) \neq s \Rightarrow\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$=$ [Lemma E.9 replaces $semantics(OM')$]
$\forall h : heaps(P', filter) \bullet$
$\exists\ i : \{i : interpretation \mid i \in semantics(OM) \wedge i \trianglerighteq X.name\} \bullet$
$\forall s : sigs(OM) \bullet\ (\mathbf{sig}\ X) \neq s \Rightarrow\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$

$\forall\, r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [Lemma E.10 replaces $heaps(P', filter)$]

$\forall\, h : \{h : Heap \mid h \in heaps(P, filter) \wedge h \trianglerighteq X.name\} \bullet$
$\quad \exists\ i : \{i : Interpretation \mid i \in semantics(OM) \wedge i \trianglerighteq X.name\} \bullet$
$\quad\quad \forall\, s : sigs(OM) \bullet\ (\mathbf{sig}\ U) \neq s \Rightarrow\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad \forall\, r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name)$

= [From predicate calculus, choosing an arbitrary $h1$]

$\exists\ i : \{i : Interpretation \mid i \in semantics(OM) \wedge i \trianglerighteq X.name\} \bullet$
$\quad \forall\, s : sigs(OM) \bullet\ (\mathbf{sig}\ X) \neq s \Rightarrow\ i.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\quad \forall\, r : rels(OM) \bullet\ i.mapRel(r.name) = mapField(h1)(r.name)$

= [For existential quantification, we choose interpretation $i1$ which repeats the mappings for model names in $h1$, removing the mapping from X signature]

$\forall\, s : sigs(OM) \bullet\ (\mathbf{sig}\ X) \neq s \Rightarrow\ i1.mapSig(s.name) = h1.mapClass(s.name) \wedge$
$\forall\, r : rels(OM) \bullet\ i.mapRel1)(r.name) = mapField(h1)(r.name)$

= [from $premise(OM,\ OM',\ P)$]

$\forall\, s : sigs(OM) \bullet\ (\mathbf{sig}\ U) \neq s \Rightarrow\ i1.mapSig(s.name) = h1.mapClass(s.name)$

= [from $premise(OM,\ OM',\ P)$, every original model signature and class is mapped to the same values, except X]

true

□

E.6 fromSetToOptionalField

In this strategy, a specific field is changed from a set representation to a single variable, based on an invariant from the model. Let OM, OM' be any two object models and P, P' two programs as follows:

OM

```
sig S [extends]{
   rs
   r : set T
}
fact F{
   forms
   all s : S | lone s.r
}
```

$\Longrightarrow$

OM'

```
ps
sig S [extends]{
   rs
   r : lone T
}
fact F{
   forms
}
```

P

```
CT
class S [extends]{
   ads;  set T r;
   mts
}
```

$\Longrightarrow$

P'

```
CT
class S [extends]{
   ads;  T r;
   mts'
}
```

where:

$mts'' = mts[\textbf{if } (exp = \varnothing) \textbf{ then self}.r := \textbf{null else self}.r := \textbf{elem}(exp)$
$/\textbf{self}.r := exp]$

$mts' = mts''[\textbf{if } (\textbf{self}.r = \textbf{null}) \textbf{ then } cmd[exp([\varnothing/\textbf{self}.r])] \textbf{else } cmd[exp[\{\textbf{self}.r\}/\textbf{self}.r]]$
$/cmd[exp[\textbf{self}.r]]]$

Syntactic conformance

Proof. The goal is to prove validity of the predicate, based on the premise predicates, mainly $syntConformance(OM, P)$.

$syntConformance(OM', P')$
$=$ [definition]
$\forall s : sigs(OM') \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM') \bullet relMapping(r, P') \wedge$
$abstractConstraint(OM', P')$
$=$ [from definitions of OM, OM',$sigs(OM') = sigs(OM)$ and $rels(OM') = rels(OM) - \{r : \textbf{set } T\} \cup \{r : \textbf{lone } T\}$]
$\forall s : sigs(OM) \bullet sigMapping(s, P') \wedge$
$\forall r : rels(OM) - \{r : \textbf{set } T\} \cup \{r : \textbf{lone } T\} \bullet relMapping(r, P') \wedge$
$abstractConstraint(OM', P')$
$=$ [From definitions of $OM', P, P', \forall r : sigs(OM) \bullet sigMapping(s, P) = sigMapping(s, P')$]
$\forall s : sigs(OM) \bullet sigMapping(s, P) \wedge$
$\forall r : rels(OM) - \{r : \textbf{set } T\} \cup \{r : \textbf{lone } T\} \bullet relMapping(r, P') \wedge$
$abstractConstraint(OM', P')$
$=$ [From definitions of OM', P', $abstractConstraint(OM', P') = abstractConstraint(OM, P)$, as hierarchies are not affected]
$\forall s : sigs(OM) \bullet sigMapping(s, P) \wedge$
$\forall r : rels(OM) - \{r : \textbf{set } T\} \cup \{r : \textbf{lone } T\} \bullet relMapping(r, P')$
$=$ [From definition of $syntConformance$, and premise,$\forall s : sigs(OM) \bullet sigMapping(s, P)$ is valid]
$\forall r : rels(OM) - \{r : \textbf{set } T\} \cup \{r : \textbf{lone } T\} \bullet relMapping(r, P')$
$=$ [set theory]
$\forall r : rels(OM) \bullet relMapping(r, P') \wedge relMapping((r : \textbf{lone } T), P')$
$=$ [from $premise(OM, OM', P)$,$\forall r : rels(OM) \Rightarrow relMapping(r, P') \Leftrightarrow relMapping(r, P)$, which is valid]
$relMapping((r : \textbf{lone } T), P')$
$=$ [definition of $relMapping$, already simplified]
$\exists f : fields(P') \bullet (r : \textbf{lone } T).name = f.name \wedge$
$(r : \textbf{lone } T).leftType = f.leftType \wedge (r : \textbf{lone } T).rightType = f.rightType \wedge$
$(\neg isScalar(r : \textbf{lone } T) \Rightarrow \neg isScalar(f))$
$=$ [Predicate calculus, choose newly-introduced field r in P']
$(r : \textbf{lone } T).name = (T\ r).name \wedge$
$(r : \textbf{lone } T).leftType = (T\ r).leftType \wedge (r : \textbf{lone } T).rightType = (T\ r).rightType \wedge$
$(\neg isScalar(r : \textbf{lone } T) \Rightarrow \neg isScalar(T\ r))$

= [Name and types match]
$(\neg isScalar(r : \textbf{lone}\ T) \Rightarrow \neg isScalar(T\ r))$
= [From definitions of OM', P', either field and relation are scalar]
true

□

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement. In this case, no changes for confinement are seen.

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. Field `T r'` is introduced with Law 28 *field elimination* (R-L), with the following provisos:

   (a) `r'` is not declared in `S` or in its super or subclasses in CT.

2. As representation independence is proved for confined programs with references [8], refinement is given by simulation – the coupling invariant is $(\textbf{self}.r \Longrightarrow \textbf{self}.r' = \textbf{null}) \wedge (\neg(\textbf{self}.r = \varnothing) \Rightarrow \textbf{elem}(\textbf{self}.r) = \textbf{self}.r')$.

   - Constructors in `S` force changes to field `r` to be duplicated for `r'`, but for a single value, so the invariant is established. Also, expressions using `r` are always replaced by the respective scalar value, with an **if** statement;
   - Same for methods, which maintains the invariant.

3. We rename `r'` in class `S` to `r`, which is a straightforward equivalence;

□

Semantic conformance

Proof. Since $semantics(OM) = semantics(OM')$ and $heaps(P, filter) = heaps(P', filter)$, it is straightforward to prove that the semantic conformance is maintained.

E.7 splitField

An alternative path between two classes is created, adding a new classes and two fields, along with an invariant defining how instanced are linked.

Let OM, OM' be any two object models and P, P' two programs as follows:

OM

```
ps
sig S [extends]{
   rs
   r : set T
}
fact F{
   forms
}
```

$\Longrightarrow$

OM′

```
ps
sig S [extends]{
   rs
   r : set T,
   x : set U
}
sig U{
   y : set T
}
fact F{
   forms
   r = x.y
}
```

P

```
CT
class S [extends]{
   ads; set T r;
   W x;
   constr {
      b'}
   mts
}
class U [extends]{..}
```

$\Longrightarrow$

P′

```
CT'
class S [extends]{
   ads; set T r;
   W x';
   set U x;
   constr {
      b';
      self.x := {new U} }
   mts'
}
class U' [extends]{..}
class U{
   set T y;
   unit setY(T e){
      self.y := e}
   T getY(){
      result := self.y}
}
```

where:
$CT', mts'' = CT, mts[U'/U]$
$mts' = mts''[$**unpack self**; **self**$.r := exp$; **elem**(**self**$.x).setY(exp)$; **pack self**/**self**$.r := exp]$

Syntactic conformance

Proof. The goal is to prove validity of the predicate, based on the premise predicates, mainly *syntConformance*(OM, P).

$\mathit{syntConformance}(OM', P')$
$= $ [definition]
$\quad \forall s : \mathit{sigs}(OM') \bullet \mathit{sigMapping}(s, P') \wedge$
$\quad \forall r : \mathit{rels}(OM') \bullet \mathit{relMapping}(r, P') \wedge$
$\quad \mathit{abstractConstraint}(OM', P')$
$= $ [from definitions of OM, OM', $\mathit{sigs}(OM') = \mathit{sigs}(OM) \cup \{\mathbf{sig}\ U\}$ and $\mathit{rels}(OM') = \mathit{rels}(OM) \cup \{x : \mathbf{set}\ U\} \cup \{y : \mathbf{set}\ T\}$]
$\quad \forall s : \mathit{sigs}(OM) \cup \{\mathbf{sig}\ U\} \bullet \mathit{sigMapping}(s, P') \wedge$
$\quad \forall r : \mathit{rels}(OM) \cup \{x : \mathbf{set}\ U\} \cup \{y : \mathbf{set}\ T\} \bullet \mathit{relMapping}(r, P') \wedge$
$\quad \mathit{abstractConstraint}(OM', P')$
$= $ [From definitions of OM', P', $\mathit{abstractConstraint}(OM', P') = \mathit{abstractConstraint}(OM, P)$, as hierarchies are not affected]
$\quad \forall s : \mathit{sigs}(OM) \cup \{\mathbf{sig}\ U\} \bullet \mathit{sigMapping}(s, P') \wedge$
$\quad \forall r : \mathit{rels}(OM) \cup \{x : \mathbf{set}\ U\} \cup \{y : \mathbf{set}\ T\} \bullet \mathit{relMapping}(r, P')$
$= $ [set theory]
$\quad \forall s : \mathit{sigs}(OM) \bullet \mathit{sigMapping}(s, P') \wedge$
$\quad \mathit{sigMapping}((\mathbf{sig}\ U), P') \wedge$
$\quad \forall r : \mathit{rels}(OM) \bullet \mathit{relMapping}(r, P') \wedge$
$\quad \mathit{relMapping}((x : \mathbf{set}\ U), P') \wedge$
$\quad \mathit{relMapping}((y : \mathbf{set}\ T), P')$
$= $ [From definitions of $OM', P, P', \forall r : \mathit{sigs}(OM) \bullet \mathit{sigMapping}(s, P) = \mathit{sigMapping}(s, P')$, same for relations]
$\quad \forall s : \mathit{sigs}(OM) \bullet \mathit{sigMapping}(s, P) \wedge$
$\quad \mathit{sigMapping}((\mathbf{sig}\ U), P') \wedge$
$\quad \forall r : \mathit{rels}(OM) \bullet \mathit{relMapping}(r, P) \wedge$
$\quad \mathit{relMapping}((x : \mathbf{set}\ U), P') \wedge$
$\quad \mathit{relMapping}((y : \mathbf{set}\ T), P')$
$= $ [From definition of $\mathit{syntConformance}$, and premise, $\forall s : \mathit{sigs}(OM) \bullet \mathit{sigMapping}(s, P)$ is valid (same for $\mathit{rels}(OM)$)]
$\quad \mathit{sigMapping}(\mathbf{sig}\ U\}, P') \wedge$
$\quad \mathit{relMapping}(x : \mathbf{set}\ U), P') \wedge$
$\quad \mathit{relMapping}(y : \mathbf{set}\ T), P')$
$= $ [definition of $\mathit{sigMapping}$]
$\quad \exists\, cl : \mathit{classes}(P') \bullet (\mathbf{sig}\ U).\mathit{name} = cl.\mathit{name} \wedge$
$\qquad \mathit{list2set}((\mathbf{sig}\ U).\mathit{extends}) \subseteq \mathit{list2set}(cl.\mathit{extends})$
$\quad \mathit{relMapping}((x : \mathbf{set}\ U), P') \wedge$
$\quad \mathit{relMapping}((y : \mathbf{set}\ T), P')$
$= $ [from predicate calculus, choosing cl to be the newly-introduced `U` class]
$\quad (\mathbf{sig}\ U).\mathit{name} = (\mathbf{class}\, U).\mathit{name} \wedge$
$\quad \mathit{list2set}((\mathbf{sig}\ U).\mathit{extends}) \subseteq \mathit{list2set}((\mathbf{class}\, U).\mathit{extends})$
$\quad \mathit{relMapping}((x : \mathbf{set}\ U), P') \wedge$
$\quad \mathit{relMapping}((y : \mathbf{set}\ T), P')$
$= $ [from definition of P', name and superclasses are the same]
$\quad \mathit{relMapping}((x : \mathbf{set}\ U), P') \wedge$
$\quad \mathit{relMapping}((y : \mathbf{set}\ T), P')$
$= $ [From definition of $\mathit{relMapping}$, already simplified]

$\exists f1 : \mathit{fields}(P') \bullet (r : \textbf{set}\ T).\mathit{name} = f1.\mathit{name} \wedge$
$(x : \textbf{set}\ U).\mathit{leftType} = f1.\mathit{leftType}\ \wedge\ (x : \textbf{set}\ U).\mathit{rightType} = f1.\mathit{rightType} \wedge$
$(\mathit{isScalar}(x : \textbf{set}\ U) \Rightarrow \mathit{isScalar}(f1)) \wedge$
$\exists f2 : \mathit{fields}(P') \bullet (y : \textbf{set}\ T).\mathit{name} = f2.\mathit{name} \wedge$
$(y : \textbf{set}\ T).\mathit{leftType} = f2.\mathit{leftType}\ \wedge\ (y : \textbf{set}\ T).\mathit{rightType} = f2.\mathit{rightType} \wedge$
$(\mathit{isScalar}(y : \textbf{set}\ T) \Rightarrow \mathit{isScalar}(f2))$
= [Predicate calculus, choose newly-introduced field x for $f1$ and y for $f2$ in P']
$(r : \textbf{set}\ T).\mathit{name} = (\textbf{set}\ Ux).\mathit{name} \wedge$
$(x : \textbf{set}\ U).\mathit{leftType} = (\textbf{set}\ Ux).\mathit{leftType} \wedge (x : \textbf{set}\ U).\mathit{rightType} = (\textbf{set}\ Ux).\mathit{rightType} \wedge$
$(\mathit{isScalar}(x : \textbf{set}\ U) \Rightarrow \mathit{isScalar}(\textbf{set}\ U\ x)) \wedge$
$(y : \textbf{set}\ T).\mathit{name} = (\textbf{set}\ Ty).\mathit{name} \wedge$
$(y : \textbf{set}\ T).\mathit{leftType} = (\textbf{set}\ Ty).\mathit{leftType} \wedge (y : \textbf{set}\ T).\mathit{rightType} = (\textbf{set}\ Ty).\mathit{rightType} \wedge$
$(\mathit{isScalar}(y : \textbf{set}\ T) \Rightarrow \mathit{isScalar}(\textbf{set}\ T\ y))$
= [Name and types match for both `x,y`]
$(\mathit{isScalar}(x : \textbf{set}\ U) \Rightarrow \mathit{isScalar}(\textbf{set}\ U\ x)) \wedge$
$(\mathit{isScalar}(y : \textbf{set}\ T) \Rightarrow \mathit{isScalar}(\textbf{set}\ T\ y))$
= [From definitions of OM', P', both are declared as set]
true

□

Confinement

Proof. By case analysis on P' for the six static analysis rules of confinement. In this case, `S,U` $\in$ *Own* and `U,T` $\in$ *Rep*.

1. No method interface is changed, thus from $\mathit{premise}(OM,\ OM',\ P)$ there are no methods in *Own* with *Rep* return types;

2. No inherited methods are added, thus from $\mathit{premise}(OM,\ OM',\ P)$ no inherited methods have *Rep* parameters;

3. Same as above, thus from $\mathit{premise}(OM,\ OM',\ P)$, *Rep* classes do not inherit methods from non-*Rep* classes;

4. No public fields of *Own* classes are used outside their declaring module, thus from $\mathit{premise}(OM,\ OM',\ P)$ no `e.f` is seen, unless `e` is **self**;

5. No **new** is changed, thus from $\mathit{premise}(OM,\ OM',\ P)$ *Rep* instance is created outside *Own* classes;

6. The only method call added is within class `S`, calling `setY()`. The object called is typed `U`$\in$ *Own*, so it does not break confinement.

□

Refinement

Proof. For proving refinement, show that all strategy's steps are refinements or apply laws of programming from the catalog presented in Appendix B.

1. We rename `x` in class `S` to `x'`, which is a straightforward equivalence;
2. We rename class `U` to `U'`, which is a straightforward equivalence;
3. Law 3 *class elimination* (R-L) is applied, whose provisos are valid:
   (a) The name of class `U` is distinct from those of all classes declared in CT.
   (b) The superclass appearing in `U` is `Object`.
   (c) The field and method names declared by `U` are not declared by its superclasses in CT, since it is `Object`.
4. Field **setT** `x` is introduced with Law 28 *field elimination* (R-L), with the following provisos:
   (a) `x` is not declared in `S` or in its super or subclasses in CT.
5. As representation independence is proved for confined programs with references [8], refinement is given by simulation – the coupling invariant is $(\mathbf{self}.r = \varnothing \Rightarrow \mathbf{self}.r' = \mathbf{null}) \wedge (\neg(\mathbf{self}.r = \varnothing) \Rightarrow \mathbf{elem}(\mathbf{self}.r) = \mathbf{self}.r')$.
   - Constructors in `S` force `x` to be non-null for every `S` object. Also, writings to field `r` are duplicated for the indirection by calling **self**`.x.setY()`;
   - Methods maintain the invariant.

□

Semantic conformance

First we prove a few auxiliary lemmas as they are used in the proof.

Lemma E.11. The semantics of OM' can be defined in terms of $semantics(OM)$ as follows.

$$\begin{aligned}
&semantics(OM') = \{i' \oplus\ i'' \mid i' \in semantics(OM) \wedge \\
&\quad i'' : Interpretation \mid i''.mapSig(U) = \mathbb{P}\, objValue \wedge \\
&\quad i''.mapRel(x) = \mathbb{P}(i'.mapSig(S) \leftrightarrow i''.mapSig(U)) \wedge \\
&\quad i''.mapRel(y) = \mathbb{P}(i''.mapSig(U) \leftrightarrow i'.mapSig(T)) \wedge \\
&\quad i''.mapRel(x) \mathbin{;} i''.mapRel(y) = i'.mapRel(r)\}
\end{aligned}$$

Proof. We start from the definition of $semantics(OM')$. In this proof, we use a small auxiliary interpretation i'', which provides mappings for `U,x` and `y`.
$semantics(OM')$
$=$ [definition]

$\{i' : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM', i') \land \ \mathit{satisfyExpInvs}(OM', i')\}$
$=$ [when introducing signature and relations, no **extends** clause is affected, thus $\mathit{satisfyImpInvs}(OM, i') = \mathit{satisfyImpInvs}(OM', i')$]
$\{i' : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i') \land \ \mathit{satisfyExpInvs}(OM', i')\}$
$=$ [from definition of $\mathit{satisfyExpInvs}(OM', i')$]
$\{i' : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i') \land \forall f : \mathit{factInvs}(OM') \bullet$
$\mathit{satisfyFormula}(f, i')\}$
$=$ [from definitions of OM, OM',$\mathit{factInvs}(OM') = \mathit{factInvs}(OM) \cup (r = x.y)$]
$\{i' : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i') \land$
$\forall f : \mathit{factInvs}(OM) \cup \{r = x.y\} \bullet \mathit{satisfyFormula}(f, i')\}$
$=$ [set theory]
$\{i' : \mathit{Interpretation} \mid \mathit{satisfyImpInvs}(OM, i') \land$
$\forall f : \mathit{factInvs}(OM) \bullet \mathit{satisfyFormula}(f, i') \land$
$\mathit{satisfyFormula}(\{r = x.y\}, i')\}$
$=$ [from definition of $\mathit{semantics}(OM)$]
$\{i' : \mathit{Interpretation} \mid i' \in \mathit{semantics}(OM) \ \land \ \mathit{satisfyFormula}(\{r = x.y\}, i')\}$
$=$ [From Lemma 7.5, where $v = \mathbb{P}\, \mathit{objValue}$]
$\{i' \oplus i'' \mid i' \in \mathit{semantics}(OM) \ \land \ \mathit{satisfyFormula}(\{r = x.y\}, i' \oplus i'')$
$i''.\mathit{mapSig}(U) = \mathbb{P}\, \mathit{objValue}\}$
$=$ [From Lemma 7.5, where $v = \mathbb{P}(i'.\mathit{mapSig}(S) \leftrightarrow i''.\mathit{mapSig}(U))$]
$\{i' \oplus i'' \mid i' \in \mathit{semantics}(OM) \ \land \ \mathit{satisfyFormula}(\{r = x.y\}, i' \oplus i'')$
$i''.\mathit{mapSig}(U) = \mathbb{P}\, \mathit{objValue}$
$i''.\mathit{mapRel}(x) = \mathbb{P}(i'.\mathit{mapSig}(S) \leftrightarrow i''.\mathit{mapSig}(U))\}$
$=$ [From Lemma 7.5, where $v = \mathbb{P}(i''.\mathit{mapSig}(U) \leftrightarrow i'.\mathit{mapSig}(T))$]
$\{i' \oplus i'' \mid i' \in \mathit{semantics}(OM) \ \land \ \mathit{satisfyFormula}(\{r = x.y\}, i' \oplus i'')$
$i''.\mathit{mapSig}(U) = \mathbb{P}\, \mathit{objValue}$
$i''.\mathit{mapRel}(x) = \mathbb{P}(i'.\mathit{mapSig}(S) \leftrightarrow i''.\mathit{mapSig}(U))$
$i''.\mathit{mapRel}(y) = \mathbb{P}(i''.\mathit{mapSig}(U) \leftrightarrow i'.\mathit{mapSig}(T))\}$
$=$ [from $\mathit{satisfyFormula}(\{r = x.y\}, i' \oplus i'')$, $i''.\mathit{mapRel}(x) \fatsemi i''.\mathit{mapRel}(y) = i''.\mathit{mapRel}(r)$]
$\{i' \oplus \ i'' \mid i' \in \mathit{semantics}(OM) \land$
$i'' : \mathit{Interpretation} \mid i''.\mathit{mapSig}(U) = \mathbb{P}\, \mathit{objValue} \land$
$i''.\mathit{mapRel}(x) = \mathbb{P}(i'.\mathit{mapSig}(S) \leftrightarrow i''.\mathit{mapSig}(U)) \land$
$i''.\mathit{mapRel}(y) = \mathbb{P}(i''.\mathit{mapSig}(U) \leftrightarrow i'.\mathit{mapSig}(T)) \land$
$i''.\mathit{mapRel}(x) \fatsemi i''.\mathit{mapRel}(y) = i'.\mathit{mapRel}(r)\}$

Lemma E.12. For P and P':
$\mathit{heaps}(P', \mathit{filter}) =$
$\{h \oplus \ h' \mid h \in \mathit{heaps}(P, \mathit{filter}) \land$
$h' : \mathit{Heap} \mid h'.\mathit{mapClass}(U) = \mathbb{P}(\mathit{objValue}) \land$
$h'.\mathit{mapField}(x) = h.\mathit{mapClass}(S) \rightarrowtail\!\!\!\!\twoheadrightarrow h'.\mathit{mapClass}(U)) \land$
$h'.\mathit{mapField}(y) = \mathbb{P}(h'.\mathit{mapClass}(U) \leftrightarrow h.\mathit{mapClass}(T)) \land$
$\nexists o : h'.\mathit{mapClass}(U) \ \bullet \ o \in \mathit{values}(h) \land$
$h'.\mathit{mapField}(x) \fatsemi h'.\mathit{mapField}(y) = h.\mathit{mapField}(r)$

Proof. The changed commands do not add or remove heaps. Also, every `X` object is mapped to a single new `U` object by field `x`, forming a bijection; the invariant $h'.mapField(x) \mathbin{⨾} h'.mapField(y) = h.mapField(r)$ is given by modifications on **self**.`x`.`y`.

Main proof. Lemma 7.4 is then proved, using the result of the auxiliary lemmas and premise.

$semanticConformance(OM', P')$
$= $ [definition]
$\quad \forall h : heaps(P', filter) \bullet$
$\quad\quad \exists\ i : semantics(OM') \bullet$
$\quad\quad\quad \forall s : sigs(OM') \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad \forall r : rels(OM') \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$= $ [from definitions of OM, OM',$sigs(OM') = sigs(OM) \cup \{\textbf{sig}\ U\}$ and $rels(OM') = rels(OM) \cup \{x : \textbf{set}\ U\} \cup \{y : \textbf{set}\ T\}$]
$\quad \forall h : heaps(P', filter) \bullet$
$\quad\quad \exists\ i : semantics(OM') \bullet$
$\quad\quad\quad \forall s : sigs(OM) \cup \{\textbf{sig}\ U\} \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad \forall r : rels(OM) \cup \{x : \textbf{set}\ U\} \cup \{y : \textbf{set}\ T\} \bullet\ i.mapRel(r.name) = h.mapField(r.name)$
$= $ [set theory]
$\quad \forall h : heaps(P', filter) \bullet$
$\quad\quad \exists\ i : semantics(OM') \bullet$
$\quad\quad\quad \forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad i.mapSig((\textbf{sig}\ U).name) = h.mapClass((\textbf{sig}\ U).name) \wedge$
$\quad\quad\quad \forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name) \wedge$
$\quad\quad\quad i.mapRel((x : \textbf{set}\ U).name) = h.mapClass((x : \textbf{set}\ U).name) \wedge$
$\quad\quad\quad i.mapRel((y : \textbf{set}\ T).name) = h.mapClass((y : \textbf{set}\ T).name)$
$= $ [Lemma E.11 replaces $semantics(OM')$]
$\quad \forall h : heaps(P', filter) \bullet$
$\quad\quad \exists\ i : \{i' \oplus\ i'' \mid i' \in semantics(OM) \wedge$
$\quad\quad\quad i'' : Interpretation \mid i''.mapSig(U) = \mathbb{P}\, objValue \wedge$
$\quad\quad\quad\quad i''.mapRel(x) = \mathbb{P}(i'.mapSig(S) \leftrightarrow i''.mapSig(U)) \wedge$
$\quad\quad\quad\quad i''.mapRel(y) = \mathbb{P}(i''.mapSig(U) \leftrightarrow i'.mapSig(T)) \wedge$
$\quad\quad\quad\quad i''.mapRel(x) \mathbin{⨾} i''.mapRel(y) = i'.mapRel(r)\} \bullet$
$\quad\quad\quad \forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h.mapClass(s.name) \wedge$
$\quad\quad\quad i.mapSig((\textbf{sig}\ U).name) = h.mapClass((\textbf{sig}\ U).name) \wedge$
$\quad\quad\quad \forall r : rels(OM) \bullet\ i.mapRel(r.name) = h.mapField(r.name) \wedge$
$\quad\quad\quad i.mapRel((x : \textbf{set}\ U).name) = h.mapClass((x : \textbf{set}\ U).name) \wedge$
$\quad\quad\quad i.mapRel((y : \textbf{set}\ T).name) = h.mapClass((y : \textbf{set}\ T).name)$
$= $ [Lemma E.12 replaces $heaps(P', filter)$, followed by predicate calculus taking an arbitrary $h1$]
$\quad \exists\ i : \{i' \oplus\ i'' \mid i \in semantics(OM) \wedge$
$\quad\quad\quad i'' : Interpretation \mid i''.mapSig(U) = \mathbb{P}\, objValue \wedge$
$\quad\quad\quad\quad i''.mapRel(x) = \mathbb{P}(i'.mapSig(S) \leftrightarrow i''.mapSig(U)) \wedge$
$\quad\quad\quad\quad i''.mapRel(y) = \mathbb{P}(i''.mapSig(U) \leftrightarrow i'.mapSig(T)) \wedge$
$\quad\quad\quad\quad i''.mapRel(x) \mathbin{⨾} i''.mapRel(y) = i'.mapRel(r)\} \bullet$
$\quad\quad \forall s : sigs(OM) \bullet\ i.mapSig(s.name) = h1.mapClass(s.name) \wedge$

$\quad i.\mathit{mapSig}((\mathbf{sig}\ U).\mathit{name}) = h1.\mathit{mapClass}((\mathbf{sig}\ U).\mathit{name}) \land$
$\quad \forall r : \mathit{rels}(OM) \bullet\ i.\mathit{mapRel}(r.\mathit{name}) = \mathit{mapField}(h1)(r.\mathit{name}) \land$
$\quad i.\mathit{mapRel}((x : \mathbf{set}\ U).\mathit{name}) = h1.\mathit{mapClass}((x : \mathbf{set}\ U).\mathit{name}) \land$
$\quad i.\mathit{mapRel}((y : \mathbf{set}\ T).\mathit{name}) = h1.\mathit{mapClass}((y : \mathbf{set}\ T).\mathit{name})$

= [from predicate calculus, choosing $i1$ to be an interpretation for OM' with the same mappings as $h1$, taking `x` as a bijection exactly as in the program, which is allowed from its definition $(\mathbb{P}(i.\mathit{mapSig}(S) \leftrightarrow i'.\mathit{mapSig}(U)))$]

$\quad \forall s : \mathit{sigs}(OM) \bullet\ i1.\mathit{mapSig}(s.\mathit{name}) = h1.\mathit{mapClass}(s.\mathit{name}) \land$
$\quad i1.\mathit{mapSig}((\mathbf{sig}\ U).\mathit{name}) = h1.\mathit{mapClass}((\mathbf{sig}\ U).\mathit{name}) \land$
$\quad \forall r : \mathit{rels}(OM) \bullet\ i1.\mathit{mapRel}(r.\mathit{name}) = \mathit{mapField}(h1)(r.\mathit{name}) \land$
$\quad i1.\mathit{mapRel}((x : \mathbf{set}\ U).\mathit{name}) = h1.\mathit{mapClass}((x : \mathbf{set}\ U).\mathit{name}) \land$
$\quad i1.\mathit{mapRel}((y : \mathbf{set}\ T).\mathit{name}) = h1.\mathit{mapClass}((y : \mathbf{set}\ T).\mathit{name})$

= [from $\mathit{premise}(OM,\ OM',\ P)$, mappings are maintained for names in OM]

$\quad i1.\mathit{mapSig}((\mathbf{sig}\ U).\mathit{name}) = h1.\mathit{mapClass}((\mathbf{sig}\ U).\mathit{name}) \land$
$\quad i1.\mathit{mapRel}((x : \mathbf{set}\ U).\mathit{name}) = h1.\mathit{mapClass}((x : \mathbf{set}\ U).\mathit{name}) \land$
$\quad i1.\mathit{mapRel}((y : \mathbf{set}\ T).\mathit{name}) = h1.\mathit{mapClass}((y : \mathbf{set}\ T).\mathit{name})$

= [from definitions of $h1, i1$, values for signature `U` and relation `y` are equivalent]

$\quad i1.\mathit{mapRel}((x : \mathbf{set}\ U).\mathit{name}) = h1.\mathit{mapClass}((x : \mathbf{set}\ U).\mathit{name})$

= [from definition of $i1$, which match the bijection from $h1$]

$\quad \mathit{true}$

□

Bibliography

[1] Action Semantics Consortium. Precise Action Semantics for the Unified Modelling Language, 2000. http://www.kabira.com/as/.

[2] Scott Ambler. *Agile Modeling*. Wiley, 2002.

[3] AndroMDA.org. Andromda, 2007. http://www.andromda.org/.

[4] Robert S. Arnold. Software Restructuring. *Proceedings of the IEEE*, 77(4):607–617, April 1989.

[5] L. J. Arthur. *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, 1988.

[6] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo, Finland, 1987. Ser. A No. 55.

[7] Marc J. Balcer and Stephen J. Mellor. *Executable UML: A Foundation for Model Driven Architecture*. 2002.

[8] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005.

[9] F. Bannwart and P. Müller. Changing programs correctly: Refactoring with specifications. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 492–507, 2006.

[10] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[11] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[12] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[13] L.A. Belady and M.M. Lehman. A model of large program development. *IBM System Journal*, 15(3):225–252, 1976.

[14] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *PODS*, 2006.

[15] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[16] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Marcio Cornelio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, October 2004.

[17] Paulo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Coordinated Distributed Diagram Transformation for Software Evolution. *Electronic Notes in Theoretical Computer Science*, 72(4), 2003.

[18] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *Software Tools for Technology Transfer*, 2005.

[19] Jordi Cabot and Ernest Teniente. Determining the structural events that may violate an integrity constraint. In *UML 2004*, volume 3273, pages 320–334, 2004.

[20] Jordi Cabot and Ernest Teniente. Constraint support in mda tools: A survey. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *LNCS*, pages 256–267, 2006.

[21] A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713 – 728, 2000.

[22] Walter Cazzola, Sonia Pini, Ahmed Ghoneim, and Gunter Saake. Co-evolving application code and design models by exploiting meta-data. In *SAC*, pages 1275–1279. ACM Press, 2007.

[23] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2001.

[24] Anthony Cleve and Jean-Luc Hainaut. Co-transformations in database applications evolution. In *GTTSE*, pages 409–421, 2006.

[25] Compuware. Compuware optimalj, 2007. http://www.compuware.com/-products/optimalJ.

[26] Marcio Lopes Cornélio. *Refactorings as Formal Refinements*. PhD thesis, Federal University of Pernambuco, 2004.

[27] Michelle L. Crane and Juergen Dingel. Runtime Conformance Checking of Objects Using Alloy. In *3rd Workshop on Runtime Verification*, pages 62–73, 2003.

[28] Alcino Cunha, José Nuno Oliveira, and Joost Visser. Type-safe two-level data transformation. In *14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 284–299, 2006.

[29] Jim Davies, Charles Crichton, Edward Crichton, David Neilson, and Ib Holm. Formality, Evolution and Model-Driven Software Engineering. In *SBMF*, pages 32–47, November 2004.

[30] Eclipse.org. Eclipse Project, 2007. http://www.eclipse.org.

[31] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[32] Escher Technologies. Perfect Developer, 2008. http://www.eschertech.com/-products/index.php.

[33] Esterel Technologies. SCADE Suite, 2008. http://www.esterel-technologies.com/-products/scade-suite/.

[34] Andy S. Evans. Reasoning with UML Class Diagrams. In *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pages 102–113. IEEE CS Press, 1998.

[35] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* 2004.

[36] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *ACM SIGPLAN Notices*, 37(5):234–245, 2002.

[37] Martin Fowler. *Refactoring—Improving the Design of Existing Code.* Addison Wesley, 1999.

[38] Robert France and James M. Bieman. Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software. In *Proceedings of The IEEE International Conference on Software Maintenance*, November 2001.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[40] Gentleware. Poseidon for UML, 2005. http://www.gentleware.com/.

[41] Rohit Gheyi. Basic Laws of Object Modeling. Master's thesis, Federal University of Pernambuco, February 2004.

[42] Rohit Gheyi. *A Refinement Theory for Alloy.* PhD thesis, Informatics Center – Federal University of Pernambuco, August 2007.

[43] Rohit Gheyi, Tiago Massoni, and Paulo Borba. An abstract equivalence notion for object models. *ENTCS*, 130:3–21, 2005.

[44] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A rigorous approach for proving model refactorings. In *20th IEEE/ACM ASE*, pages 372–375, Long Beach, United States, nov 2005.

[45] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A static semantics for alloy and its impact in refactorings. *ENTCS*, 184:209–233, jun 2007.

[46] Martin Gogolla and Mark Richters. Equivalence Rules for UML Class Diagrams. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 87–96, 1998.

[47] Pieter Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent uml refactorings. In *UML 2003*, volume 2863 of *LNCS*, pages 144–158, 2003.

[48] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, third edition, June 2005.

[49] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[50] Aaron Greenhouse, T.J. Halloran, and William L. Scherlis. Observations on the Assured Evolution of Concurrent Java Programs. In *Workshop on Concurrency and Synchronization in Java Programs*, 2004. Technical Report 2004-01.

[51] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, University of Washington, 1991.

[52] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In *5th Refinement Workshop, Workshops in Computing*, pages 272–297, 1992.

[53] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering Binary Class Relationships: Putting Icing on the UML Cake. In *Proceedings of the 19th OOPSLA*, pages 301–314. ACM Press, October 2004.

[54] Brent Hailpern and Peri Tarr. Model-driven development: the good, the bad, and the ugly. *IBM System Journal*, 45(3):451–461, 2006.

[55] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping UML Designs to Java. In *Proceedings of OOPSLA 2000*, pages 178–187. ACM Press, 2000.

[56] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of Programming. *Communications of the ACM*, 30(8):672–686, 1987.

[57] Walter Hürsch. *Maintaining Consistency and Behavior of Object-Oriented Systems during Evolution.* PhD thesis, Northeastern University, 1995.

[58] D. Jackson. *Software Abstractions: Logic, Language and Analysis.* MIT Press, 2006.

[59] Daniel Jackson, Ian Schechter, and Ilya Shlyahter. Alcoa: the Alloy Constraint Analyzer. In *Proceedings of the 22nd ICSE*, pages 730–733. ACM Press, 2000.

[60] Daniel Jackson and Mandana Vaziri. Finding Bugs with a Constraint Solver. In *ISSTA 2000*, pages 14–25. ACM Press, 2000.

[61] Ivar Jacobson, James Rumbaugh, and Grady Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.

[62] Jet Brains, Inc. IntelliJ Idea, 2004. http://www.intellij.com/idea/.

[63] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.

[64] Stuart Kent. Model Driven Engineering. In *Integrated Formal Methods 2002*, LNCS 2335, pages 286–298, 2002.

[65] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An Analyzable Annotation Language. In *Proceedings of the 17th OOPSLA*, pages 231–245. ACM Press, 2002.

[66] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: the Practice and Promise of The Model Driven Architecture*. Addison Wesley, 2003.

[67] Ralph Lammel. Coupled software transformations. In *SET*, pages 31–35, 2004.

[68] Kevin Lano and Juan Bicarregui. Semantics and Transformations for UML Models. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France*, volume 1618 of *LNCS*, 1999.

[69] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *4th IWPSE*, pages 70–74, 2002.

[70] B. Liskov and J. Guttag. *Program Development in Java*. Addison Wesley, 2001.

[71] Andrew Martin. *Machine-Assisted Theorem-Proving for Software Engineering*. PhD thesis, Penbroke College, 1994.

[72] Tiago Massoni, Rohit Gheyi, and Paulo Borba. A UML Class Diagram Analyzer. In *3rd International Workshop on Critical Systems Development with UML, affiliated with 7th UML Conference*, pages 143–153, 2004.

[73] Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal Refactoring for UML Class Diagrams. In *19th SBES*, pages 152–167, Uberlandia, Brazil, 2005.

[74] Tiago Massoni, Rohit Gheyi, and Paulo Borba. An approach to invariant-based program refactoring. In *Software Evolution through Transformations: Embracing the Change*, pages 91–101, Natal, Brazil, 2006.

[75] Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal model-driven program refactoring. In *FASE, ETAPS*, 2008. To appear.

[76] Kim Mens, Andy Kellens, Frdric Pluquet, and Roel Wuytsc. Co-evolving code and design with intensional views: a case study. *Computer Languages, Systems & Structures*, 32(2 – 3):140–156, July 2006.

[77] Carrol Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1998.

[78] Shamkant B. Navathe and Ramez A. Elmasri. *Fundamentals of Database Systems.* Addison-Wesley, 2003.

[79] U.A. Nickel, J. Niere, J.P. Wadsack, and A. Zündorf. Roundtrip engineering with fujaba. In J. Ebert, B. Kullbach, and F. Lehner, editors, *Proc of* 2^{nd} *Workshop on Software-Reengineering (WSR), Bad Honnef, Germany.* Fachberichte Informatik, Universität Koblenz-Landau, August 2000.

[80] Interactive Objects. Arcstyler, 2007. http://www.arcstyler.com/.

[81] OMG. Uml 2.0 superstructure specification. Technical report, Object Management Group, 2004. Document ptc/04-10-02.

[82] William Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[83] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th CADE*, volume 607 of *LNAI*, pages 748–752, USA, jun 1992.

[84] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 16(3):11, 2007.

[85] Patrick Lam and Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2003)*, Darmstadt, Germany, July 2003.

[86] S. L. Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall, 2001.

[87] Martin Rinard and Viktor Kuncak. Object Models, Heaps, and Interpretations. Technical Report 81, January 2001. MIT Laboratory of Computer Science.

[88] Don Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[89] Augusto Sampaio. *An Algebraic Approach to Compiler Design.* World Scientific, 1997.

[90] S. Sendall and J. Küster. Taming Model Round-Trip Engineering. In *Workshop on Best Practices for Model-Driven Software Development at OOPSLA*, 2004.

[91] Rational Software. Rational software architect, 2007. http://www-306.ibm.com/-software/awdtools/architect/swarchitect/.

[92] Mirko Streckenbach and Gregor Snelting. Refactoring Class Hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330. ACM Press, 2004.

[93] Sun Microsystems. Java 2 Enterprise Edition Specification, 2003.

[94] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML Models. In *UML 2001*, volume 2185 of *LNCS*, pages 134–148, 2001.

[95] Mana Taghdiri. Inferring Specifications to Detect Errors in Code. In *19th International Conference on Automated Software Engineering (ASE 2004)*, pages 144–153. IEEE Computer Society, 2004.

[96] Dave Thomas and Brian M. Barry. Model driven development: the case for domain oriented programming. In *OOPSLA '03: Companion*, pages 2–7, New York, NY, USA, 2003. ACM.

[97] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for Generalization Using Type Constraints. In *18th OOPSLA*, pages 13–26. ACM Press, 2003.

[98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, 2003.

[99] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

[100] Yoonsik Cheon and Gary Leavens. A Runtime Assertion Checker for the Java Modeling Language. In *Software Engineering Research and Practice (SERP'02)*, pages 322–328. CSREA Press.

[101] Xiaofang Zhang, Michal Young, and John H. E. F. Lasseter. Refining Code-Design Mapping with Flow Analysis. In *12th ACM SigSoft FSE*, pages 231–240. ACM Press, 2004.