Universidade do Minho
Escola de Engenharia

Nuno Filipe Moreira Macedo

A Relational Approach to Bidirectional Transformation

The MAP Doctoral Program in Computer Science of the Universities of Minho, Aveiro and Porto

A thesis submitted at the University of Minho for the degree of Doctor of Philosophy in Informatics (PhD)

under the supervision of
Professor Doutor Manuel Alcino Pereira da Cunha

November, 2014

Acknowledgments

When I enrolled college in an informatics engineering degree, I was convince by "society" that it was not feasible to be an researcher in Portugal, and that I should just get a real job. Almost a decade later, it is probably no surprise to find myself with this PhD dissertation in computer science, still hoping for "society" to be wrong.

My supervisor Alcino Cunha is one of those responsible for me taking that turn, whose approach to computer science captivated me during my undergraduate years. I am very thankful for his support and dedication, as well as for all the hours spent with me so that this dissertation could materialize.

If someone else is responsible for my initiation into research is José Nuno Oliveira with his contagious enthusiasm about the science in computer science, whom I thank for the valuable advice and discussions.

I would like to thank Hugo Pacheco, who acted as an unofficial co-supervisor during my first years of research and helped me kick-start what eventually became this dissertation. I am also grateful to all my other colleagues from 2.07 and surrounding labs for all those moments spent not working.

Finally, I thank my family and friends for all the support during this long journey. I hope that they are all aware of how important they are to me and that no further words are necessary.

I thank FCT (Portuguese Foundation for Science and Technology) for supporting the development of this thesis (grant SFRH/BD/69585/2010) through the *Programa Operacional Potencial Humano* (POPH), of the QREN framework, within the European Social Fund. This work was also supported by the FATBIT project (FCOMP-01-0124-FEDER-020532), funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by national funds through the FCT.

A Relational Approach to Bidirectional Transformation

The ubiquity of data transformation problems in software engineering has led to the development of *bidirectional transformation* techniques in a variety of application domains. Model-driven engineering (MDE) is one of those areas, where such techniques are essential to maintain the consistency between multiple coexisting and simultaneously evolving models.

However, the lack of in-depth research about certain characteristics of MDE has hindered the development of effective bidirectional model transformations that are able to address realistic MDE scenarios. This dissertation tackles two of these issues: that of constrained transformation domains and least-change transformations. The first regards the transformations' ability to take into consideration the constraints imposed by the meta-models, and is essential to achieve correctness; the second regards the transformations' ability to control the selection of updates from among those considered correct, and is essential to achieve a predictable system.

These two issues are addressed under two popular bidirectional transformation schemes: in the context of the asymmetric framework of *lenses*, following a combinatorial approach; and in the context of the symmetric framework of *constraint maintainers*, proposing a solution based on model finding. The latter was effectively deployed as Echo, a tool for model repair and transformation. The expressiveness and flexibility provided by *relational logic* enabled it to be used as the unifying formalism throughout this dissertation.

Uma Abordagem Relacional à Transformação Bidirecional

A ubiquidade de problemas de transformação de dados em engenharia de *software* levou ao desenvolvimento de técnicas para *transformação bidirecional* numa variedade de domínios de aplicação. A Engenharia Baseada em Modelos (MDE) é uma dessas áreas, onde essas técnicas são essenciais para gerir a consistência entre múltiplos modelos que coexistem e evolvem simultaneamente.

No entanto, a falta de estudos aprofundados sobre algumas características da MDE tem dificultado o desenvolvimento de técnicas de transformação bidirecional de modelos eficazes e que consigam lidar com cenários MDE realísticos. Esta dissertação aborda dois destes problemas: o de domínios de transformação restringidos e o de transformações com mudanças-mínimas. O primeiro tem que ver com a capacidade das transformações de ter em consideração as restrições impostas pelos meta-modelos e é essencial para atingir correcção; a segunda tem que ver com a capacidade de controlar a seleção de modificações entre as consideradas corretas, e é essencial para obter um sistema previsível.

Esta tese aborda estes dois problemas sob dois populares esquemas de transformação bidirecional: no contexto da *framework* assimétrica das *lentes*, seguindo uma abordagem combinatorial, e no contexto da *framework* simétrica dos *constraint maintainers*, sendo proposta uma solução baseada em "procura de modelos". Esta última foi efetivamente implementada como Echo, uma ferramenta para a reparação e transformação de modelos. A expressividade e flexibilidade proporcionada pela *lógica relacional* permitiu que esta fosse usada como o formalismo unificador desta dissertação.

Contents

List of Figures

List of Tables

Acronyms

ATL ATLAS Transformations Language.

EMF Eclipse Modeling Framework.

GED Graph-edit Distance.

MDA Model-driven Architecture.

MDE Model-driven Engineering.

MF Model Finding.

OBD Operation-based Distance.

OCL Object Constraint Language.

QVT Query/View/Transformation.

QVT-R QVT Relations.

TO-MF Target-oriented Model Finding.

TRC Typed Relational Constraint.

Chapter 1

Introduction

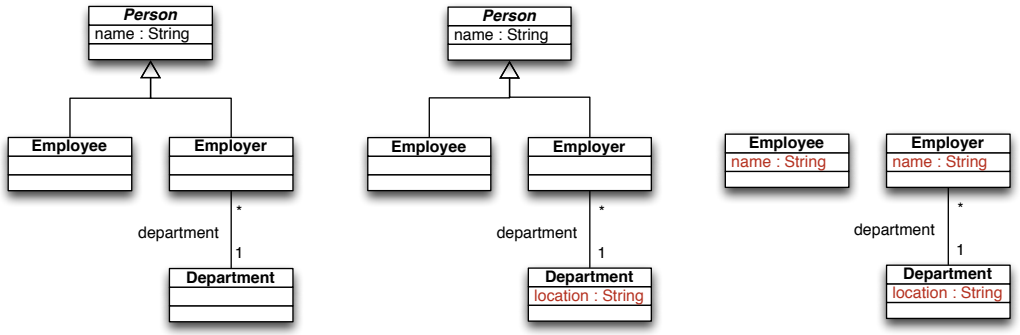
Transforming data between different formats is an essential task in computer science and software engineering. Ordinary as this exercise may seem, creating a target artifact b from a source artifact a is often just the first step in a dynamic evolution process: the initial transformation implicitly binds a and b, and as either artifact gets updated, modifications must be propagated to the other side in order to keep the overall system *consistent*. For decades, this kind of problem has been addressed via *ad hoc* or domain-specific techniques in virtually every area of computer science—the view-update problem from the database community being the classic example. However, in the last few years, research on *bidirectional transformation* (Czarnecki et al., 2009) has exploded, with developments in areas like heterogeneous data synchronization (Brabrand et al., 2005; Kawanaka and Hosoya, 2006; Foster et al., 2007), string manipulation (Bohannon et al., 2008; Barbosa et al., 2010), functional languages (Matsuda et al., 2007; Voigtländer, 2009; Pacheco and Cunha, 2010), model transformation (Ehrig et al., 2007; Cicchetti et al., 2010; Macedo and Cunha, 2013), user interfaces (Meertens, 1998; Hu et al., 2008), relational databases (Bancilhon and Spyratos, 1981; Dayal and Bernstein, 1982; Bohannon et al., 2006), graph transformation (Schürr, 1994; Hidaka et al., 2010), or spreadsheet systems (Cunha et al., 2012; Macedo et al., 2014c). The main idea behind bidirectional transformation frameworks is having a single transformation artifact denote the transformations in both directions (either by construction or through calculation), avoiding the cumbersome and error-prone task of manually writing and maintaining two coherent transformations.

1.1 Bidirectional Model Transformation

A particular area where bidirectional transformation plays an essential role is *model-driven engineering* (MDE), a family of development processes that focus on models as the main development artifact. The higher level of abstraction provided by models promotes maintenance, modularization, reusability and communication between different stakeholders, and thus MDE has been increasingly adopted by software engineering practitioners (Schmidt, 2006). At the core of MDE are *model transformations* (Gerber et al., 2002; Sendall and Kozaczynski, 2003), that enable the system to evolve both horizontally—by presenting different perspectives or different components of the system—and vertically—by navigating across different levels of detail through abstraction and refinement operations (Mens and Gorp, 2006). These (unidirectional) model transformations essentially allow users to generate fresh *target* models from *source* models.

A classic example of such transformations is the object-relational mapping (`cd2dbs`) problem, concerned with persisting object-oriented data in relational databases. To be effective, the class diagram denoting the structure of the data objects and the relational database schema denoting the structure of the database—that provide two vertically different views of the same system—must be consistent with each other. Consider as an example class diagram c_0 (Figure 1.1a) that represents a simple company model comprised of employees and employers, which are both persons with names, with each employer being assigned to a department. This class diagram could be persisted in a database following schema s_0 (Figure 1.2a), which could be generated from c_0 by a transformation $\overrightarrow{\texttt{cd2dbs}}$ that maps each persistent class (denoted by non-italic title) to a table, with a column for each of its attributes and associations, including those inherited from super-classes.

MDE gives rise to a highly dynamic development environment, where different models are expected to be individually modified by different stakeholders, and inconsistencies will undoubtedly be introduced: a developer could update schema s_0 to s_1 (Figure 1.2b) by assigning locations to departments, breaking the consistency with the original class diagram c_0. Thus, coexisting models introduce the problem of *inter-model consistency* management—in contrast to *intra-model consistency* that manages the conformity of a model with its meta-model (Huzar et al., 2004). Although these inconsistencies should be tolerated to some extent (Balzer, 1991), they must eventually be repaired, and thus a transformation $\overleftarrow{\texttt{cd2dbs}}$, that converts database schemas back



(a) Model c_0. (b) Model c_1. (c) Model c_2.

Figure 1.1: Example class diagrams.

into class diagrams, is required. Maintaining these two transformations ($\overrightarrow{\texttt{cd2dbs}}$ and $\overleftarrow{\texttt{cd2dbs}}$) as a coherent pair is clearly a cumbersome and error-prone task, as they must be manually defined and checked for correctness. The goal of *bidirectional model transformation* is precisely to mitigate this problem. As defined by Stevens (2007)

> "A bidirectional model transformation is some way of specifying algorithmically how *consistency* should be restored, which will [...] be able to modify either of the two models."

The burden is then to derive the pair of consistency-restoring transformations from a single specification artifact. To tame this task, in an early position paper, Sendall and Küster (2004) divide inter-model consistency management in four steps: *a*) define in which circumstances the models are consistent; *b*) detect that models are inconsistent; *c*) devise a plan to restore consistency that embodies the intention of the user; *d*) apply the plan and restore consistency.

In the general case, model transformations are not bijective, in general typically each model may be consistent with a (possibly empty) set of opposite models. One of the most famous bidirectional transformation frameworks is that of *lenses*, proposed by Foster et al. (2007) to tackle asymmetric transformation scenarios. Inspired by the classic view-update problem from the database community (how can updates on a database view be propagated back to the source table?), lenses are suitable for scenarios where one of the transformation domains (the *view*) contains less information than the other (the *source*). In this context, the transformation from the source to the view is usually easier to specify and implicitly defines the relationship between the models.

```
CREATE TABLE Employee (
  _Id  int,
  name varchar (255),
  PRIMARY KEY (_Id)
);
CREATE TABLE Employer (
  _Id         int,
  name        varchar (255),
  department int,
  PRIMARY KEY (_Id),
  FOREIGN KEY (department)
    REFERENCES Department (_Id)
);
CREATE TABLE Department (
  _Id int
);
```

(a) Model s_0.

```
CREATE TABLE Employee (
  _Id  int,
  name varchar (255),
  PRIMARY KEY (_Id)
);
CREATE TABLE Employer (
  _Id         int,
  name        varchar (255),
  department int,
  PRIMARY KEY (_Id),
  FOREIGN KEY (department)
    REFERENCES Department (_Id)
);
CREATE TABLE Department (
  _Id       int,
  location varchar (255)
);
```

(b) Model s_1.

Figure 1.2: Example database schemas.

This forward transformation is embedded with backward semantics, which should be able to retrieve information discarded by the forward transformation when creating the view from the original source. Such is the case of our example, where the forward transformation $\overrightarrow{\texttt{cd2dbs}}$ is straight-forward to define and discards information from the class diagram when creating the database schema (the classes' hierarchy tree).

Classic lens frameworks are *combinatorial*, providing a bidirectional transformation language comprised of individual primitives and combinators that are given bidirectional semantics at design time. This allows the user to write complex correct-by-construction bidirectional transformations, while implicitly defining both the forward and backward transformations.

Model transformation, however, introduces an additional complexity layer on the bidirectional transformation problem. Meta-models typically entail additional rich constraints over conforming models: a model transformation is said to be *correct* if, besides restoring inter-model consistency, produces well-formed models. Consider the case of class diagrams, where class inheritance must not be cyclic. Depending on the definition of `cd2dbs`, the occurrence of cyclic inheritance may not affect inter-model consistency management, but will without a doubt affect the overall consistency of the environment. Thus, bidirectional model transformations that do not take into consideration constraints inherent to the transformation domains are unpredictable and ultimately not

useful. Although lenses are one of the most widely spread bidirectional transformation frameworks, having been extended and applied to a variety of data domains, most are designed to handle tree-like data structures without native support for node references, which hinders the representation of model artifacts. A lens framework to transform graph data structures has been proposed by Hidaka et al. (2010), but although models can naturally be represented as graphs, the lack of support for additional constraints over the transformation domains renders the framework inadequate in the MDE setting.

Bidirectional transformations are expected to follow certain round-tripping laws that render them well-behaved. In classic lens frameworks, the pair of transformations is expected to be *acceptable*—the updated view data is recoverable from the updated source data—and *stable*—consistent models should remain unmodified. However, these round-tripping laws cover only boundary scenarios, allowing the definition of many unreasonable transformations in-between. Consider propagating the update that originated schema s_1 back to class diagram c_0. According to the lenses' laws, any $\overleftarrow{\texttt{cd2dbs}}$ that produces a class diagram from which $\overrightarrow{\texttt{cd2dbs}}$ recovers schema s_1 will be acceptable. Model c_2 (Figure 1.1b) is one such candidate, even though the existing tree hierarchy was destroyed (the notion of person was removed and the name attribute pushed down the tree to employees and employers). However, the user is probably not expecting the generation of a fresh new class diagram every time the schema is updated, but to have information from the existing class diagram not present in the schema preserved. Since the backward transformation is able to retrieve information from the original source model c_0, model c_1 (Figure 1.1c) would clearly be a better choice, as it preserves more information from c_0—its hierarchy tree. This touches one of the main challenges in model transformation, as Sendall and Küster (2004) already emphasize: to embody "the intent and expectation of the user".

Meertens (1998, p. 3–4), one of the first to formalize and study bidirectional transformation, was already aware of this issue, and went a step further to specify an additional design principle:

> "The claim is that although there are many formal solutions to the constraint, some are *more intuitively natural* in terms of user expectation than others. [...] Generalizing the example leads to the *Principle of Least Change*: *The action taken by the maintainer of a constraint after a violation should change no more than is needed to restore the constraint.*"

While seemingly simple, correctly capturing, formalizing and enforcing this policy is a

pretty challenging task and is typically disregarded by existing techniques, that only informally and loosely approximate this intuition using *ad hoc* or heuristic mechanisms. At the core of the issue is the inevitability that no general-purpose technique will be able to capture the notion of "closest" solution in each particular context without input from the user (Pacheco et al., 2014).

Addressing these two issues—support for domain constraints and least-change policies—is particularly challenging in lens frameworks due to its combinatorial nature, where combinators are expected to put together opaque lenses while preserving bidirectional properties. Since combinators are not aware of global constraints or succeeding transformations, they are not able to adapt its behavior accordingly. In fact, even though the round-tripping laws are loose, lens combinators are typically assigned particular computations at design time, disregarding the consequences of such choices on the global produced output.

Transformation-induced constraints, however, fail to describe certain classes of inter-model consistency maintenance problems, either because the transformation domains are symmetric—both containing information not present in the other—or due to the very nature of the problem—like the case of coexisting class diagrams and hierarchical state machines, where the latter are typically not generated by model transformation. Rather than a transformation, inter-model constraints in these scenarios are better specified by a *consistency relation* that states when two models are consistent. This class of transformations is usually characterized under the *constraint maintainer* framework initially proposed by Meertens (1998) to tackle bidirectionality in the context of user interfaces. Here, a bidirectional transformation is specified by a non-executable binary relation that allows consistency testing, associated with two executable consistency-restoring transformations that must be derived from it.

Likewise to lens frameworks, concrete instantiations of the constraint maintainer framework typically enforce two loose laws, already proposed by Meertens (1998): transformations should be *correct*—(inter-model) consistency is restored—and *stable*—consistent models are not updated. Notwithstanding, Meertens was already aware that these laws under-specify the behavior of the transformations, and went a step further by specifying the above-mentioned principle of least-change.

Unfortunately, Meertens' initially proposed technique is not able to scale up to the complexity inherent to model transformation and, in fact, so far no bidirectional transformation framework based on the constraint maintainer scheme has success-

fully addressed the two issues previously mentioned. A typical example regards the *Query/View/Transformation* (QVT) standard proposed by the OMG (2011a) in the context of the *Model-driven Architecture* (MDA) initiative with the intention of standardizing model transformation. One of the proposed QVT languages is *QVT Relations* (QVT-R), in which transformations are specified through consistency relations, allowing bidirectional reasoning. QVT-R transformations are expected to restore inter-model consistency (correctness) and "check-before-enforce" (OMG, 2011a, p. 15) (stability). Other than that, there are no guarantees that information from the models being updated is preserved. Moreover, intra-model constraints are disregarded, and thus updated models may potentially be ill-formed.

In terms of expressiveness, constraint maintainer frameworks subsume that of lens frameworks—the previous `cd2dbs` problem could be specified by a consistency relation that stated when a class diagram and a database schema are considered consistent. Moreover, since transformations are now derived from constraints over the model-based environment, global concepts like meta-model constraints and least-change principles seem to be more naturally encoded. However, that comes at a cost: forfeiting the combinatorial approach means that two coherent transformations must be derived from a possibly non-executable consistency relation rather than being attained for free from bidirectional primitives. In fact, constraint maintainers are not even suitable for simple compositional reasoning (Meertens, 1998, p. 42).

1.2 Goals and Contributions

The goal of this thesis is to explore mechanisms to render bidirectional transformation frameworks for non-tree-like data usable and predictable by addressing precisely the two above-mentioned issues: *correctness*—transformations produce solutions that are both inter- and intra-model consistent—and *least-change*—transformations apply minimal updates, according to specified metrics.

The main contributions can be summarized as follows:

1. the extension of the combinatorial **lens framework** to support datatype invariants over the transformation domains and the enforcement of least-change policies over backward transformations is explored;

2. the formalization of the **constraint maintainer framework** over model finding

procedures is proposed, with regard for intra- and inter-model constraints and least-change semantics;

3. the proposed constraint maintainer framework is **implemented**, providing bidirectional semantics for existing model transformation languages and focusing on the seamless integration with state-of-the-art MDE environments.

Lens framework We start by exploring the feasibility of extending the classic combinatorial lens framework with expressive datatype invariants over the transformation domains in order to support meta-model constraints. Since existing lens frameworks are mostly functional, the specific behavior of each transformation is decided at design time—even though the round-tripping laws are loose and allow a wide range of valid updates—and thus may produce outputs that break the overall domain constraints. We study how providing the backward transformations with looser specifications, enabling them to explore the whole range of valid updates and select one deemed more suitable, affects the overall behavior of this lens framework. To promote usability, classic lens frameworks require forward transformations to be surjective to assure the totality of the backward transformation. By providing more refined datatypes we are also able to support a more expressive language without lifting updatability guarantees.

Multi-valued transformations impose performance limitations, and thus we explore whether constraint-aware syntactic approaches can be developed. Two instantiations for the invariant-constrained lens framework are proposed: the first is a general-purpose approach whose invariants and backward transformations are arbitrary relational expressions and the second is a domain-specific approach in the context of spreadsheet formulas where a synthesis procedure calculates concrete backward transformations from the system's current invariants.

Next we formalize the concept of least-change lens and extensively explore how the adoption of this principle affects the behavior of the transformations, in particular under the sequential composition combinator. Again, we rely on looser specifications that, by being able to explore the whole range of valid updates, may select those considered minimal and support composition in scenarios where single-valued transformations would fail. We show that the ability to customize the metric over which least-change is enforced is essential to control the overall behavior of the system and capture the intention of the user, and propose several criteria to test whether, under given metrics, least-change is preserved by composition.

Constraint maintainer framework While the initial proposal of constraint maintainers already advocated the principle of least-change, the technique could not scale up to more realistic applications, namely those in the MDE context. We provide a general technique for the deployment of constraint maintainers for graph-like data structures based on relational model finding procedures. To attain correctness, enforcement semantics take into consideration both intra- and inter-model constraints, while enforcing least-change involves extending model finding procedures as to produce minimal updates. The impact of different model distances on the generation of solutions in the context of MDE is also explored. The underlying model finding procedure can be generalized to address other model transformation scenarios. In particular, we show that it can be trivially extended to accommodate model repair, synchronization and the multidirectional transformation scenarios.

We then present the translation of a concrete model transformation language into this formalization of the constraint maintainer framework, namely that of QVT-R transformations, whose update semantics are known to be affected by ambiguities. This approach allowed us to clarify some of these issues due to its clear and predictable semantics based on least-change, as opposed to the standard semantics and other proposed implementations. Taking advantage of the general technique, we also address the (heretofore widely disregarded) problem of QVT-R transformation over more than two transformation domains.

Finally, we show that our technique can be used to bidirectionalize unidirectional transformation languages, as long as sensible inter-model constraints can be derived from the specifications. As a proof-of-concept, we propose an embedding of the declarative subset of the widely used *ATLAS transformation language* (ATL) (Jouault et al., 2008). ATL is inherently unidirectional and this embedding provides clear bidirectional semantics with minimal effort, enabling the maintenance of the consistency between models generated by ATL transformations.

Implementation The complexity of the proposed constraint maintainer and lens frameworks amounted to that of solving procedures. While this may be expected in the constraint maintainer framework where the system has to deal with global constraints, its occurrence in lens frameworks is more serious: combinatorial approaches are expected to be built over simple and efficient primitives. Constraint maintainers are also more expressive than lenses, since they may specify bidirectional transfor-

mations in symmetric scenarios. Hence, we chose to pursue the development of the proposed constraint maintainer technique into an effective bidirectional transformation framework.

This was attained by embedding MDE artifacts into model finding problems using the Alloy specification language (Jackson, 2012), which can be analyzed through an embedding in Kodkod, resulting in a procedure that is guaranteed to be correct and complete (all valid solutions are calculated) over a bounded universe. To guarantee the former, the system translates meta-models and any additional constraints over them (e.g. annotations in the *Object Constraint Language* (OCL) proposed by the OMG (2012)), building on previous work (Cunha et al., 2013), and model transformations (concretely, QVT-R and ATL). To enforce the latter, model metrics are also inferred from the meta-model.

To be useful in practice, the tool should seamlessly integrate standard MDE processes and support standard MDE data formats. The outcome is Echo, a tool whose goal is to promote the correct and predictable bidirectional model transformations, deployed as an Eclipse[1] plug-in, one of the most widely used IDE's in MDE contexts, and built over the *Eclipse Modeling Framework* (EMF)[2]. An extensive evaluation of the behavior and performance of the developed tool is performed. Being solver-based, performance is the bottleneck of our technique, so we attempt additional optimizations by simplifying Alloy specifications, which prove to be of general application.

The unifying concept behind the development of this thesis is that of *relational logic*, a formalization of first-order logic with relational operators and extended with transitive closure operations. On the one hand, relational logic provides a higher level of abstraction than plain first-order logic, rendering it more suitable to reason about MDE problems, as the success of Kodkod/Alloy toolchain, built over relational logic, has been proving; on the other hand, partial and multi-valued transformations are natural concepts in relational formalisms, and have proved essential in the formalization of the proposed frameworks.

[1] http://www.eclipse.org/.
[2] http://www.eclipse.org/modeling/emf/.

1.3 Overview

The remainder of this manuscript is structured in three parts.

Part I introduces the essential concepts required to understand the remainder of the manuscript.

Chapter 2 introduces relational logic, the formal foundations over which the work of this thesis is developed.

Chapter 3 presents basic bidirectional transformation concepts that are relevant for the understanding and assessment of the proposed techniques.

Part II explores the development of lens frameworks over transformation domains enhanced with datatype invariants and least-change policies.

Chapter 4 introduces a framework of invariant-constrained lenses as well two instantiations that validate its feasibility. This framework was proposed in a publication by Macedo, Pacheco, and Cunha (2012) and given a concrete instantiation in Macedo, Pacheco, Cunha, and Sousa (2014c).

Chapter 5 introduces a framework of least-change lenses, analyzing their behavior under sequential composition. This framework was proposed in a publication by Macedo, Pacheco, Cunha, and Oliveira (2013b).

Part III explores the development of a correct and least-change constraint maintainer framework over graph-like data structures.

Chapter 6 proposes the deployment of constraint maintainers (and other general MDE tasks) as model finding procedures with minimality concerns.

Chapter 7 proposes the embedding into model finding problems of a concrete instantiation of the constraint maintainer framework, namely QVT-R. This embedding was proposed in publications by Macedo and Cunha (2013, 2014) and Macedo, Cunha, and Pacheco (2014b).

Chapter 8 shows how to provide bidirectional semantics to unidirectional model transformation language, using ATL as a proof-of-concept. This embedding was proposed in a publication by Macedo and Cunha (2014).

Chapter 9 presents the deployment of the constraint maintainer framework over the Alloy model finder as an Eclipse plugin. This resulted in a published tool demonstration by Macedo, Guimarães, and Cunha (2013a) whose evaluation was presented in a publication by Macedo and Cunha (2014).

Chapter 10 assesses the overall contribution of the thesis, and presents some possible future work directions.

Part I

Preliminaries

Chapter 2

Relational Logic

The notion of *relation* is essential in various scientific areas, mathematics or philosophy, since it embodies concepts that are ubiquitous in human knowledge—it is the simplest mechanism through which relationships between two or more entities can be denoted. As put by Augustus de Morgan in his seminal work on the calculus of binary relations (Morgan, 1966, p. 119):

> "When two objects, qualities, classes, or attributes, viewed together by the mind, are seen under some connexion, that connexion is called a relation."

Although discussion over the nature of relations dates back to classical Greece, the calculus of binary relations in its current form was first formalized by Morgan in 1860. His works were soon extended by Charles Sanders Peirce, and further developed by Ernst Schröder. In 1941, Alfred Tarski finally axiomatized the calculus as the *relation algebra* in the shape known today, allowing powerful equational reasoning about relational expressions (Tarski and Givant, 1987).

Often relational logic provides a more natural way to specify programs than purely functional formalisms: most so-called functions in computer science are actually *partial*, and in many contexts *multi-valued* functions are essential to characterize programs. With that in mind, since its first axiomatization by Tarski, a *point-free* version of relational logic, inspired by category theory, has been used in a variety of areas of computer science (Bird and de Moor, 1997; Oliveira, 2007, 2009) in order to specify and reason about programs, due to its high simplicity and ease of manipulation, generalizing the functional point-free notation popularized by John Backus (Backus, 1978).

Relations are also a natural framework in which to represent certain data domains, the classic example being *relational algebra*, that formalizes operations over relational databases and evolved from the calculus first proposed by Edgar F. Codd (Codd, 1970). They are also a core concept in MDE: just consider many-to-many associations and the natural notion of inverse navigations. Therefore, relational expressions will prove suitable to represent not only transformations but also their data domains, an aspect that relational specification languages fully exploit. In fact the relational logic behind the Alloy specification language is inspired by both relation algebra and relational algebra (Jackson, 2012, p. 326).

This chapter presents the relational formalism that will be used throughout the dissertation. Section 2.1 presents the syntax of the supported language, while Section 2.2 presents the associated type system. Section 2.3 presents the semantics that allow the evaluation of relational specifications. Finally, Section 2.4 presents some useful properties of relational expressions. Section 2.5 wraps up the chapter and comments on the choice of this formalism. Our formalism is mainly inspired by the application of relation algebra to program construction by Bird and de Moor (1997) and by the lightweight, set-theoretic, approach followed in Alloy.

2.1 Syntax

The relational formalism used throughout this dissertation is essentially a characterization of first-order logic with operators from the calculus of relations, extended with the transitive closure operator (sometimes called *transitive-closure logic*). As a running example, recall the object-relational problem introduced in Chapter 1 and the very simple constraint on `CD` meta-models, forcing the `general` association over classes to be acyclic, already in relational logic notation:

$$\forall\ c : \texttt{Class} \mid \neg(\langle c, c\rangle \in \texttt{general}^{+}) \qquad \textsc{AcyclicGen}$$

Basic relational expressions Relational expressions (nonterminal *Expr* at Figure 2.1) represent relations between sets of elements drawn from the universe, denoted by uppercase Latin characters ($R, S, ...$). A relation R that relates $n > 0$ elements is said to have arity n (retrieved by $|R|$) or to be n-ary. Our formalism is assumed to be many-sorted over $\mathcal{S}$, with the fact that an element a belongs to a sort $A \in \mathcal{S}$ being denoted by $a : A$.

$$
\begin{aligned}
\mathit{Expr} \;::=\; & \mathit{Sort} \mid \mathit{Rel} \mid \mathit{Var} \mid \mathsf{id}_{\mathit{Sort}} \mid \mathit{Expr} \circ \mathit{Expr} \mid \mathit{Expr}^{\circ} \mid \overline{\mathit{Expr}} \mid \\
& \mathit{Expr} \cup \mathit{Expr} \mid \mathit{Expr} \cap \mathit{Expr} \mid \mathit{Expr} \times \mathit{Expr} \mid \\
& \mathit{Expr}^{+} \mid \mathit{Expr} \rhd \mathit{Expr} \mid \mathit{Expr} \lhd \mathit{Expr} \mid \\
& \texttt{'\{'}\ \mathit{Decl}\ \{\texttt{,}\ \mathit{Decl}\}\ \texttt{'|'}\ \mathit{Form}\ \texttt{'\}'} \mid \\
& \pi_{1\,\mathit{Sort}\times\mathit{Sort}} \mid \pi_{2\,\mathit{Sort}\times\mathit{Sort}} \mid i_{1\,\mathit{Sort}+\mathit{Sort}} \mid i_{2\,\mathit{Sort}+\mathit{Sort}} \mid \\
& \mathsf{cons}_{\mathit{Sort}} \mid \mathsf{nil}_{\mathit{Sort}} \mid \mathsf{length}_{\mathit{Sort}} \\
\mathit{Form} \;::=\; & \mathsf{True} \mid \mathsf{False} \mid \mathit{Tuple} \in \mathit{Expr} \mid \mathit{Expr} \subseteq \mathit{Expr} \mid \\
& \neg \mathit{Form} \mid \mathit{Form} \wedge \mathit{Form} \mid \mathit{Form} \vee \mathit{Form} \mid \\
& \forall\ \mathit{Decl}\ \texttt{'|'}\ \mathit{Form} \mid \exists\ \mathit{Decl}\ \texttt{'|'}\ \mathit{Form} \\
\mathit{Decl} \;::=\; & \mathit{Var} : \mathit{Expr} \\
\mathit{Tuple} \;::=\; & \langle \mathit{Var}\ \{\texttt{,}\ \mathit{Var}\} \rangle \\
\mathit{Var} \;::=\; & \mathit{Iden} \mid (\mathit{Var}, \mathit{Var}) \\
\mathit{Rel} \;::=\; & \mathit{Iden} \\
\mathit{Sort} \;::=\; & \mathit{Iden} \mid \mathit{Sort} \times \mathit{Sort} \mid \mathit{Sort} + \mathit{Sort} \mid [\mathit{Sort}]
\end{aligned}
$$

Figure 2.1: Concrete syntax of relational expressions and formulae.

Sorts are typically identified by upper-case Latin characters ($A, B, ...$), while variables are usually denoted by their container sort identifier in lower-case characters ($a, b, ...$) (possibly with indexes). In this context, an n-ary relation R may be seen as a subset of the cartesian product of n sorts $A_1, ..., A_n \in \mathcal{S}$, a fact denoted by $R : A_1 \leftrightarrow ... \leftrightarrow A_n$.

The strength of relational logic lies on its ability to build complex relations through expressive combinators, that simplify formula reading and reasoning. The fundamental operation is the sequential *composition* of relations as $S \circ R$, for any two relations R and S with $|R| + |S| - 2 > 0$. For any sort A, the *identity* binary relation id_A embodies variable equality between elements of A, being the neutral element of composition. The sort identifier is dropped whenever clear from context. The *inverse* operation is defined for every binary relation R as R°, and inverts the order of the comprising tuples. Two relations R and S with the same arity can also be combined under the typical set-theoretic operations of *union* $R \cup S$ and *intersection* $R \cap S$, while a relation R may have its *complement* $\overline{R}$ calculated. The *difference* operation can be derived from the complement as $R \setminus S = R \cap \overline{S}$. For any two relations R and S their *cartesian product* $R \times S$ with arity $|R| + |S|$ is also assumed to exist. Variables $a : A$ can also be called in expressions, denoted by the nonterminal Var. For any sort B, it is possible to define the

constant relation $\underline{a}_B$ that relates every element from B with a, defined by the cartesian product $A \times a$. Given an unary relation R, a binary relation S may be left- or right-restricted by R values as $R \rhd S$ and $S \lhd R$, respectively. Nonterminal symbol Rel denotes *free relation variables*, denoted by typewriter upper-case Latin characters ($\mathsf{R}, \mathsf{S}, ...$), which must be assigned concrete values at evaluation time, essential to incorporate external information into relational expressions. In ACYCLICGEN, `general` is one such relation: whenever the constraint is tested, it must be assigned a constant tuple set that embodies the general association of the model instance being checked. Relations may also be constructed by *comprehension*, selecting the variable tuples for which the provided formula holds.

Relational logic extends first-order logic by supporting *transitive closure* and *reflexive transitive closure* operators, that allow the definition of (otherwise inexpressible) reachability properties. The transitive closure R^+ of a binary relation R is the smallest transitive relation that contains R. The reflexive transitive closure R^* of a binary relation R is the smallest relation containing R that is transitive and reflexive, and can be defined as $R^* = R^+ \cup \mathsf{id}$ (these properties will be defined shortly in Section 2.4). In ACYCLICGEN, the transitive closure $\texttt{general}^+$ retrieves all parent classes of a given class.

Expressions over products, sums and lists are also supported. These will be explored further below, where these composite sorts are presented.

Relational formulae Formulae (nonterminal $Form$ at Figure 2.1) are expressions that (given an interpretation and an assignment for the free variables) evaluate to either true or false, and are typically denoted by lower case Greek characters ($\phi, \psi, ...$). Atomic formulae either consist of the application of relations to tuples of variables with the appropriate arity, as $\langle a_1, ..., a_n \rangle \in R$ for $|R| = n$, or the comparison of relations with the same arity through *inclusion* as $R \subseteq S$. Relational equality is typically defined by anti-symmetry as $R = S \equiv R \subseteq S \wedge S \subseteq R$. In ACYCLICGEN there is a single atomic formula $\langle c, c \rangle \in \texttt{general}^+$ that tests whether c is its own super class. Since variables can be called in expressions, this formula could also be rewritten as $\langle c \rangle \in \texttt{general}^+ \circ c$, giving it a more object-oriented flavor.

Relational formulae are built from typical logical connectives, concretely, unary *negation* $\neg$ and binary *conjunction* $\wedge$, *disjunction* $\vee$ and the derived *implication* $\Rightarrow$ and *equivalence* $\equiv$, as well as the boolean constants True and False. Element variables

are introduced by first-order *universal* $\forall$ and *existential* $\exists$ quantifications (second-order quantifications of relation variables are not allowed). For an unary relation R, notation $(\forall\ a : R \mid \phi)$ and $(\exists\ a : R \mid \phi)$ is used to restrict the *range* of a to values belonging to R. Nested variable quantifications are typically abbreviated as $(\forall\ a_{11}, ..., a_{1\,i} : R_1, ..., a_{n1}, ..., a_{nj} : R_n)$ (and similarly for existential quantifications). Occasionally the *uniqueness* quantification $(\exists^1\ a : R \mid \phi)$ will also be used, which abbreviates the formula $(\exists\ a : R \mid \phi \wedge \neg(\exists\ a' : R \mid \phi \wedge a \neq a'))$. In ACYCLICGEN, the constraint quantifies over all classes c, and states that it may not belong to its own super classes.

To improve readability, if-then-else statements are occasionally used, where expressions $\mathbf{if}\ \phi\ \mathbf{then}\ \psi\ \mathbf{else}\ \theta$ abbreviates $(\phi \wedge \psi) \vee (\neg\phi \wedge \theta)$.

2.2 Static Semantics

Type systems provide extra information about the behavior of expressions, and improve the ability to statically detect errors, filtering out expressions whose result would be deemed ill-formed (Pierce, 2002). In the context of relational logic, a (lightweight) type system can be simulated by assuming the formalism to be many-sorted.

Order-sorted logic The chosen relational formalism is *order-sorted*, meaning that $\mathcal{S}$ forms a lattice of sorts. In this setting, sorts are not necessarily disjoint, but may be contained one in another, entailed by the partial order $\sqsubseteq_{\mathcal{S}}$: for any two sorts $A_1, A_2 \in \mathcal{S}$ such that $A_1 \sqsubseteq_{\mathcal{S}} A_2$, $a : A_1$ implies $a : A_2$. Unrelated sorts, i.e., $A_1 \not\sqsubseteq_{\mathcal{S}} A_2 \wedge A_2 \not\sqsubseteq_{\mathcal{S}} A_1$, are disjoint. Thus, this setting embodies a simple hierarchic type system. Two sorts A and B always have a greatest lower-bound $A \sqcap_{\mathcal{S}} B$—which, at the limit, is the smallest sort $\emptyset$ that contains no elements—and smallest upper-bound $A \sqcup_{\mathcal{S}} B$—which, at the limit, is the largest sort $\mathcal{U}$, containing all elements of the universe. Sorts are either primitive ($\mathbb{Z}, \mathbb{N}, ...$) or user-defined (denoted by typewriter upper-case Latin characters), as the sort `Class` in the running example. Table 2.1 summarizes the supported sorts and the corresponding type-checking operations.

For any relational formula ϕ or relational expression $R : A_1 \leftrightarrow ... \leftrightarrow A_n$, their type (retrieved by $\Gamma(\phi)$ or $\Gamma(R)$) is inferred by the typing rules from Figures 2.2 and 2.3, respectively. There, $\Gamma(R)^i$, for $1 \leqslant i \leqslant |R|$, denotes the sort of the ith component of R. To infer the type of an expression, the type of occurring free relations must be

definition	sort	test
primitive sort	$\mathbb{B}$	$a : \mathbb{B} \equiv a = \mathsf{True} \vee a = \mathsf{False}$
	$\mathbb{N}$	$a : \mathbb{N} \equiv a \in \mathbb{N}$
	$\mathbb{Z}$	$a : \mathbb{Z} \equiv a \in \mathbb{Z}$
	...	
user-defined	A	$a : \mathsf{A} \equiv a \in e\ (\mathsf{A})$
constants	$\mathcal{U}$	$a : \mathcal{U} \equiv \mathsf{True}$
	$\emptyset$	$a : \emptyset \equiv \mathsf{False}$
product	$A_1 \times A_2$	$(a_1, a_2) : A_1 \times A_2 \equiv a_1 : A_1 \wedge a_2 : A_2$
sum	$A_1 + A_2$	$(i_1\ a_1) : A_1 + A_2 \equiv a_1 : A_1$
		$(i_2\ a_2) : A_1 + A_2 \equiv a_2 : A_2$
list	$[A]$	$[\,] : [A] \equiv \mathsf{True}$
		$(h : t) : [A] \equiv h : A \wedge t : [A]$

Table 2.1: Supported sorts.

declared and provided in the starting context; this set of declarations is denoted by $\mathcal{R}$. In ACYCLICGEN, there is a single free relation, declared as `general`:`Class` $\leftrightarrow$ `Class`. In an order-sorted setting, relation application is not strict, meaning that a formula that applies a relation to a tuple with mismatching type is still well-formed and evaluates to false; the application of a relation $R : A_1 \leftrightarrow ... \leftrightarrow A_n$ to a tuple $\langle a_1, ..., a_n \rangle$ is equivalent to $a_1 : A_1 \wedge ... \wedge a_n : A_n \wedge \langle a_1, ..., a_n \rangle \in R$, meaning that it is only evaluated if the variables type-check.

Since sorts essentially represent sets of elements, each sort $A \in \mathcal{S}$ is assumed to give rise to a unary relation $A : \mathbb{P}(A)$ that represents the elements belonging to sort A (nonterminal *Sort* in Figure 2.1). In ACYCLICGEN, the sort `Class` also introduces a unary relation `Class` that contains all elements of the respective sort, over which the c quantification is defined. From these sort-derived unary relations, two families of relational constants emerge. For any n sorts $A_1, ..., A_n \in \mathcal{S}$, the universal relation $\top_{A_1 \times ... \times A_n}$, the largest relation over those sorts, is defined by their cartesian product $A_1 \times ... \times A_n$; for any arity n, the empty n-ary relation $\bot_n$ is assumed is defined by the n-ary cartesian product of the empty sort $\emptyset$ (the subscripts are dropped if no ambiguity arises).

$$\frac{}{\Gamma \vdash \mathsf{True}} \qquad \frac{}{\Gamma \vdash \mathsf{False}}$$

$$\frac{\Gamma \vdash a_1 : A_1 \quad ... \quad \Gamma \vdash a_n : A_n \quad \Gamma \vdash R : B_1 \leftrightarrow ... \leftrightarrow B_n}{\Gamma \vdash \langle a_1, ..., a_n \rangle \in R}$$

$$\frac{\Gamma \vdash R : A_1 \leftrightarrow ... \leftrightarrow A_n \quad \Gamma \vdash S : B_1 \leftrightarrow ... \leftrightarrow B_n}{\Gamma \vdash R \subseteq S}$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \neg\phi}$$

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \qquad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}$$

$$\frac{\Gamma \vdash R : \mathbb{P}\, A \quad \Gamma \oplus a : A \vdash \phi}{\Gamma \vdash \forall\ a : R \mid \phi} \qquad \frac{\Gamma \vdash R : \mathbb{P}\, A \quad \Gamma \oplus a : A \vdash \phi}{\Gamma \vdash \exists\ a : R \mid \phi}$$

Figure 2.2: Type-inference rules for relational formulae.

Composite datatypes To improve the expressiveness of the formalism, $\mathcal{S}$ is assumed to be closed under binary *products* $(\times)$ (i.e., pairs of elements) and *coproducts* $(+)$ (i.e., tagged unions), thus $(\forall\ A_1, A_2 \in \mathcal{S} \mid A_1 \times A_2 \in \mathcal{S} \wedge A_1 + A_2 \in \mathcal{S})$. Pairs $(a_1, a_2) : A_1 \times A_2$ are deconstructed by projections $\pi_{1\,A_1 \times A_2}$ and $\pi_{2\,A_1 \times A_2}$ that retrieve its first $(a_1 : A_1)$ or second $(a_2 : A_2)$ element, respectively. Tagged unions $A_1 + A_2$ are introduced by injections $i_{1\,A_1+A_2}$ and $i_{2\,A_1+A_2}$, that tag elements $a_1 : A_1$ or $a_2 : A_2$ to the left $(i_1\ \ a_1 : A_1 + A_2)$ or right $(i_2\ \ a_2 : A_1 + A_2)$ sort, respectively. The type subscript of projections and injections is dropped when there is no ambiguity. Notice that the notation i_1 and i_2 is overloaded to represent both the operations that tag the element and the tag itself (much like constructors in the Haskell functional language). Type-checking operations over composite types are also presented in Table 2.1.

Useful combinators over composite datatypes may be derived from the primitive combinators. Expressions may introduce tuples through the *fork* operator $R \vartriangle S : A \leftrightarrow B \times C$ that applies two relations $R : A \leftrightarrow B$ and $S : A \leftrightarrow C$ to the same input value, defined as $R \vartriangle S = (\pi_1^\circ \circ R) \cap (\pi_2^\circ \circ S)$. In a similar manner, disjoint unions may be deconstructed by the *either* operation $R \triangledown S : B + C \leftrightarrow A$ that applies $R : B \leftrightarrow A$ or $S : C \leftrightarrow A$ to the input depending on the tag, defined as $R \triangledown S = (R \circ {i_1}^\circ) \cup (S \circ {i_2}^\circ)$.

For every sort $A \in \mathcal{S}$, $\mathcal{S}$ is also assumed to be closed under list types $[A]$. These

$$\frac{\Gamma\,(a) = A}{\Gamma \vdash a : A} \qquad \frac{\Gamma\,(\mathsf{R}) = A_1 \leftrightarrow ... \leftrightarrow A_n}{\Gamma \vdash \mathsf{R} : A_1 \leftrightarrow ... \leftrightarrow A_n}$$

$$\frac{}{\Gamma \vdash A : \mathbb{P}\,A} \qquad \frac{}{\Gamma \vdash \mathsf{id}_A : A \leftrightarrow A}$$

$$\frac{\Gamma \vdash R : A_1 \leftrightarrow ... \leftrightarrow A_n \qquad \Gamma \vdash S : B_1 \leftrightarrow ... \leftrightarrow B_m}{\Gamma \vdash S \circ R : A_1 \leftrightarrow ... \leftrightarrow A_{n-1} \leftrightarrow B_2 \leftrightarrow B_m}$$

$$\frac{\Gamma \vdash R : A \leftrightarrow B}{\Gamma \vdash R^{\circ} : B \leftrightarrow A} \qquad \frac{\Gamma \vdash R : A \leftrightarrow B}{\Gamma \vdash \overline{R} : A \leftrightarrow B}$$

$$\frac{\Gamma \vdash R : A_1 \leftrightarrow ... \leftrightarrow A_n \qquad \Gamma \vdash S : B_1 \leftrightarrow ... \leftrightarrow B_n}{\Gamma \vdash R \cup S : A_1 \sqcup_{\mathcal{S}} B_1 \leftrightarrow ... \leftrightarrow A_n \sqcup_{\mathcal{S}} B_n}$$

$$\frac{\Gamma \vdash R : A_1 \leftrightarrow ... \leftrightarrow A_n \qquad \Gamma \vdash S : B_1 \leftrightarrow ... \leftrightarrow B_n}{\Gamma \vdash R \cap S : A_1 \sqcap_{\mathcal{S}} B_1 \leftrightarrow ... \leftrightarrow A_n \sqcap_{\mathcal{S}} B_n}$$

$$\frac{\Gamma \vdash R : A_1 \leftrightarrow ... \leftrightarrow A_n \qquad \Gamma \vdash S : B_1 \leftrightarrow ... \leftrightarrow B_m}{\Gamma \vdash R \times S : A_1 \leftrightarrow ... \leftrightarrow A_n \leftrightarrow B_1 \leftrightarrow ... \leftrightarrow B_m}$$

$$\frac{\Gamma \vdash R : A \leftrightarrow B \qquad S : \mathbb{P}\,C}{\Gamma \vdash S \lessdot R : A \sqcap_{\mathcal{S}} C \leftrightarrow B} \qquad \frac{\Gamma \vdash R : A \leftrightarrow B \qquad S : \mathbb{P}\,C}{\Gamma \vdash R \gtrdot S : A \leftrightarrow B \sqcap_{\mathcal{S}} C}$$

$$\frac{\Gamma \vdash R_1 : \mathbb{P}\,A_1 \quad ... \quad \Gamma \vdash R_1 : \mathbb{P}\,A_1 \quad \Gamma \oplus a_1 : A_1 \oplus ... \oplus a_n : A_n \vdash \phi}{\Gamma \vdash \{\, a_1 : R_1, ..., a_n : R_n \mid \phi \,\} : A_1 \leftrightarrow ... \leftrightarrow A_n}$$

$$\frac{}{\Gamma \vdash \pi_{1\,A \times B} : A \times B \leftrightarrow A} \qquad \frac{}{\Gamma \vdash \pi_{2\,A \times B} : A \times B \leftrightarrow B}$$

$$\frac{}{\Gamma \vdash i_{1\,A+B} : A \leftrightarrow A + B} \qquad \frac{}{\Gamma \vdash i_{2\,A+B} : B \leftrightarrow A + B}$$

$$\frac{}{\Gamma \vdash \mathsf{cons}_{[A]} : A \times [A] \leftrightarrow [A]} \qquad \frac{}{\Gamma \vdash \mathsf{nil}_{[A]} : [A]}$$

$$\frac{}{\Gamma \vdash \mathsf{length}_{[A]} : [A] \leftrightarrow \mathbb{N}} \qquad \frac{\Gamma \vdash R : A \leftrightarrow B}{\Gamma \vdash R^{+} : A \leftrightarrow B}$$

Figure 2.3: Type-inference rules for primitive relational combinators.

$$
\begin{aligned}
&[\![R \subseteq S]\!]^e \quad \equiv \\
&\quad [\![\forall\ a_1 : \Gamma(R)^1, ..., a_{|R|} : \Gamma(R)^{|R|} \mid \langle a_1, ..., a_{|R|}\rangle \in R \Rightarrow \langle a_1, ..., a_{|R|}\rangle \in S]\!]^e \\
&[\![\neg\phi]\!]^e \quad \equiv \neg[\![\phi]\!]^e \\
&[\![\phi \wedge \psi]\!]^e \quad \equiv [\![\phi]\!]^e \wedge [\![\psi]\!]^e \\
&[\![\phi \vee \psi]\!]^e \quad \equiv [\![\phi]\!]^e \vee [\![\psi]\!]^e \\
&[\![\forall\ a : R \mid \phi]\!]^e \equiv \forall\ \mathsf{k} : \Gamma(R)^1 \mid [\![\langle a\rangle \in R \Rightarrow \phi]\!]^{e \oplus a \mapsto \mathsf{k}} \\
&[\![\exists\ a : R \mid \phi]\!]^e \equiv \exists\ \mathsf{k} : \Gamma(R)^1 \mid [\![\langle a\rangle \in R \wedge \phi]\!]^{e \oplus a \mapsto \mathsf{k}}
\end{aligned}
$$

Figure 2.4: Semantics of relational formulae.

are constructed by the $\mathsf{cons}_{[A]}$ (usually denoted in infix notation $(h : t) = \mathsf{cons}\ (h, t)$) and $\mathsf{nil}_{[A]}$ (usually denoted simply by $[\,]$) operations[1]. Operations head and tail, that deconstruct lists, can be derived as $\mathsf{head} = \pi_1 \circ \mathsf{cons}^\circ$ and $\mathsf{tail} = \pi_2 \circ \mathsf{cons}^\circ$, since cons° splits a non-empty list into its head and tail. Other standard operations over lists, like length are assumed to exist[2].

2.3 Dynamic Semantics

To be evaluated, the combinators that comprise relational formulae and expressions must be given concrete semantics.

Formula semantics For a relational formula ϕ, its evaluation $[\![\phi]\!]^e$ calculates its truth value given an environment e that binds free variables to concrete tuple sets. Logical connectives and quantifications are interpreted under standard boolean semantics, as depicted in Figure 2.4. Relational inclusion is expanded to its point-wise definition: a relation is contained in another if all its elements are. Inequations are converted to their point-wise definition, with the introduced variables assumed to be fresh. Quantifications

[1] In the calculus of binary relations, recursive data types are defined as fixed points of regular functors. Given a base functor F, let μF denote the inductive type generated by its least fixed point. For example, for lists we would have $[A] = \mu(\underline{1} + (\underline{A} \times \mathsf{Id}))$. Two unique functions $\mathsf{in} : F\ (\mu F) \to \mu F$ and $\mathsf{out} : \mu F \to F\ (\mu F)$, that are each other's inverse, are associated with each data type μF, which allow to encode and inspect values of the given type, respectively. In this context, the list constructors can be defined as $\mathsf{nil} = \mathsf{in} \circ i_1$ and $\mathsf{cons} = \mathsf{in} \circ i_2$, (thus $\mathsf{nil} \triangledown \mathsf{cons} = \mathsf{in}$).

[2] Recursion in the calculus of binary relations is typically defined using well-known recursion patterns, namely folds (catamorphisms) and unfolds (anamorphisms), that encode the recursion patterns of iteration and coiteration, respectively. In lists, fold $(\!|R|\!) : [A] \leftrightarrow B$ consumes lists according to an algebra $R : F\ B \leftrightarrow B$, while the dual unfold $[\!(S)\!] : B \leftrightarrow [A]$ produces lists according to a coalgebra $S : F\ A \leftrightarrow A$. In this context, length can be defined as $(\!|\underline{0} \triangledown (\mathsf{succ} \circ \pi_2)|\!)$.

$$
\begin{array}{ll}
[\![\langle a\rangle \in A]\!]^e & \equiv e\ (a) : A \\
[\![\langle a_1, ..., a_n\rangle \in \mathsf{R}]\!]^e & \equiv (e\ (a_1), ..., e\ (a_n)) \in e\ (\mathsf{R}) \\
[\![\langle a_1\rangle \in a_2]\!]^e & \equiv e\ (a_1) = e\ (a_2) \\
[\![\langle a_1, a_2\rangle \in \mathsf{id}]\!]^e & \equiv e\ (a_1) = e\ (a_2) \\
[\![\langle a_1, ..., a_{|R|-1}, b_2, ..., b_{|S|}\rangle \in S \circ R]\!]^e & \equiv \\
\quad \exists\ \mathsf{k} : \Gamma(R)^{|R|} \sqcap_{\mathcal{S}} \Gamma(S)^1 \mid [\![\langle a_1, ..., a_{|R|-1}, k\rangle \in R]\!] \wedge [\![\langle k, b_2, ..., b_{|S|}\rangle \in S]\!]^{e\, e \,\oplus\, k \mapsto \mathsf{k}} & \\
[\![\langle a, b\rangle \in R^\circ]\!]^e & \equiv [\![\langle b, a\rangle \in R]\!]^e \\
[\![\langle a_1, ..., a_n\rangle \in \overline{R}]\!]^e & \equiv \neg [\![\langle a_1, ..., a_n\rangle \in R]\!]^e \\
[\![\langle a_1, ...a_n\rangle \in R \cup S]\!]^e & \equiv [\![\langle a_1, ...a_n\rangle \in R]\!]^e \vee [\![\langle a_1, ...a_n\rangle \in S]\!]^e \\
[\![\langle a_1, ...a_n\rangle \in R \cap S]\!]^e & \equiv [\![\langle a_1, ...a_n\rangle \in R]\!]^e \wedge [\![\langle a_1, ...a_n\rangle \in S]\!]^e \\
[\![\langle a_1, ..., a_n, b_1, ..., b_m\rangle \in R \times S]\!]^e & \equiv \langle a_1, ..., a_n\rangle \in R \wedge \langle b_1, ..., b_m\rangle \in S \\
[\![\langle a_1, a_2\rangle \in R^+]\!]^e & \equiv [\![\langle a_1, a_2\rangle \in R]\!]^e \vee [\![\langle a_1, a_2\rangle \in R \circ R]\!]^e \vee\ ... \\
[\![\langle a_1, a_2\rangle \in S \lessdot R]\!]^e & \equiv \langle a_1, a_2\rangle \in R \wedge \langle a_1\rangle \in S \\
[\![\langle a_1, a_2\rangle \in R \gtrdot S]\!]^e & \equiv \langle a_1, a_2\rangle \in R \wedge \langle a_2\rangle \in S \\
[\![\langle a_1, ..., a_n\rangle \in \{k_1 : R_1, ..., k_n : R_n \mid \phi\}]\!]^e & \equiv \\
\quad [\![\langle a_1\rangle \in R_1 \wedge ... \wedge \langle a_n\rangle \in R_n \wedge \phi\ [k_1 \mapsto a_1, ..., k_n \mapsto a_n]]\!]^e & \\
[\![\langle a_1, (a_2, a_3)\rangle \in \pi_1]\!]^e & \equiv e\ (a_1) = e\ (a_2) \\
[\![\langle a_1, (a_2, a_3)\rangle \in \pi_2]\!]^e & \equiv e\ (a_1) = e\ (a_3) \\
[\![\langle a_2, a_1\rangle \in i_1]\!]^e & \equiv e\ (a_2) = i_1\ (e\ (a_1)) \\
[\![\langle a_2, a_1\rangle \in i_2]\!]^e & \equiv e\ (a_2) = i_2\ (e\ (a_1)) \\
[\![\langle (a_1, l_1), l_2\rangle \in \mathsf{cons}]\!]^e & \equiv e\ (a_1) : e\ l_1 = e\ l_2 \\
[\![\langle l\rangle \in \mathsf{nil}]\!]^e & \equiv e\ (l) = [\,] \\
[\![\langle l, n\rangle \in \mathsf{length}]\!]^e & \equiv \mathsf{length}\ (e\ (l)) = e\ (n)
\end{array}
$$

Figure 2.5: Semantics of relational expressions.

over $a : R$ are converted to quantifications at the meta-logic level that iterate through concrete elements of the type of R, denoted in sans serif typeface (k). Given two mappings, operation $\oplus$ *overrides* shared keys with the value of the second.

For a relational expression R, variable application $[\![\langle a_1, ..., a_n\rangle \in R]\!]^e$ is evaluated following the definitions in Figure 2.5 that expands the definitions of the relational operators. These are inspired by the set-theoretic representation of relations and are mostly self-explanatory. Note that at the primitive combinator level, the valuation of each element and relation variable must be retrieved from the environment e.

Expression execution When dealing exclusively with binary relations, expressions can be seen as *multi-valued functions* (functions that return sets of values). Thus, for a binary relation $R : A \leftrightarrow B$, variable application $\langle a, b\rangle \in R$ is usually denoted by $b \in R\ a$ (binary relations are assumed to be run from left to right, consuming A values and generating B values). This alternative semantics can be trivially defined by set comprehension as $[\![R]\!]^e\ a = \{\, b \mid [\![\langle a, b\rangle \in R]\!]^e, b \leftarrow B \,\}$. However, as should be expected, such definition is highly inefficient and may not even terminate if the output sort B is infinite. Even with a finite output sort, the execution of a relational expression may not terminate due to the occurrence of composition, that entails an existential quantification over a possibly infinite range of elements.

With product sorts, every relation with arity $n > 2$ may be converted to a binary version that consumes and/or produces pairs. Unary relations are typically represented by coreflexive relations (more on this in Section 2.4).

Datatype invariants The simple order-sorted type system may fall short when a more in-depth understanding of the specification is required: a more refined type system, that conveys more information about the data domains, allows the development of extended static checking procedures. In a more technical concern, types can also be used to improve the efficiency of execution (Pierce, 2002). One possible solution is to enhance data domains with datatypes *invariants* that restrict the range of acceptable values. In fact, the calculus of binary relations has shown be sufficiently expressive and manageable to perform this kind of tasks (Oliveira, 2009). As should be expected, this improved expressibility power comes at the cost of the decidability of the type-checking and type-inference.

A datatype invariant ϕ over domain A is interpreted as a subset of A. Such constrained datatype is denoted by A_ϕ, and an element is said to belong to it only if it also belongs to ϕ, i.e., $a : A_\phi \equiv a : A \wedge \langle a\rangle \in \phi$. For a binary relation $R : A \leftrightarrow B$, expression $R : A_\phi \leftrightarrow B_\psi$ denotes the fact that R, when fed values for which ϕ holds, only produces values for which ψ holds, i.e.:

$$\forall\ a : A, b : B \mid \langle a\rangle \in \phi \wedge \langle a, b\rangle \in R \Rightarrow \langle b\rangle \in \psi$$

2.4 Binary Relation Properties

Binary relations are the most commonly occurring relations—being intuitively seen as a procedure that given an input variable produces a (possibly empty) set of outputs—and as mentioned in Section 2.3, n-ary relations can always be converted to a binary version. Additionally, binary relations also benefit from interesting properties which deserve a deeper presentation. Some of the laws ruling the calculus of binary relations are presented in Appendix A.

Coreflexives An endo-relation $R : A \leftrightarrow A$ (a binary relation between the same sorts) is said to be *coreflexive* if it is a subset of the identity relation (i.e., $R \subseteq \mathsf{id}$), as opposed to *reflexive* relations, that are a superset of the identity relation (i.e., $\mathsf{id} \subseteq R$). Coreflexive relations share interesting properties and thus are usually differentiated by upper-case Greek characters ($\Phi, \Psi, ...$). Since they always return the same value taken as input, they act as filters of data: an input value is either outputted unmodified or nothing is outputted. They can also be used to represent sets (relations with arity $n = 1$) under the binary context. For a set $\phi, \psi, ... : \mathbb{P}\, A$, we denote the matching coreflexive by the respective upper-case Greek character $\Phi, \Psi, ... : A \leftrightarrow A$, i.e., $\langle a, a\rangle \in \Phi \equiv \langle a\rangle \in \phi$. Coreflexives have interesting algebraic properties that simplify their manipulation like, for example, $\Phi^\circ = \Phi$, $\Phi \circ \Phi = \Phi$, and $\Phi \circ \Psi = \Phi \cap \Psi$.

For every binary relation $R : A \leftrightarrow B$, we assume the existence of two special coreflexives that represent its *domain* ($\delta R : A \leftrightarrow A$) and *range* ($\rho R : B \leftrightarrow B$). By domain (range) we mean the exact set of values that are consumed (produced) by a relation, that is typically smaller than B (A). As should be expected, $R = R \circ \delta R = \rho R \circ R$.

A useful operator over coreflexives is the *guard*. For a coreflexive $\Phi : A \leftrightarrow A$, $\Phi? = (\Phi \triangledown (\mathsf{id} \setminus \Phi))^\circ : A + A \leftrightarrow A$ tags the input as a left or right value in a sum, depending on the result of the test Φ. Composed with an either $R \triangledown S : A + A \leftrightarrow B$, it allows the representation of conditionals $\Phi\ ?\ R : S = (R \triangledown S) \circ \Phi\ ? : A \leftrightarrow B$, which apply R if the input element belongs to Φ, and S otherwise.

Families of relations For every binary relation $R : A \leftrightarrow B$ we define its *kernel* and *image* as $\mathsf{ker}\ R : A \leftrightarrow A = R^\circ \circ R$ and $\mathsf{img}\ R : B \leftrightarrow B = R \circ R^\circ$, respectively. The kernel of a relation establishes a kind of equivalence relation between input values that share the same output, and vice-versa for its image. Thus, these artifacts can be used to explore certain properties of binary relations, as presented in Table 2.2. For instance, if

	kernel	**image**
reflexive	total	surjective
coreflexive	simple	injective

Table 2.2: Relation classification under kernel and image (Oliveira, 2007).

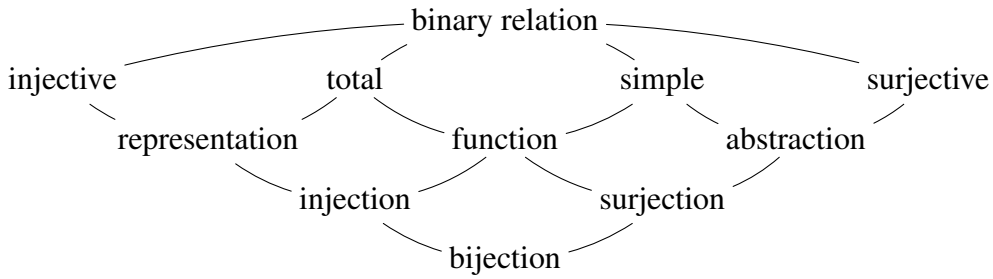


Figure 2.6: A taxonomy for binary relations (Oliveira, 2007).

the image of a relation is coreflexive, then no inputs share the same output, and thus the relation is injective.

Different combinations of these properties give rise to different classes of binary relations, as depicted in Figure 2.6 (Oliveira, 2007). Following this taxonomy, *functions* are the special class of relations which are simple and total. These will typically be denoted by lower-case Latin characters in order to be distinguished from relations, and typed as $f : A \to B$, with membership test usually denoted by $b = f\ a$. Partial functions (i.e., simple relations) are usually typed by $f : A \rightharpoonup B$.

Since the kernel (and image) of a relation relates inputs (outputs) that share outputs (inputs)—including themselves—it also contains information about the domain and range of a relation. Concretely, in the binary context, for a relation R, coreflexives denoting its domain δR may be calculated as $\ker R \cap \mathsf{id}$ and its range ρR as $\mathsf{img}\ R \cap \mathsf{id}$. As expected, if a relation is total then $\mathsf{id} \subseteq \ker R$, and thus $\delta R = \mathsf{id}$ (and similarly for surjective relations).

Orders Binary relations may also be used to represent orders over data domains. A *preorder* $R : A \leftrightarrow A$ over A is an endo-relation that is reflexive and *transitive* ($R \circ R \subseteq R$ or $\langle a_1, a_2 \rangle \in R \wedge \langle a_2, a_3 \rangle \in R \Rightarrow \langle a_1, a_3 \rangle \in R$). A preorder that is also *anti-symmetric* ($R \cap R^\circ \subseteq \mathsf{id}$ or $\langle a_1, a_2 \rangle \in R \wedge \langle a_2, a_1 \rangle \in R \Rightarrow a_1 = a_2$) is a *partial order*, while a preorder that is *symmetric* ($R = R^\circ$ or $\langle a_1, a_2 \rangle \in R \equiv \langle a_2, a_1 \rangle \in R$) is an *equivalence*

relation. If every pair of elements is comparable under a preorder $R : A \leftrightarrow A$ ($R \cup R^\circ = \top$ or $\langle a_1, a_2 \rangle \in R \vee \langle a_2, a_1 \rangle \in R$), then the preorder is said to be *total*; a total partial order is a *linear order*. Orders are typically depicted in infix notation $a_1\ R\ a_2$ if $\langle a_1, a_2 \rangle \in R$.

Point-free notation Relational expressions comprised of the introduced operators and compared through inequations, give rise to a simple *point-free* (or variable-free) notation that, along with the powerful equational laws from Appendix A, provide a simpler way to reason and manipulate relational expressions. In fact, with the introduction of products and forks, the resulting logical system is as expressive as first-order logic. Thus, point-wise formulae are sometimes converted to their point-free counterpart to ease manipulation, a method known as the *point-free transform* (Oliveira, 2009). The simplicity of this calculus of binary relations is manifest, for instance, in the definition of injectivity, as the following simple calculations demonstrate:

$$
\begin{array}{ll}
& R^\circ \circ R \subseteq \mathsf{id} \\
\equiv & \{\text{-}\subseteq \text{ (Figure 2.4) -}\} \\
& \forall\ a_2, a_2' : A_2 \mid \langle a_2, a_2' \rangle \in R^\circ \circ R \Rightarrow \langle a_2, a_2' \rangle \in \mathsf{id} \\
\equiv & \{\text{-}\circ \text{ (Figure 2.5) -}\} \\
& \forall\ a_2, a_2' : A_2 \mid (\exists\ a_1 : A_1 \mid \langle a_2, a_1 \rangle \in R \wedge \langle a_1, a_2' \rangle \in R^\circ) \Rightarrow \langle a_2, a_2' \rangle \in \mathsf{id} \\
\equiv & \{\text{-}^\circ, \mathsf{id} \text{ (Figure 2.5) -}\} \\
& \forall\ a_2, a_2' : A_2 \mid (\exists\ a_1 : A_1 \mid \langle a_2, a_1 \rangle \in R \wedge \langle a_2', a_1 \rangle \in R) \Rightarrow a_2 = a_2'
\end{array}
$$

2.5 Discussion

Our main goal when defining the logical system was to achieve a balance between the many-sorted, category theory-inspired, calculus of binary relations used in the "algebra of programming" (Bird and de Moor, 1997; Oliveira, 2009) and the loosely-typed and flexible language over n-ary relations supported by the Alloy/Kodkod relational specification languages, over which the technique from Part III is built. The two main gaps are precisely the binary vs. arbitrary arity of relations and the type system. This relationship between the calculus of binary relations and Alloy has already been previously explored (Frias et al., 2004; Macedo, 2010; Macedo and Cunha, 2012).

Basic binary relational expressions amount to the relation algebra formalized by Tarski (Tarski and Givant, 1987). Relation algebra is however only as expressive as

first-order logic restricted to three variables. The introduction of product types and forks gives rise to *fork algebra* (Frias, 2002), which is as expressive as unrestricted first-order logic. Further developments formalized the calculus of binary relations under category theory, where it embodies the notion of *allegory* (Freyd and Scedrov, 1990). Unlike relation (and fork) algebra, the categorical perspective is many-sorted and thus is better suited to represent and calculate programs (Bird and de Moor, 1997; Oliveira, 2009).

Instead of following this strict formalism, by defining an order-sorted formalism, we follow the approach of systems like Alloy (Edwards et al., 2004), that possess a laxer type system. The consequence is that operations like the union of two relations $R : A_1 \leftrightarrow A_2$ and $S : B_1 \leftrightarrow B_2$ is well-formed in our formalism, while in the categorical setting it would not be. Nonetheless, the type-inference and type-checking remain decidable.

Relations of arbitrary arity are required for the development of this dissertation, thus the choice of allowing relations with arity different than 2. Nonetheless, as has been shown, relations with arity $n > 2$ may be easily embedded in binary relations between tuples, while those with arity $n = 1$ may be encoded as coreflexives. Thus, when suitable, n-ary relations are interpreted as binary relations, in order to harness the full power of the calculus of binary relations.

Chapter 3

Bidirectional Transformation

Bidirectional transformation frameworks emerged in a variety of computer science disciplines and application contexts, and thus exhibit different shapes and properties (Czarnecki et al., 2009). In this chapter we provide a brief overview of the characteristics deemed relevant for the understanding of this dissertation, with a particular focus on bidirectional model transformation and least-change bidirectional transformation. Unless explicitly stated, in this chapter a "transformation" amounts to a partial function (i.e., a simple binary relation).

3.1 Basic Concepts

Scheme Bidirectional transformations take different shapes depending on the context being applied. In fact, the choice of the transformations' *scheme* greatly affects the expressivity and range of possible properties of the bidirectional transformation framework (Diskin, 2011; Pacheco et al., 2013).

In the most simple setting, a bidirectional transformation between data domains A and B may consist simply of transformations $\mathsf{to}_t : A \rightharpoonup B$ and $\mathsf{from}_t : B \rightharpoonup A$, denoting a bidirectional *mapping* t. Such is the case, for instance, when to_t and from_t are each others inverse, resulting in a bijection between the transformation domains (Brabrand et al., 2005). More relaxed frameworks may allow transformations between data domains containing different information as long as one of the transformations is injective, i.e., it is information-preserving (Mu et al., 2004). Bidirectional mappings over non-injective transformations may also be defined, but due to information loss, the system must disambiguate the range of possible solutions (Kawanaka and Hosoya,

2006).

In order to achieve better round-tripping properties, bidirectional transformations typically take into consideration extra information from the pre-state of the target domain (the transformation domain into which the update is being propagated), rather than just the state of the updated source. The bidirectional transformation framework of *lenses* developed by Foster et al. (2007) is one such example[1]. Bidirectional transformations in lenses are asymmetrical: a lens f is comprised of transformations $\mathsf{get}_f : A \rightharpoonup B$, that produces B values from A values, and $\mathsf{put}_f : A \times B \rightharpoonup A$, that produces A values from B values, possibly considering extra information from the pre-state of the A domain. For this reason, lenses are better suited for scenarios where B contains less information than A, i.e., B is a *view* of A. In fact, lenses are inspired by the seminal works on the *view-update problem* from the database community (how can updates on a database view be propagated back to the original tables?) studied by Bancilhon and Spyratos (1981), Dayal and Bernstein (1982) and Gottlob et al. (1988).

In the most general scenario both transformation domains have information not present in the other. Bidirectionality in this context is usually achieved by the symmetric *constraint maintainer* framework introduced by Meertens (1998)—since there is no "dominant" data domain, rather than specified by a transformation, maintainers are specified by a declarative constraint that defines the notion of consistent state. A maintainer T between transformation domains A and B consists of a consistency relation $T : A \leftrightarrow B$, that defines when two values are consistent, and two transformations $\overrightarrow{T} : A \times B \rightharpoonup B$ and $\overleftarrow{T} : A \times B \rightharpoonup A$ that set as the target domain either B or A, respectively, retrieving extra information from their pre-state.

Specification artifact One main difference between these schemes is the artifact through which the bidirectional transformation is specified—either through one of the *transformations*, from which the system must derive the backward propagation procedure, or through a declarative *constraint*, from which both the forward and backward propagation procedures must be derived.

Typically, transformation specifications are used in asymmetric contexts when there is a dominant flow of information, making one of the transformations "easier" to specify (although some advocate the specification of the backward transformation

[1] In this presentation we consider to be a lens framework any asymmetric framework comprised of transformations of the shape $\mathsf{get}_f : A \rightharpoonup B$ and $\mathsf{put}_f : A \times B \leftrightarrow A$, rather than just the combinatorial framework over semantic types first introduced by Foster et al. (2007).

instead (Pacheco et al., 2014)). This kind of framework is also more prone to compositional reasoning, as transformations are typically composable. Such is the case of lenses, where writing a forward transformation entails the backward semantics. Constraint specifications usually consist of consistency relations that define when the system is consistent, and is the approach followed by the constraint maintainer framework. Constraints are typically more expressive than transformations, as they may naturally represent partial and many-to-many relationships. To achieve a bidirectional transformation however, either a constraint is given executable semantics—like embedding them on a constraint solving problem—or concrete transformations are syntactically derived from it. This kind of framework is also not prone to compositional reasoning (Meertens, 1998, p. 42).

Bidirectionalization technique *Bidirectionalization* is the process through which the bidirectional behavior of a bidirectional transformation framework is attained[2]. These can be broadly divided into three classes.

Combinatorial transformation languages are typically domain-specific languages where each primitive or combinator is embedded with both forward and backward semantics (Foster et al., 2007; Hu et al., 2008; Pacheco and Cunha, 2010; Hofmann et al., 2011). In this kind of framework, when the user writes a transformation she is implicitly defining both transformations, which are correct-by-construction. The strength of these frameworks lies on the fact that combinators are expected to preserve the round-tripping properties given opaque lenses.

In contrast, *syntactic* approaches produce the bidirectional transformation through the syntactic analysis of the specification artifact at compilation time (Brabrand et al., 2005; Kawanaka and Hosoya, 2006; Matsuda et al., 2007; Ehrig et al., 2007). These are required if operational transformations are to be derived from declarative specifications, when a global analysis of the program is required or in the bidirectionalization of general-purpose languages. Other approaches require extra information from the state of the system to perform backward evaluation, and thus post-pone the generation of the backward transformation to run-time (Voigtländer, 2009). In these *semantic* approaches, backward evaluation is performed at run-time by analyzing the state of specific executions. One advantage of these frameworks is that they may abstract away

[2]Some literature (Czarnecki et al., 2009) considers "bidirectionalization" the process through which a forward transformation is given backward semantics, excluding combinatorial approaches where the putback is implicitly defined in each combinator. Here a broader perspective is considered.

from the concrete language over which the transformation was specified. Classification is many times not as straight-forward as choosing one of these approaches. For instance, some syntactic techniques may generate transformations that perform additional tests at run-time, borrowing aspects from semantic techniques (Voigtländer et al., 2010; Cicchetti et al., 2010).

Update representation Transformations may be provided different levels of information about how the post-state of the source domain was attained[3]. In *state-based* bidirectional transformations, transformations take as input only the post-state value of the source, ignoring how it was attained from the pre-state value. These can lead to incorrect updates, since, for instance, the removal and insertion of an object with the same name may be identified as a modification. This problem of matching information from the pre- and post-states is known as the *alignment* problem, and is still an active research topic in bidirectional transformation (Barbosa et al., 2010; Diskin et al., 2011; Pacheco et al., 2012).

One possible solution is to rely on *delta-based* (or operation-based) transformations (Diskin et al., 2011), where besides the updated state, transformations are given extra information on how it was attained from the pre-state. Concrete instantiations of operation-based frameworks include the representation of the exact sequence of editing operations (a *history-based*, or directed, representation) or simply establishing a correspondence between elements of the original and updated states (a *canonical*, or symmetric, representation). As the name implies, history-based updates can be converted into canonical updates by removing possible redundancies in the editing sequence. However, operation-based approaches may be more complex to implement, and sometimes there is still the need to restrict the range of possible operations in order to avoid an excessive complexity of the system.

Multi-valued transformations As already stated, and shall be made evident in Section 3.2, round-tripping laws are sufficiently loose to allow multiple valid update propagations. Typically, the bidirectionalization technique chooses at design time the specific behavior of the transformations, in order to render the framework functional and effective in practice. This however may reduce the updatability of the system, since

[3]This axis regards representability of *vertical* updates—the traceability between the pre- and post-states of the source domain being consumed by the transformation—in contrast to that of *horizontal* updates—the traceability between the two transformation domains (Diskin et al., 2011).

particular execution environments (like those with domain constraints) may render some updates invalid. An alternative would be to allow *multi-valued* transformations, enabling transformations to produce multiple valid values at each step, thus not committing to particular values and not reducing the choices of succeeding transformations. Multi-valued transformations are not necessarily non-deterministic: if the same set of values is always produced by a transformation, this is actually a deterministic procedure. Pure non-determinism—like simply selecting a random element from the output of a multi-valued transformation—is undesirable as it reduces the predictability of the system (Stevens, 2010).

Transformation domain Bidirectional transformation frameworks are defined with different application scenarios in mind, which has implications on the data structures manipulated by the transformations. While some are clearly more expressive than others, there is always a trade-off between the expressiveness, the efficiency and the complexity of the transformations over those structures.

Strings are one of the simplest data structures, defining ordered unstructured data. This simplicity allows extremely simple and efficient string manipulation transformations (Bohannon et al., 2008). Data in string format abounds, thus while simple, these prove to be extremely useful. *Trees* provide a way to define hierarchical structured data, comprising most of the possible types in common programming languages (Foster et al., 2007). *Graph* structures generalize trees by allowing cycles and multiple parents, providing means to represent more complex data structures (Hidaka et al., 2010; Hermann et al., 2011). Typically, unless lacking associations, models are isomorphic to graphs. However, graph manipulation and transformation is much more complex. Data can also be presented as *relations*, sets of tuples, as in the representation of relational databases (Bohannon et al., 2006) or spreadsheets (Cunha et al., 2012). Graphs (and thus models) can also be embedded in binary relations. If relations of arbitrary arity are allowed, these are more expressive than graphs (they are instead isomorphic to hyper-graphs). Both graphs and relations can eventually be encoded in trees, although some of its characteristics will not be naturally captured and may not be exploited by transformation systems.

3.2 Bidirectional Transformation Properties

Whatever the chosen scheme, transformations are expected to follow certain behavior rules that render them *well-behaved*. In bidirectional transformations, this usually entails some correctness (updates restore consistency), invertibility (updates on one of the values are preserved on the opposite) or stability (null updates on one of the domains should induce null updates on the opposite) properties.

3.2.1 Round-tripping Properties

In the case of a bidirectional mapping t between transformation domains A and B, the transformation may be expected to be at least left- or right-invertible.

$$a_0 \in \mathsf{from}_t\ b_0 \Rightarrow b_0 \in \mathsf{to}_t\ a_0 \qquad \textsc{Left-Invertible}$$

$$b_0 \in \mathsf{to}_t\ a_0 \Rightarrow a_0 \in \mathsf{from}_t\ b_0 \qquad \textsc{Right-Invertible}$$

Essentially, these laws force one of the transformations to be injective in order to be invertible. Given some additional totality assumptions, such mapping represents either an abstraction (if A is "larger" than B) or a refinement (if A is "smaller" than B) (Oliveira, 2007). If a transformation is both left- and right-invertible, to_t and from_t are isomorphisms between their domains (and, if total, bijections). Since the transformations in mappings have no knowledge of the pre-state of the target domain, there is no notion of stability.

In lens frameworks, the putback is aware of the pre-state of the target domain, and thus is expected to follow stricter rules.

$$a \in \mathsf{put}_f\ (\_, b_0) \Rightarrow b_0 \in \mathsf{get}_f\ a \qquad \textsc{PutGet}$$

$$b_0 \in \mathsf{get}_f\ a_0 \Rightarrow a_0 \in \mathsf{put}_f\ (a_0, b_0) \qquad \textsc{GetPut}$$

PUTGET, also known as *acceptability*, is the lenses version of invertibility, stating that the update on a view must somehow be preserved on the updated source a, and GETPUT, also known as *stability*, guarantees that, if a view is not updated, then neither will the source when applying the backward transformation. In the regular lens framework proposed by Foster et al. (2007), lenses are expected to follow these two laws.

Definition 3.1 (very well-behaved lens)**.** *A lens f between transformation domains A*

and B, comprised by transformations $\mathsf{get}_f : A \rightharpoonup B$ and $\mathsf{put}_f : A \times B \rightharpoonup A$, is said to be a very well-behaved lens*, denoted by $f : A \rhd B$, if* PUTGET *and* GETPUT *hold.*

Since we do not assume any totality constraints *a priori*, these laws simply entail bounds on the behavior of the putback: PUTGET restricts the range of acceptable computations, while GETPUT sets the range of mandatory computations.

The main strength of combinatorial frameworks like the lens frameworks is their ability to build complex transformations from simple ones while preserving "well-behavedness". In this context, the most essential combinator is the sequential *composition* of two very well-behaved lenses $f : A \rhd B$ and $g : B \rhd C$ as $g \circ f : A \rhd C$, defined by

$$\begin{aligned} \mathsf{get}_{g \circ f}\ a &= \mathsf{get}_g\ (\mathsf{get}_f\ a) \\ \mathsf{put}_{g \circ f}\ (a_0, c) &= \mathsf{put}_f\ (a_0, \mathsf{put}_g\ (\mathsf{get}_f\ a_0, c)) \end{aligned}$$

which is also well-behaved, whatever the provided lenses f and g represent.

The notion of consistency in bidirectional mappings and lenses is rather ambiguous. Typically, one of the transformation—the one used to specify the bidirectional transformation—is assumed to represent an implicit consistency relation. In the case of lenses, get_f would play that role, in which case the correctness property degenerates into PUTGET (Pacheco et al., 2013).

Some lens frameworks enforce stricter laws over their transformations. For instance, some frameworks requires lenses to be *undoable* or even *history-ignorant*. The latter is specified by the following law:

$$a \in \mathsf{put}_f\ (a_0, \_) \wedge a' \in \mathsf{put}_f\ (a, b_0) \Rightarrow a' \in \mathsf{put}_f\ (a_0, b_0) \qquad \text{PUTPUT}$$

This law states that the propagation of a source update is independent of the view update history. Lenses for which PUTPUT holds are dubbed *well-behaved lenses*.

Other frameworks (Mu et al., 2004; Liu et al., 2007; Hu et al., 2008; Hidaka et al., 2010) rely instead on looser laws, requiring only "one-and-a-half" round-tripping properties, as the following law:

$$a \in \mathsf{put}_f\ (\_, b_0) \Rightarrow a \in \mathsf{put}_f\ (\mathsf{get}_f\ a, b_0) \qquad \text{PUTGETPUT}$$

Under this law, also known as *weak-acceptability*, source updates are allowed to introduce side-effects of the view state.

Rather than specifying how the forward and backward transformations interact with each other, in constraint maintainers, whose core artifact is a consistency relation, the laws state instead the relationship between it and the transformations. Since constraint maintainers are symmetric, the laws are dual for both consistency-restoring transformations.

$$b \in \overrightarrow{T}\ (a_0, b_0) \Rightarrow T\ (a_0, b)$$
$$a \in \overleftarrow{T}\ (a_0, b_0) \Rightarrow T\ (a, b_0) \qquad \text{CORRECT}$$
$$T\ (a_0, b_0) \Rightarrow b_0 \in \overrightarrow{T}\ (a_0, b_0)$$
$$T\ (a_0, b_0) \Rightarrow a_0 \in \overleftarrow{T}\ (a_0, b_0) \qquad \text{HIPPOCRATIC}$$

CORRECT simply states that transformations are correct if they produce consistent values, while HIPPOCRATIC states that they are hippocratic values that are already consistent are not updated by the transformations. Similarly to the lens round-tripping laws, these laws also entail bounds on the behavior of the transformations: CORRECT restricts the range of acceptable computations, while HIPPOCRATIC imposes mandatory computations.

Definition 3.2 (well-behaved maintainer)**.** *A constraint maintainer T between transformation domains A and B, comprised by the consistency relation $T : A \leftrightarrow B$ and transformations $\overrightarrow{T} : A \times B \rightharpoonup B$ and $\overleftarrow{T} : A \times B \rightharpoonup A$, is said to be a* well-behaved constraint maintainer*, denoted by $T : A \bowtie B$, if* CORRECT *and* HIPPOCRATIC *hold.*

3.2.2 Totality

The round-tripping properties presented above were defined modulo totality. This means that, besides GETPUT and CORRECT forcing null updates to be propagated as null updates, transformations may be undefined everywhere else and still be considered well-behaved.

In fact, some bidirectional transformation frameworks (Matsuda et al., 2007; Voigtländer, 2009; Voigtländer et al., 2010) provide no totality guarantees. Although these languages are designed with sensible transformations, the lack of totality restrictions compromises the predictability of the system, since the transformations may fail at run-time. Thus, some restrictions on the totality of the transformations are typically imposed.

The strongest of these is to guarantee that the bidirectional transformation never fails. In lens frameworks (Foster et al., 2007; Pacheco and Cunha, 2010; Wang et al., 2010), this amount to having get_f and put_f total.

Definition 3.3 (total lens)**.** *A very well-behaved lens* $f : A \rhd B$ *is said to be* total*, denoted by* $f : A \trianglerighteq B$*, if* get_f *and* put_f *are total.*

A direct consequence of having a total lens is that the forward transformation must be surjective: since put_f must be defined for every pair (a, b), PUTGET will force every b to be the view of some source element[4]. This compromises the expressibility of the bidirectional transformation language, since many interesting transformations (like duplication or constant operations) are not surjective. Thus, some frameworks assume instead some weaker totality properties. For instance, a lens is said to be *safe* (Pacheco, 2012) (or domain correct (Diskin, 2008)) if they are defined at least for the values in the range of the opposite transformations. In the case of lenses this results in the following property:

$$b_0 = \mathsf{get}_f\ a_0 \Rightarrow \forall\ a' : A \mid (\exists\ a : A \mid a \in \mathsf{put}_f\ (a', b_0))$$

Regarding constraint maintainer frameworks, CORRECT does not impose any updatability criteria, while HIPPOCRATIC only require the transformations to be defined for input pairs that are already consistent. Nonetheless some approaches require the transformation to be always defined, like the original constraint maintainer proposal (Meertens, 1998).

Definition 3.4 (total constraint maintainer)**.** *A well-behaved constraint maintainer* $T : A \bowtie B$ *is said to be* total*, denoted by* $T : A \trianglerighteq\!\trianglelefteq B$*, if* $\overrightarrow{T}$ *and* $\overleftarrow{T}$ *are total.*

This will force relation T to be total: if the transformations are always able to propagate updates in both directions over a pair (a, b), then, from CORRECT, there is always at least an element consistent with a and an element consistent with b.

Many times the lack of totality stems from the reduced expressibility of the type system, specially in transformations over rich data domains like models, where datatype constraints abound. In fact, the original lens proposal by Foster et al. (2007) is designed over types enhanced with invariants, that allowed the definition of total combinators over the refined datatypes.

[4]The reverse implication is also true: every total and surjective function can be lifted to a total very well-behaved lens (Foster et al., 2007).

3.2.3 Exhaustive Bidirectional Transformations

Multi-valued transformations are well-behaved if the set of produced updates is within that range of valid sources. We introduce the novel notion of *exhaustive* bidirectional transformations, which return *every* acceptable element, in contrast to *selective* ones that select a single element to be returned. In the lens scheme, these can be defined as follows.

Definition 3.5 (exhaustive lens)**.** *A* well-behaved exhaustive lens $f : A \blacktriangleright B$ *consists of a transformation* $\mathsf{get}_f : A \rightharpoonup B$ *and a multi-valued multi-valued* $\mathsf{Put}_f : A \times B \leftrightarrow A$ *such that* PUTGET*,* GETPUT *and the following property hold:*

$$b \notin \mathsf{get}_f\ a_0 \wedge b \in \mathsf{get}_f\ a \Rightarrow a \in \mathsf{Put}_f\ (a_0, b) \qquad \text{EX-PUTGET}$$

First, note that the putback is not longer restricted to be simple (a fact denoted by the upper case Put_f). Law EX-PUTGET states that, for non-null view updates, the putback of an exhaustive lens must return every acceptable source. If instead the view update is null ($\mathsf{get}_f\ a = \mathsf{get}_f\ a'$), the behavior of the putback is unrestricted (except for the case $a \in \mathsf{Put}_f\ (a, \mathsf{get}_f\ a)$, already entailed by GETPUT). This does not affect the totality of the lens since it forces $\mathsf{Put}_f\ (a, \mathsf{get}_f\ a)$ to be defined for every $a : A$. These multi-valued putbacks can be directly implemented, for example using the non-determinism monad in a functional language like Haskell. Since our setting is many-sorted, the range of such exhaustive transformations depends on the valuations assigned to each sort.

Exhaustive totality is defined in the same way as the selective case.

Definition 3.6 (exhaustive total lens)**.** *An exhaustive lens* $f : A \blacktriangleright B$ *is said to be* total*, denoted by* $f : A \trianglerighteq B$*, if* get_f *and* Put_f *are total.*

The notion of exhaustive constraint maintainer is attained in a similar manner: if the two elements are not consistent with each other, then the transformations must return every valid update.

Definition 3.7 (exhaustive constraint maintainer)**.** *A* well-behaved exhaustive constraint maintainer $f : A \blacktriangleright\!\blacktriangleleft B$ *consists of a consistency relation* $T : A \times B$ *and multi-valued transformations* $\overrightarrow{T} : A \times B \leftrightarrow B$ *and* $\overleftarrow{T} : A \times B \leftrightarrow A$*, such that* HIPPOCRATIC*,* CORRECT

and the following property hold:

$$\neg(T\ (a_0, b_0)) \wedge T\ (a_0, b) \Rightarrow b \in \overrightarrow{T}\ (a_0, b_0)$$
$$\neg(T\ (a_0, b_0)) \wedge T\ (a, b_0) \Rightarrow a \in \overleftarrow{T}\ (a_0, b_0) \qquad \text{EX-CORRECT}$$

3.2.4 Disambiguating Updates

In general, the bidirectional transformation laws presented in the previous section allow multiple valid backward transformations, which may affect the predictability of the bidirectional transformation system.

These laws only provide loose bounds on the behavior of the transformations, and should be taken as first principles only: for example, GETPUT "only provides a relatively loose constraint on the behavior of lenses", as originally remarked by the authors of the lens framework (Foster, 2009). To understand the behavior of a lens f, a user cannot rely solely on the laws and must directly inspect the definition of put_f. Therefore, laws providing more guarantees about put_f would provide a better account of its behavior to users: instead of requiring only that the source remains unchanged when the view is unchanged, they could enforce some selection criterion on update translation. The same applies to the constraint maintainer framework, since CORRECT only states that the resulting value m' is consistent. Such selection criterion may be obtained by defining a preference order over the acceptable updates, which could embody, for instance, the notion of minimal update.

Much existing work in the context of database view updating is concerned with, given a view function, deriving an update strategy that translates view updates into source updates according to some minimization criteria. Hegner (2004) introduces a notion of order on sources ($\preceq_A$) and on views ($\preceq_B$) and postulates, among other properties, that view updating shall be order-compatible ($\mathsf{get}_f\ a \preceq_B b \Rightarrow a \preceq_A \mathsf{put}_f\ (a, b)$). His notion of order is only between two values of the transformation domain (hence $\preceq_A$, for a type A), and the above property formalizes that, if an updated view is at most an original view, then the updated source shall be at most the original source. In the context of database tables, his particular orders imply the reflection of view insertions as source insertions, and similarly for deletions. He then establishes that, under particular conditions and for monotonic get_f functions, there is a unique translation of insertion and deletion view updates under a constant complement approach (Bancilhon and Spyratos,

1981). More recently, Johnson et al. (2010) show the connection between the constant complement update translators from (Hegner, 2004) and lenses, by demonstrating that they arise from well-behaved lenses in a category of ordered sets.

These "order-based" lenses impose an absolute order on the elements of the transformation domain, which may not always be sensible. A significantly distinct approach is followed by Meertens (1998), whose notion of order is "triangular", as it relates two updated values in regard to their distance to an original value. This principle of least-change tightens the bounds imposed by the CORRECT law of constraint maintainers, thus making the behavior of the comprising transformations more predictable. Let A be a transformation domain and $\preceq : A \to (A \leftrightarrow A)$ be a family of total preorders that, given an element $a_0 : A$, compare elements relative to their "distance" to a_0[5].

Definition 3.8 (stable preorder)**.** *For any value $a_0 \in A$, a preorder $\preceq_{a_0}$ is said to be* stable *if a_0 is its unique minimal value, i.e.,*

$$a \preceq_{a_0} a_0 \equiv a = a_0 \qquad \text{ORDSTABLE}$$

Condition ORDSTABLE ensures that the element m is the single closest value to itself, i.e., a is the universal lower-bound of $\preceq_a$. Given families of stable preorders $\preceq$ and $\sqsubseteq$ over transformation domains A and B, respectively, least-change in the constraint maintainer framework can then be defined as follows:

$$\begin{aligned} &b \in \overrightarrow{T}\ (a_0, b_0) \Rightarrow T\ (a_0, b) \wedge (\forall\ b' : B \mid T\ (a_0, b') \Rightarrow b \sqsubseteq_{b_0} b') \\ &a \in \overleftarrow{T}\ (a_0, b_0) \Rightarrow T\ (a, b_0) \wedge (\forall\ a' : A \mid T\ (a', b_0) \Rightarrow a \preceq_{a_0} a') \qquad \text{LC-CORRECT} \end{aligned}$$

That is, whatever the value b produced by $\overrightarrow{T}$, it will be at least as close to the original b_0 as any other value b' that is also consistent with a_0 (and dually for $\overleftarrow{T}$). This reduces the upper-bound on the transformations imposed by CORRECT that allowed any consistent solution to be returned.

Definition 3.9 (least-change constraint maintainer)**.** *Given families of stable preorders $\preceq$ and $\sqsubseteq$ over transformation domains A and B, respectively, a constraint maintainer $T : A \bowtie B$ is said to be a least-change constraint maintainer, denoted by $T : A_{\preceq} \bowtie B_{\sqsubseteq}$, if* HIPPOCRATIC *and* LC-CORRECT *hold.*

[5] Anti-symmetry is not required because any two points at the same distance will be comparable in either way and they need not be the same.

In the context of exhaustive bidirectional transformations, the transformations is expected to return every minimal value, rather than a single one.

Definition 3.10 (exhaustive least-change constraint maintainer)**.** *Given families of stable preorders $\preceq$ and $\sqsubseteq$ over transformation domains A and B, respectively, an exhaustive constraint maintainer $T : A_{\bowtie}\ B$ is said to be an exhaustive least-change constraint maintainer, denoted by $T : A_{\preceq} \bowtie B_{\sqsubseteq}$, if* CORRECT *and the following property hold:*

$$T\ (a_0, b) \wedge (\forall\ b' : B \mid T\ (a_0, b') \Rightarrow b \sqsubseteq_{b_0} b') \Rightarrow b \in \overrightarrow{T}\ (a_0, b_0)$$
$$T\ (a, b_0) \wedge (\forall\ a' : A \mid T\ (a', b_0) \Rightarrow a \preceq_{a_0} a') \Rightarrow a \in \overleftarrow{T}\ (a_0, b_0)$$

LC-HIPPOCRATIC

Unlike LC-CORRECT, LC-HIPPOCRATIC tightens the lower-bound on the transformations imposed by HIPPOCRATIC: the latter requires the transformations to be defined for every consistent input pair; the former requires them to be defined whenever there is a valid minimal solution (which degenerates into HIPPOCRATIC if the input is already consistent).

One possible way to construct such family of total preorders is to rely on a distance function $\Delta : A \to A \to \mathbb{N}$ on A values as (Meertens, 1998):

$$a \preceq_{a_0} a' \equiv (\Delta\ a_0\ a)\ \leqslant\ (\Delta\ a_0\ a')$$

where $\leqslant : \mathbb{N} \leftrightarrow \mathbb{N}$ is the standard order on natural numbers[6]. The preorder is stable if the underlying distance is stable, i.e., the closest value to a value m is itself[7]:

$$\Delta\ (a_1, a_2) = 0 \equiv a_1 = a_2 \qquad \text{DISTSTABLE}$$

We shall denote this "lifted" preorder as $[\Delta\ a_0] : A \leftrightarrow A$ and the whole derived family as $[\Delta] : A \to (A \leftrightarrow A)$.

Some approaches address the ambiguity of view-update translation without relying explicit preference orders. Keller (1986) tackles this issue for non-constant complement

[6]Instead of natural numbers any well ordered set could be used in the range of the distance function, providing more flexibility when computing distances. Notice also that if, for a given a_0, $\Delta\ a_0$ is an injective distance function, then the induced preorder $\preceq_{a_0}$ will be anti-symmetric (i.e., a partial order).

[7]In a metric space, this clause is known as *identity of indiscernibles*. In fact, it could be reasonable to assume that (A, Δ) is a metric space, satisfying other properties such as *symmetry* or *triangle inequality*, although these are orthogonal to the properties of least-change bidirectional transformation.

approaches (very well-behaved lenses), and proposes an interactive algorithm that runs a dialog with the view programmer to choose a particular $\mathtt{put}_f$ function that obeys 5 minimization criteria. Larson and Sheth (1991) claim that the view information is not sufficient to disambiguate view updates, and propose to consider not only a dialog with the view programmer but also a dialog with the view user, obeying similar minimization criteria. More recent developments by Pacheco et al. (2014) advocate the best way to disambiguate update propagation is to directly specify the putback.

3.3 Discussion

In this section we intend only to provide a rough overview of the possible bidirectional transformation frameworks. Nonetheless, the variety of bidirectional transformation languages and frameworks that have been developed deem this task incomplete by nature. Abstract frameworks over which the frameworks can be instantiated (Diskin, 2011; Pacheco et al., 2013; Terwilliger et al., 2012) and bidirectional transformation surveys (Czarnecki et al., 2009; Hu et al., 2011) have been proposed to attenuate these gaps. Each of the succeeding chapters provides a more in-depth discussion of the relevant state-of-the-art on bidirectional transformation.

Round-tripping laws were purposely defined modulo totality, as to allow transformations to be partial. If dealing with total transformations, such will be explicitly stated. The definition of the multi-valued exhaustive bidirectional transformations is novel, and will prove to be useful throughout this dissertation, developed in a relational, rather than functional, setting.

Part II

Lens Framework

Chapter 4

Invariant-constrained Lenses

Datatype constraints are a common requirement in data transformation: they play an essential role on improving the predictability and updatability of the system, since without them transformations may fail at run-time for inputs deemed well-formed. For instance, although applying a transformation $\mathsf{tail} : [A] \rightharpoonup [A]$ to an empty list type-checks, it will fail to execute—a fact not directly captured by algebraic type systems like that of Haskell. This predictability problem is even more significant in the bidirectional transformation context. For a lens to be considered total, its forward transformation must be total and surjective, since in order to be propagated backwards by the putback, an updated view must have a matching source element. For this reason, many interesting transformations are not admissible as total well-behaved lenses. The duplication transformation $\mathsf{id} \vartriangle \mathsf{id} : A \to A \times A$ is a paradigmatic example, where the backward transformation is only well-behaved for pairs whose elements are the same, a fact impossible to capture in plain algebraic datatypes and about which the user may be unaware. Since the backward transformations, unlike the forward transformations, emerge from the bidirectionalization procedure through possibly obscure processes, rather than being written by the user, the type system plays an even greater role on managing the expectations of the user.

These issues lead to a well-known tradeoff in the design of bidirectional transformation languages between the expressiveness provided by its syntax and the robustness enforced by the totality and round-tripping laws. Some approaches (Pacheco and Cunha, 2010; Wang et al., 2010) compromise expressiveness and provide a set of combinators that build only total (over unconstrained domains) and surjective transformations: the above duplication operation would simply not be allowed. Others ignore the totality

requirement in order to admit non-surjective transformations (Matsuda et al., 2007; Voigtländer, 2009; Voigtländer et al., 2010), in which case round-tripping laws can be trivially satisfied, for example by making putbacks always undefined for non-null updates. Obviously these languages are designed with sensible backward transformations, but since their behavior is not fully documented the user's intuitions may be severely compromised due to unquantifiable updatability. In this context, a duplication lens has view type $A \times A$ and yet its putback fails for any pair whose elements are different. Preserving totality, it is possible to weaken the round-tripping laws to work modulo equivalence relations, i.e., for an equivalence relation $\sim$ between B views, enforce only

$$a \in \mathsf{put}_f\ (\_, b) \Rightarrow \mathsf{get}_f\ a\ \sim\ b$$

and state that values outside the range of a forward transformation are indistinguishable from values inside that range (Foster et al., 2008). Here, with an equivalence relation over pairs $(a, \_)\ \sim\ (a', \_) \equiv a = a'$, a putback for the duplication combinator that always returns the first element of the pair is regarded as well-behaved:

$$\mathsf{put}_{\mathsf{id}\triangle\mathsf{id}}\ (\_, (a_1, a_2)) = a_1, \mathsf{get}_{\mathsf{id}\triangle\mathsf{id}}\ a_1 = (a_1, a_1), (a_1, a_1)\ \sim\ (a_1, a_2)$$

Yet another option is to relax both totality and round-tripping laws (Mu et al., 2004; Liu et al., 2007; Hu et al., 2008; Hidaka et al., 2010), with the PUTGET law relaxed to an one-and-a-half round-trip variant that permits view side-effects, i.e., enforce only weak-acceptability as enforced by PUTGETPUT. In this case, duplication could be supported by assigning it a putback that coherently selects one element of the updated pair (in the cited frameworks, it would the one most recently updated), breaking PUTGET but not its relaxed version:

$$\mathsf{put}_{\mathsf{id}\triangle\mathsf{id}}\ (\_, (a_1, a_2)) = a_1, \mathsf{get}_{\mathsf{id}\triangle\mathsf{id}}\ a_1 = (a_1, a_1), \mathsf{put}_{\mathsf{id}\triangle\mathsf{id}}\ (\_, (a_1, a_1)) = a_1$$

Enhancing the datatypes with invariants would allow the definition of more refined transformation domains, allowing partial lenses to be perceived as total and well-behaved under those restricted domains. For instance, duplication could be interpreted as a total *invariant-constrained lens* $\mathsf{id}\triangle\mathsf{id}{:}A \trianglerighteq (A \times A)_{\mathsf{same}}$, whose view domain invariant restricts pairs to those with equal elements, i.e., $\langle (a, b) \rangle \in \mathsf{same} \equiv a = b$. This alternative tradeoff was actually the one followed in the regular lens framework (Foster et al., 2007): in order to preserve totality, a powerful semantic type system with invariants was used to specify the exact transformation domains, which allowed the

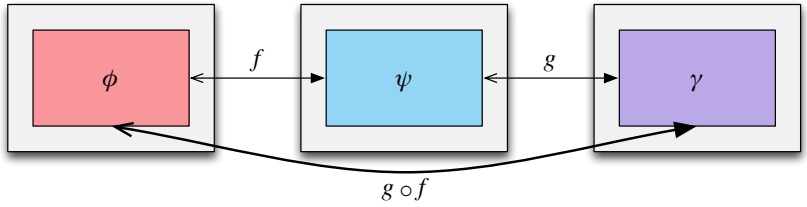
definition of duplication, constants and conditional combinators as total well-behaved lenses.

The ability to introduce datatype constraints could in theory provide an additional benefit in the context of lens frameworks: to allow the user to have a finer control over the behavior of the backward transformations, which are under-specified by the regular round-tripping laws, as already exposed in Section 3.2. In a combinatorial approach, when defining the forward transformation the user is simultaneously defining the behavior of the backward transformation, which was "chosen" when the language was designed from among those deemed well-behaved by the round-tripping laws. For instance, for the $\mathsf{tail} : [A] \rhd [A]$ lens, the assigned putback would typically preserve the original list's head:

$$\mathsf{put}_{\mathsf{tail}}\ ((h_0 \colon \_), t) = (h_0 : t)$$

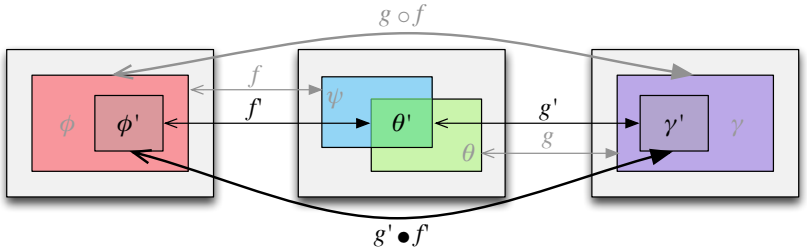
When defined at design time, without knowledge about the execution environment, this seems to be a sensible putback definition. Yet, the notion of sensible greatly varies depending on the context, and given that the round-tripping laws allow an arbitrary element to be attached to the updated tail t, the user may wish to have a finer control over the putback's behavior. Allowing the user to introduce additional constraints on the source domain would grant her such control to a certain degree. For instance, the user could introduce a constraint uniq over the source lists forcing all the elements of a list to be the same. This would have two direct consequences: first, the invariant would need to be propagated to the view domain, which would also be restricted to lists with an unique element; second, $\mathsf{put}_{\mathsf{tail}}$ would have to adapt itself to such invariant in order to attach adequate heads to the updated tail, in this case by attaching at the head of the updated tail t its unique element.

Nonetheless, to preserve the decidability of the type system, the expressiveness of the regular lens framework (Foster et al., 2007) was still restricted by forcing composed lenses to agree not only on types but also on invariants (with the type equality check performed syntactically), as depicted in following diagram:



In such scenario, duplication can, for instance, be followed by a merging operation that only accepts pairs with two equal elements, but not by a generic projection that is defined for whatever pair.

In fact, a more expressive composition is required to allow the definition of such bidirectional transformations. Consider the tentative composition of two invariant-constrained lenses $f : A_\phi \trianglerighteq B_\psi$ and $g : B_\theta \trianglerighteq C_\gamma$, that are total over their constrained domains, but where the set of values accepted by ψ and θ are not exact matches, as depicted in the following diagram:



The first research question of this chapter is precisely: *how may two total well-behaved invariant-constrained lenses over arbitrary invariants be combined into another total well-behaved invariant-constrained lens?*

This cannot be achieved using regular lens composition. To preserve totality, the invariant-constrained composition, which will be denoted by $\bullet$, must be "type-changing", calculating the exact source ϕ' and view γ' domain invariants of the combined transformation to attain a total invariant-constrained lens $g \bullet f : A_{\phi'} \trianglerighteq B_{\gamma'}$. However, restricted transformation domains by themselves do not guarantee well-behavedness nor totality—even for views within γ', the putback of g may still produce values outside ψ, for which the behavior of the putback of f is undetermined. As a concrete example, consider the composition of total invariant-constrained lenses $\mathsf{tail} : [A]_{\mathsf{len}_{[1..[}} \trianglerighteq [A]$ and $\mathsf{length} : [A] \trianglerighteq \mathbb{N}$, where $[A]_{\mathsf{len}_\phi}$ denotes the set of lists whose length belongs to ϕ (e.g., $\mathsf{len}_{[0..10]}$ contains all strings whose length is between 0 and 10).). Using the regular lens composition $\circ$ definition presented in Section 3.2.1, its putback would be defined as:

$$
\begin{aligned}
&l = \mathsf{put}_{\mathsf{length} \circ \mathsf{tail}}\ (l_0, n) \equiv \\
&\quad \textbf{let}\ t = \mathsf{put}_{\mathsf{length}}\ (\mathsf{get}_{\mathsf{tail}}\ l_0, n) \\
&\quad \textbf{in}\ l = \mathsf{put}_{\mathsf{tail}}\ (l_0, t)
\end{aligned}
$$

This composition happens to be well-behaved in the regular lens setting since tail is

surjective. This is perceived in Figure 4.1a: given any updated view, whatever the updated source selected by $\mathsf{put}_{\mathsf{length}}$ will be consumable by $\mathsf{put}_{\mathsf{tail}}$.

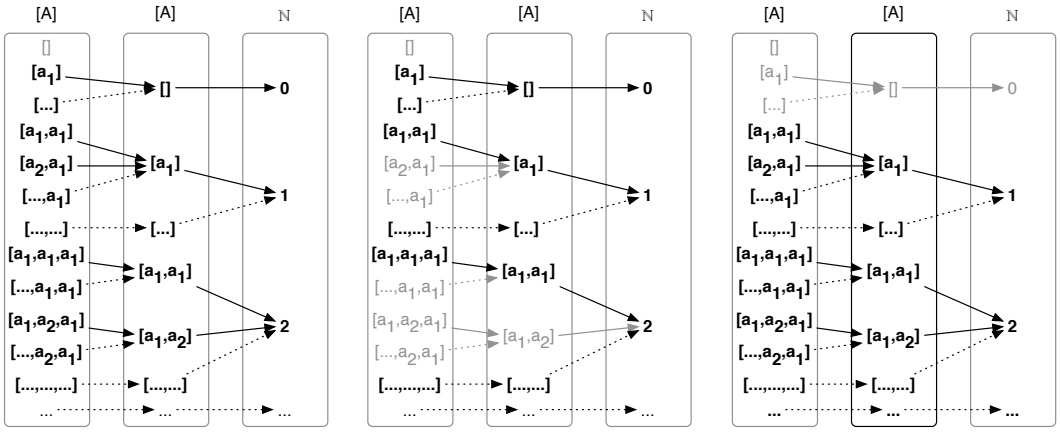
Yet, once user-defined constraints are introduced, that is no longer the case. For instance, by restricting tail to lists whose elements are the same, as $\mathsf{tail} : [A]_{\mathsf{len}_{[1..[} \cap \mathsf{uniq}} \trianglerighteq [A]_{\mathsf{uniq}}$, although the overall transformation $\mathsf{length} \circ \mathsf{tail}$ remains surjective there is no guarantee that the output of $\mathsf{put}_{\mathsf{length}}$ is within uniq and thus consumable by $\mathsf{put}_{\mathsf{tail}}$, as depicted in Figure 4.1b. This could be (naively) solved by an adapted lens composition that simply filters out invalid intermediary elements:

$$
\begin{aligned}
&l = \mathsf{put}_{\mathsf{length} \bullet \mathsf{tail}}\ (l_0, n) \equiv \\
&\quad \mathbf{let}\ t = \mathsf{put}_{\mathsf{length}}\ (\mathsf{get}_{\mathsf{tail}}\ l_0, n) \\
&\quad \mathbf{in}\ l = \mathsf{put}_{\mathsf{tail}}\ (l_0, t) \wedge \langle t \rangle \in \mathsf{uniq}
\end{aligned}
$$

This version does guarantee the well-behavedness of the invariant-constrained lens $\mathsf{length} \bullet \mathsf{tail} : [A]_{\mathsf{len}_{[1..[} \cap \mathsf{uniq}} \rhd \mathbb{N}$, but not its totality, since it is simply discarding the updated tail t produced by $\mathsf{put}_{\mathsf{length}}$ when uniq does not hold. To be total, the putback would have to be given a broader definition that allows the enumeration of all valid updates, so that a tail t within uniq is eventually produced (assuming that the view domain is exact and thus there is at least one valid view update). Concretely, $\mathsf{put}_{\mathsf{length}}$ will be able to generate every list with the given length, as allowed by the round-tripping laws, and eventually produce one whose elements are all the same. These *exhaustive* lenses $f : A \blacktriangleright B$, whose putbacks return sets of sources, allow the preservation of totality in invariant-constrained lenses, given the following putback definition:

$$
\begin{aligned}
&l \in \mathsf{Put}_{\mathsf{length} \bullet \mathsf{tail}}\ (l_0, n) \equiv \\
&\quad \exists\ t : \mathsf{Put}_{\mathsf{length}}\ (\mathsf{get}_{\mathsf{tail}}\ l_0, n) \mid \\
&\qquad l \in \mathsf{Put}_{\mathsf{tail}}\ (l_0, t) \wedge \langle t \rangle \in \mathsf{uniq}
\end{aligned}
$$

This now gives rise to a total well-behaved invariant-constrained lens $\mathsf{length} \bullet \mathsf{tail} : [A]_{\mathsf{len}_{[1..[} \cap \mathsf{uniq}} \blacktriangleright \mathbb{N}$. The putback is now depicted in upper-case to emphasize its multi-valuedness; if it happens to be single-valued, the definition degenerates into the above presented. Other invariant instantiations may not preserve the totality and surjectivity of the overall lens, as the example depicted in Figure 4.1c, where restricting the input lists to have length higher than 1 renders the view 0 invalid (tails may not be empty). These scenarios can be addressed in a similar manner given the ability to calculate exact invariants. Such exhaustive invariant-constrained lenses could also be used to interpret



(a) $\phi = \mathsf{len}_{[1..[}$ and $\psi = \mathbb{N}$. (b) $\phi = \mathsf{len}_{[1..[} \cap \mathsf{uniq}$, $\psi = \mathbb{N}$. (c) $\phi = \mathsf{len}_{[2..[}$, $\psi = [1..[$.

Figure 4.1: Instantiations of invariant-constrained lens $\mathsf{length} \bullet \mathsf{tail} : [A]_{\phi} \blacktriangleright \mathbb{N}_{\psi}$.

other inherently partial combinators, like the fork or conditional combinators, as lenses with maximum updatability. Thus, as already mentioned, extending bidirectional transformations to support datatype invariants is a two-step process: first, transformation domains must be enhanced with adequate invariants that must be propagated through the lenses' transformation domains; second, putback specifications must adapt themselves to the invariants provided in each instantiation.

Executing multi-valued putbacks is expectedly inefficient. First, this kind of framework heavily relies on the (possibly inefficient) evaluation of constraints, as they are required not only to perform type-checking, but also to guarantee the effective execution of well-behaved putbacks. Second, even in a finite universe, one can easily envision the multi-valued putback of length generating an overwhelming number of lists before getting to one within uniq. Yet, standard combinatorial approaches, with the assumption of opaque and static putbacks, establish a clear limitation on the possible optimizations. A syntactic *constraint-aware* bidirectionalization procedure, where the putback is able to effectively adapt itself to the invariants, would enable values to be rejected early in the execution, guaranteeing, for instance, that the putback of length produces only lists within uniq (rather than producing arbitrary lists that are simply discarded by the putback of tail).

This gives rise to this chapter's second research question: *can constraint-aware bidirectionalization procedures be defined that produce selective total well-behaved invariant-constrained lenses over arbitrary invariants?* We try to answer this question

following two different instantiations of the invariant-constrained lens framework. The first derives the putback through syntactic *inversion* of the forward transformation, using the relational calculus as a general-purpose language to represent both the datatype invariants and the transformations. Denoting the artifacts in a unifying language eases their definition and reasoning about their properties. Unlike in the purely combinatorial approach, the shape of the putback is known, a fact that can be exploited to improve the execution of the putbacks. Thus, procedure is also proposed that, through the propagation of invariants down the expression, is able to more efficiently execute putbacks and be selective, if desired. While benefiting from the expressibility power of the calculus of relations, it also suffers from undecidable type-checking and type-inference algorithms, which hinder the applicability of such framework. The second technique derives the selective putback through a constraint-aware *synthesis* procedure, that is well-behaved for the provided invariants. This is explored in the bidirectionalization of a domain-specific language—that of spreadsheet formulas. The main design goals were to provide a bidirectional transformation system that was *intuitive* to the user, by allowing spreadsheet formulas to be run forwards in the standard way or backwards to propagate updates on a formula cell to a set of input cells selected by the user; *conservative*, requiring only a minimal extension to the interface, thus preserving the usability of the system and keeping it predictable to the user; and *transparent*, by allowing the users to inspect (and eventually control) both the invariants and the synthesized backward transformations as spreadsheet formulas themselves. To achieve these goals, the synthesis procedure, given concrete cell constraints, generates well-behaved putbacks that are also written as standard spreadsheet formulas.

The contributions of this chapter, mirroring its structure, are the following:

- we introduce the notion of *selective* and *exhaustive invariant-constrained lens* (Section 4.1) and show that the former can be used to achieve a *combinatorial* lens framework that preserves both well-behavedness and totality, albeit inefficiently;

- we propose a constraint-aware bidirectionalization procedure for invariant-constrained lenses based on arbitrary relational expressions and expression inversion (Section 4.2) providing an expressive and flexible instantiation;

- we propose a constraint-aware bidirectionalization procedure for invariant-constrained lenses designed in a specific environment and based on a synthesis procedure (Section 4.3) providing and efficient and controlled instantiation.

Section 4.4 discusses previously developed related work and overviews the contributions.

4.1 Invariant-constrained Lens Framework

This section introduces the concept invariant-constrained lenses or *ic-lenses*, lenses that are well-behaved over constrained datatypes.

The main idea behind the ic-lens framework is the relaxation of the regular lens laws to only hold *modulo the invariants*: to be considered well-behaved, transformations must still be acceptable and stable, but only concerning values within the constrained domains. For the moment, invariants can be seen as subsets of the transformation domains, closed under typical set operations; in the succeeding sections, these will be instantiated to concrete artifacts and operations.

4.1.1 Selective Invariant-constrained Lenses

In typical lens frameworks, putbacks are single-valued, selecting a single updated source from the range of acceptable ones. Introducing invariants in such selective lenses gives rise to the following definition.

Definition 4.1 (selective ic-lens)**.** *A* well-behaved selective invariant-constrained lens $f : A_\phi \rhd B_\psi$ *consists of transformations* $\mathsf{get}_f : A_\phi \rightharpoonup B_\psi$ *and* $\mathsf{put}_f : A_\phi \times B_\psi \rightharpoonup A_\phi$ *such that the following properties hold:*

$$\langle a_0 \rangle \in \phi \wedge \langle b_0 \rangle \in \psi \wedge a \in \mathsf{put}_f\ (a_0, b_0) \Rightarrow b_0 \in \mathsf{get}_f\ a \qquad \textsc{PutGet-Inv}$$

$$\langle a_0 \rangle \in \phi \wedge b_0 \in \mathsf{get}_f\ a_0 \Rightarrow a_0 \in \mathsf{put}_f\ (a_0, b_0) \qquad \textsc{GetPut-Inv}$$

Recall from Section 2.3 that a binary relation $R : A_\phi \leftrightarrow B_\psi$ produces values in ψ when fed values in ϕ, thus typing the forward transformation as $\mathsf{get}_f : A_\phi \rightharpoonup B_\psi$ forces it to produce views in ψ when fed sources in ϕ (and similarly for the backward transformation). Thus, invariants in an ic-lens can also be seen as pre- and post-conditions of the transformations: if fed values conforming to the invariants, they are guaranteed to produce values that also do; otherwise, their behavior is unknown.

Like regular lenses, arbitrary well-behaved ic-lenses do not provide any updatability guarantees: in the limit case, transformations that are undefined for every non-null

update are considered well-behaved. Thus, some additional totality restrictions are required.

Definition 4.2 (total selective ic-lens)**.** *A selective invariant-constrained lens* $f : A_\phi \rhd B_\psi$ *is said to be* total*, denoted by* $f : A_\phi \trianglerighteq B_\psi$*, if the following properties hold:*

$$\langle a \rangle \in \phi \Rightarrow (\exists\ b' : B \mid b' \in \mathsf{get}_f\ a) \qquad \textsc{GetTotal-Inv}$$

$$\langle a \rangle \in \phi \wedge \langle b \rangle \in \psi \Rightarrow (\exists\ a' : A \mid a' \in \mathsf{put}_f\ (a, b)) \qquad \textsc{PutTotal-Inv}$$

These relaxed totality laws state that an ic-lenses is considered total if its comprising transformations are at least defined for the provided invariants (the transformations may be defined for values outside the invariants, but their behavior over those values is disregarded by the laws). Consequently, while totality and PUTGET in regular lenses entail the surjectivity of the forward transformation (Section 3.2.2), their invariant-constrained version only forces get_f to be surjective over ψ, i.e., $\langle b \rangle \in \psi \Rightarrow (\exists\ a' : A \mid \langle a' \rangle \in \phi \wedge b \in \mathsf{get}_f\ a')$.

One of the strengths of lens frameworks is their combinatorial nature, that enables building complex correct-by-construction bidirectional transformations from simple primitives. However, as already exposed above for the essential composition combinator, while producing ic-lenses that are well-behaved is straight-forward—simply cut elements that fall outside the invariants—preserving the totality of the combined lens when the invariants are not perfect matches is not possible for selective lenses. As already hinted, to obtain a framework of total ic-lenses, one must rely on exhaustive bidirectional transformations.

4.1.2 Exhaustive Invariant-constrained Lenses

To enable the preservation of totality, the putbacks must be given a looser and multi-valued specification that allows the calculation of all acceptable source updates. With these exhaustive putbacks adapted to the invariant-constrained scenario, restricting source updates no longer reduces updatability.

Definition 4.3 (exhaustive ic-lens)**.** *A* well-behaved exhaustive invariant-constrained lens $f : A_\phi \blacktriangleright B_\psi$ *consists of a transformation* $\mathsf{get}_f : A_\phi \rightharpoonup B_\psi$ *and a multi-valued binary relation* $\mathsf{Put}_f : A_\phi \times B_\psi \leftrightarrow A_\phi$ *such that* PUTGET-INV*,* GETPUT-INV *and the*

following property hold:

$$b \notin \mathsf{get}_f\ a_0 \wedge b \in \mathsf{get}_f\ a \wedge \langle a_0 \rangle \in \phi \wedge \langle a \rangle \in \phi \Rightarrow a \in \mathsf{Put}_f\ (a_0, b)\ \text{EX-PUTGET-INV}$$

Law EX-PUTGET-INV forces a behavior similar to that imposed by EX-PUTGET, but only within A elements for which ϕ holds. Again, it does not affect the totality of the lens since, because for every $\langle a \rangle \in \phi$, GETPUT-INV already forces $\mathsf{Put}_f\ (a, \mathsf{get}_f\ a)$ to contain a value in ϕ, (precisely the same a).

These multi-valued putbacks can be still be implemented using the non-determinism monad in a functional language like Haskell, with each produced source update being checked for ϕ membership.

Exhaustive totality is defined in the same way as the selective case.

Definition 4.4 (total exhaustive ic-lens)**.** *An exhaustive ic-lens $f : A_\phi \blacktriangleright B_\psi$ is said to be* total*, denoted by $f : A_\phi \trianglerighteq B_\psi$, if* GETTOTAL-INV *and* PUTTOTAL-INV *hold.*

Unlike in selective combinators, partiality can be controlled in exhaustive invariant-constrained lenses. In fact, this framework has the potential to support non-total and non-surjective transformations, which would render such languages more expressive than most supported by existing lens frameworks.

Regarding the composition of two total exhaustive ic-lenses $g : B_\theta \trianglerighteq C_\gamma$ and $f : A_\phi \trianglerighteq B_\psi$, first there is the need to calculate the exact domain and range of $\mathsf{get}_g \circ \mathsf{get}_f$, denoting the maximum updatability of the combined lens $g \bullet f$. This amounts to finding the set of A sources for which get_f generates values consumable by get_f and the set of C views for which Put_g generates values consumable by Put_f. Essentially, $\mathsf{get}_g \circ \mathsf{get}_f$ is defined for every element a in ϕ such that $\mathsf{get}_f\ a$ is within θ, i.e., $\phi \cap (\theta \circ \mathsf{get}_f)$, while its range consists of c values within γ such that one of its pre-images under get_g is within ψ, i.e., $\gamma \cap (\mathsf{get}_g \circ \psi)$. Under these constraints, both f and g will act within the constrained $B_{\psi \cap \theta}$ and thus, the putback Put_g will be guaranteed to eventually produce sources that are consumable by Put_f in each run. This is achieved with the following transformations:

$$\begin{aligned} \mathsf{get}_{g \bullet f}\ a &= \mathsf{get}_g\ (\mathsf{get}_f \rtimes \theta\ a) \\ a \in \mathsf{Put}_{g \bullet f}\ (a_0, c) &\equiv \exists\ b : (\mathsf{Put}_g \rtimes \psi)\ (\mathsf{get}_f\ a_0, c) \mid a \in \mathsf{Put}_f\ (a_0, b) \end{aligned}$$

These are simply the definitions for the regular lens composition presented in Section 3.2.1, with restricted transformations that filter out values outside the intermediary

invariant $\psi \cap \theta$. The θ restriction on the output of get_f is required because it may produce values in $\psi \setminus \theta$, for which the behavior get_g is unknown. As for the ψ restriction on the output of Put_g is required because, while a c element within $\mathsf{get}_g \circ \psi$ is guaranteed to have at least a pre-image within ψ, it may have others that are not. This gives rise to the following ic-lens combinator:

$$\frac{\Gamma \vdash f : A_{\phi} \blacktriangleright B_{\psi} \qquad \Gamma \vdash g : B_{\theta} \blacktriangleright C_{\gamma}}{\Gamma \vdash g \bullet f : A_{\phi \cap (\theta \circ \mathsf{get}_f)} \blacktriangleright C_{\gamma \cap (\mathsf{get}_g \circ \psi)}}$$

If ψ and θ are disjoint, the source domain $\phi \cap (\theta \circ \mathsf{get}_f)$ will be empty, rendering the ic-lens trivially well-behaved (and total).

Exhaustive primitive combinators can be easily defined. For instance, a total exhaustive invariant-constrained lens for the projections can be defined as

$$\frac{}{\Gamma \vdash \boldsymbol{\pi}_1 : (A \times B)_{\phi} \blacktriangleright A_{\psi}}$$

given a putback that simply returns every acceptable pair:

$$\begin{aligned} \mathsf{get}_{\boldsymbol{\pi}_1}\ (a, b) &= (\pi_1 \rtimes \psi)\ (a, b) \\ \mathsf{Put}_{\boldsymbol{\pi}_1}\ ((a_0, b_0), a) &= a \times (\phi \gg a) \end{aligned}$$

For a predicate ϕ over pairs $A \times B$ and an element $a : A$, $\phi \gg a$ partially evaluates ϕ over a, resulting in a predicate over B containing all b elements related to a by ϕ.

The conditional combinator $\omega \,?\, f : g$ between two exhaustive ic-lenses $f : A_{\phi} \blacktriangleright B_{\psi}$ and $g : A_{\theta} \blacktriangleright B_{\gamma}$ may also be defined under this setting. In this context, both Put_f and Put_f can propagate a b view update backwards if it is within its range, as long as the produced value is within ω and $A \setminus \omega$, respectively. As for the source domain, any a element within the source invariant of f or g for which ω or $A \setminus \omega$ hold, respectively, can be consumed by the ic-lens. This gives rise to the following combinator:

$$\frac{\Gamma \vdash f : A_{\phi} \blacktriangleright B_{\psi} \qquad \Gamma \vdash g : A_{\theta} \blacktriangleright B_{\gamma}}{\Gamma \vdash \omega \,\mathbf{?}\, f : g : A_{(\phi \cap \omega) \cup (\theta \setminus \omega)} \blacktriangleright B_{(\psi \cap (\mathsf{get}_f \circ \omega)) \cup (\gamma \cap (\mathsf{get}_g \circ (A \setminus \omega)))}}$$

The respective putback must simply preserve the source when the view update is null, applying both Put_f and Put_g otherwise:

$$\begin{aligned} \mathsf{get}_{\omega ? f : g}\ a &= \omega \,?\, f : g\ a \\ \mathsf{Put}_{\omega ? f : g}\ (a_0, b) &= \end{aligned}$$

$$\begin{array}{ll}
\textbf{if } \mathsf{get}_f\ a_0 = b \wedge \langle a_0 \rangle \in \omega \textbf{ then} & (\mathsf{Put}_f \gg \omega)\ (a_0, b) \\
\textbf{else if } \mathsf{get}_g\ a_0 = b \wedge \langle a_0 \rangle \notin \omega \textbf{ then} & (\mathsf{Put}_g \gg (A \setminus \omega))\ (a_0, b) \\
\textbf{else} & (\mathsf{Put}_f \gg \omega \cup \mathsf{Put}_g \gg (A \setminus \omega))\ (a_0, b)
\end{array}$$

Exhaustive invariant-constrained lenses can be converted back to a selective version through *biased selectors* (Meertens, 1998). Given a constraint ϕ and a "target" a (not necessarily in ϕ) , operation $a \S \phi$ generates a "repaired" value that satisfies constraint ϕ. The only restriction is that already valid values shall be preserved:

$$\forall\ a : \phi \mid a \S \phi = a \qquad \S\text{-STABLE}$$

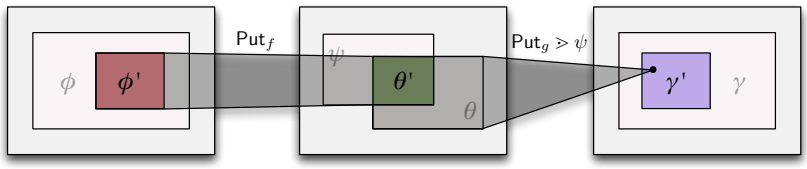
The biased selector allows the selection of a value from among a set that approximates a target. At this stage it can be seen as an abstract placeholder that could allow the user to control the selection of updates. It can be used to convert an exhaustive ic-lens $f : A_\phi \blacktriangleright B_\psi$ into a selective version, by having its putback defined as:

$$\mathsf{put}_f\ (a_0, b) = a_0 \S (\mathsf{Put}_f\ (a_0, b))$$

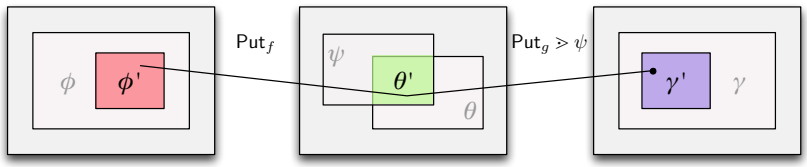
Since Put_f returns a set of values, the biased selector can be used to select a single solution.

4.1.3 Constraint-Aware Frameworks

In Section 4.1.2 a purely combinatorial framework of exhaustive invariant-constrained lenses, where the totality of the sub-lenses is preserved by the invariant-constrained combinators, was proposed. However, as might be expected, the deployment of such framework would be extremely inefficient: higher-order combinators, being given opaque and *constraint-oblivious* sub-lenses, are confined to filter out unwanted outputs. This amounts roughly to the procedure depicted in Figure 4.2a, where Put_g must produce every valid source just to have those outside ψ filtered out. A different bidirectionalization approach would be to define *constraint-aware* putbacks, that are able to adapt themselves to provided invariants, highly improving the efficiency their performance: such putback would now fall in the scenario depicted in Figure 4.2b, where a restricted set of values is produced by Put_g that is guaranteed to be within ψ. Essentially, this amounts to designing a specialized execution procedure for constrained putbacks $\mathsf{Put}_l \gg \phi$ for which the invariant-constrained lens laws still hold. Under



(a) Constraint-oblivious approaches.



(b) Constraint-aware approaches.

Figure 4.2: Approaches to totality preservation in ic-lenses.

this setting totality may be preserved under selective transformations: while in the combinatorial setting, a constraint-oblivious selective putback for g would select an arbitrary value from within θ, which may fall outside ψ (Section 4.1.1), for constraint-aware transformations the selection is guaranteed to fall within ψ and thus produces valid outputs.

The evaluation of datatype invariants is a cornerstone in the ic-lens framework, since every execution of an ic-lens $f : A_\phi \blacktriangleright B_\phi$ requires type-checking the input values—i.e., evaluating ϕ and ψ for input source and view elements, respectively. Furthermore, in constraint-aware frameworks, invariants are also fundamental in the derivation and execution of putbacks. Thus, to deploy such frameworks, concrete instantiations for the invariant language and the invariant operations used throughout the previous section must be carefully designed so that the resulting language is manageable. Namely, if bidirectional totality is expected to hold, the following operations must be given effective instantiations:

membership test $\langle a \rangle \in \phi$ the essential operation, required to both type-check transformations and allow the putback to filter out undesired values;

constrained domain $\psi \circ R$ **and range** $R \circ \phi$ required to support totality-preserving lens combinators;

invariant manipulation $\phi \cap \psi$**,** $\phi \cup \psi$**,** $\phi \setminus \psi$**, ...** the set of supported manipulation operations directly affects the expressibility of the supported language;

$$
\begin{array}{ll}
\mathsf{id} & : A \to A \\
\cdot \circ \cdot & : (A \rightharpoonup B) \to (B \rightharpoonup C) \to (A \rightharpoonup C) \\
\cdot \vartriangle \cdot & : (A \rightharpoonup B) \to (A \rightharpoonup C) \to (A \rightharpoonup B \times C) \\
\pi_1 & : A \times B \to A \\
\pi_2 & : A \times B \to B \\
\underline{b} & : A \to B \\
\cdot \,?\, \cdot : \cdot & : (A \rightharpoonup A) \to (A \rightharpoonup B) \to (A \rightharpoonup B) \to (A \rightharpoonup B)
\end{array}
$$

Figure 4.3: Sample transformation language.

partial evaluation of product invariants $\phi \gg a$ **and** $\phi \ll a$ required to support totality-preserving product combinators;

biased selector $a \S \phi$ required to define useful selective primitives.

The following sections present two contrasting approaches to such instantiation: one expects invariants to be defined in the same language of the transformations, resulting in an expressive general-purpose language, at the cost of an unquantifiable complexity; the other, designed in a controlled environment, forgoes expressibility by restricting the invariants to a normalized shape, over which the selected operations are guaranteed to be efficient. These frameworks are instantiations of ic-lenses (Definitions 4.1 and 4.3) but their dedicated putback derivation and execution procedures allow us to abandon the exhaustive combinators from Section 4.1.2.

4.2 Relational Framework

In this setting, both invariants and transformations are arbitrary relational expressions, defined as standard binary relational expressions, following a syntax similar to the one used throughout this dissertation (Figure 2.1). This gives rise to an expressive general-purpose bidirectional transformation language. For the purpose of this presentation, the forward transformation will be specified using the simple (or simplicity-preserving) combinators depicted in Figure 4.3. By supporting combinators like forks and conditionals, our language becomes more expressive than most existing lens frameworks.

This syntactic bidirectionalization approach is split into two phases: first, a relational expression representing an exhaustive putback is derived from the forward transformation through formula inversion; second, an execution procedure is defined that takes

Algorithm 1: Syntactic bidirectionalization through formula inversion.

input : forward transformation $f : A_\phi \rightharpoonup B_\psi$
output : total invariant-constrained lens $\lfloor f \rfloor : A_{\phi \cap \delta f} \blacktriangleright B_{\rho(f \circ \phi)}$
$\mathsf{get}_f \leftarrow f$;
$\mathsf{Put}_f \leftarrow [f] \,?\, \pi_1 : (f^\circ \rhd \phi \circ \pi_2)$;

advantage of the shape of the putback and the known invariants to improve performance. The first stage is rather straight-forward to define, as depicted in Algorithm 1, and results in the derivation of an invariant-constrained lens $\lfloor f \rfloor : A_{\phi \cap \delta f} \blacktriangleright B_{\rho(f \circ \phi)}$ from a transformation $f : A_\phi \rightharpoonup B_\psi$. Let $[f]$ define the invariant $\langle a, b \rangle \in [f] \equiv b \in f\ a$ (explored shortly). Then, the derived Put_f trivially satisfies the round-tripping laws, since $[f]$ explicitly tests whether the view was updated. If so, it returns the original source; otherwise, it runs f backwards to recover all possible sources that could have originated that view. The resulting invariant-constrained lens is total and well-behaved over sources in ϕ that are also in the domain of f ($\phi \cap \delta f$) and views that result from the application of f to ϕ sources ($\rho(f \circ \phi)$), which by definition of $f : A_\phi \rightharpoonup B_\psi$, is smaller than ψ. The remainder of this section focuses on the second phase, presenting effective means to calculate the invariants and execute the putbacks.

4.2.1 Relational Invariant Language

In the calculus of binary relations, invariants take the shape of coreflexives. Recall that coreflexives are relational expressions that either return the input value or fail, and thus act as filters. Recall from Section 2.4 that sets and coreflexives are isomorphic, and thus we freely switch between the set ($\phi, \psi, \gamma, \theta, \ldots : \mathbb{P}\ A$) and coreflexive ($\Phi, \Psi, \Gamma, \Theta, \ldots : A \rightharpoonup A$) representations.

Language Using coreflexives to represent invariants, rather than arbitrary sets, is beneficial because it allows us to define invariants in the same language in which the transformations are written—useful, since we hope to derive invariants from transformations and have the latter adapt themselves to the former. In this context, restricted execution $R \rhd \phi$ amounts to execute $\Phi \circ R$, since coreflexives filter out unwanted elements. This calculus of invariants is inspired by the one proposed by Oliveira (2009), where typing a relation as $R : A_\phi \rightarrow B_\psi$ denotes its pre- and post-conditions, i.e., $R \circ \Phi \subseteq \Psi \circ R$, or $\rho(R \circ \Phi) \subseteq \Psi$, much like the transformations comprising partial

ic-lenses. It follows that all typing rules from (Oliveira, 2009) are applicable to our framework. The round-tripping laws of the ic-lens framework can in this context be simply stated as:

$$\mathsf{Put}_f \circ (\Phi \times \Psi) \subseteq \mathsf{get}_f^\circ \circ \pi_2 \qquad \text{PUTGET-INV}$$

$$(\mathsf{get}_f \vartriangle \mathsf{id})^\circ \circ \Phi \subseteq \mathsf{Put}_f \qquad \text{GETPUT-INV}$$

A coreflexive on products $A \times B$ can always be specified by a relation between the elements of pairs $A \times B$. Any binary relation $R : A \leftrightarrow B$ can be converted to a normalized coreflexive $[R] : A \times B \rightharpoonup A \times B$ defined as $[R] = (\pi_2^\circ \circ R \circ \pi_1) \cap \mathsf{id}$, meaning that $\langle (a, b), (a, b) \rangle \in [R] \equiv \langle a, b \rangle \in R$. Another way to put it is to say that $[R]$ is the largest coreflexive Φ such that $\pi_2 \circ \Phi \subseteq R \circ \pi_1$, since $\Phi \subseteq [R] \equiv (\pi_2 \circ \Phi \circ \pi_1^\circ) \subseteq R$. For example, using this combinator the same predicate stating that both components of a pair are equal can be specified as $[\mathsf{id}] : A \times A \rightharpoonup A \times A$. Given coreflexives $\Phi : A \rightharpoonup A$ and $\Psi : B \rightharpoonup B$, their product is the coreflexive $\Phi \times \Psi : A \times B \rightharpoonup A \times B$ that holds for pairs whose elements are independent of each other, as long as the left element satisfies Φ and the right element satisfies Ψ. It can alternatively be specified as $\Phi \times \Psi = [\Psi \circ \top \circ \Phi]$.

From this definition many interesting properties of this relation may be derived, such as $[\top] = \mathsf{id}$, $[\bot] = \bot$, the cancellation rules $\pi_1 \circ [R] = (\mathsf{id} \vartriangle R)^\circ$ and $\pi_2 \circ [R] = (R^\circ \vartriangle \mathsf{id})^\circ$, $[\overline{R}] = \overline{[R]} \cap \mathsf{id}$ and $[\pi_2 \circ \Phi \circ \pi_1^\circ] = \Phi$ for any coreflexive on pairs Φ. Since lifting of the relation R between A and B elements is at the core of this invariant, their partial evaluation can be simply defined as $[R] \gg a = R\ a$ and $[R] \ll b = R^\circ\ b$.

Coreflexives on sums $A + B$ are considerably simpler, since predicates on sums can always be specified using the sum combinator (each sum has either an A or a B element, thus the invariant may not relate them). Given coreflexives $\Phi : A \rightharpoonup A$ and $\Psi : B \rightharpoonup B$, their sum is the coreflexive $\Phi + \Psi : A + B \rightharpoonup A + B$ that holds for left values that satisfy Φ and right values that satisfy Ψ.

Operations Membership test $\langle a \rangle \in \phi$ amounts to evaluating the expression $\langle a, a \rangle \in \Phi$, which is typically an efficient procedure following the definitions from Figure 2.5, with one exception: the occurrence in Φ of compositions $R \circ S$ where the first relation S is not simple yields undecidable algorithms, since they give rise to existential quantification tests. In this context, evaluating an expression $\langle a, a \rangle \in R \circ S$ involves S enumerating all valid outputs until R is able to process one; if no such element exists, the procedure may not even terminate with infinite sorts. To improve efficiency, the

$$\delta(f \circ g) = \delta(\delta f \circ g) \qquad \rho(f \circ g) = \rho(f \circ \rho g)$$
$$\delta\mathsf{id} = \mathsf{id} \qquad \rho\mathsf{id} = \mathsf{id}$$
$$\delta\underline{b} = \mathsf{id} \qquad \rho\underline{b} = \underline{b} \cap \mathsf{id}$$
$$\delta(f \vartriangle g) = \delta f \cap \delta g \qquad \rho(f \vartriangle g) = [g \circ f^\circ]$$
$$\delta\pi_1 = \delta\pi_2 = \mathsf{id} \qquad \rho\pi_1 = \rho\pi_2 = \mathsf{id}$$
$$\delta(\Theta\ ?\ f : g) = (\delta g \cap \Theta) \cup (\delta f \cap \overline{\Theta}) \qquad \rho(\Theta\ ?\ f : g) = \rho(f \circ \Theta) \cup \rho(g \circ (\mathsf{id} \setminus \Theta))$$

Figure 4.4: Domain and range of unrestricted expressions.

relational expressions are simplified using a rewrite system similar to one previously developed for the optimization of point-free functional expressions (Cunha and Visser, 2011; Pacheco and Cunha, 2011). Essentially, this rewrite system applies some of the point-free laws from Appendix A as unidirectional rewrite rules that eliminate problematic compositions. If the final expression still contains some of those, the implementation can issue a warning informing that its usage as an invariant checker may not be feasible.

Invariant manipulation operations are trivially implemented because the union, intersection and difference of coreflexive expressions is also coreflexive, thus $\phi \cap \psi = \Phi \cap \Psi$, $\phi \cup \psi = \Phi \cup \Psi$ and $\phi \setminus \psi = \Phi \setminus \Psi$.

Regarding the constrained domains and ranges of expressions, $\psi \circ R$ amounts to $\delta(\Psi \circ R)$—the domain of R when outputs are filtered through Ψ—and $R \circ \phi$ to $\rho(R \circ \Phi)$—the range of R when inputs are filtered through Φ. Although the default definitions for domain ($\delta f = (f^\circ \circ f) \cap \mathsf{id}$) and range ($\rho f = (f \circ f^\circ) \cap \mathsf{id}$) could be directly applied and evaluated following again the semantics of Figure 2.4, we propose a dedicated derivation algorithm for the supported language that avoids the insertion of problematic combinators. The result is depicted in Figures 4.4 to 4.6, defined by induction for the sample language defined in Figure 4.3. To avoid infinite reductions in compositions, the laws from Figures 4.5 and 4.6 should be prioritized, which detail how the domain and range of restricted transformations is calculated. These also arise from the calculation of domain and range of unrestricted compositions (e.g., as shown in Figure 4.4, the domain of a composition $R \circ S$ is defined as the domain of $\delta R \circ S$; this matches the pattern $\Phi \circ S$, processed by the rules in Figure 4.5).

The expressions resulting from these definitions are more amenable for execution (and consequently, type-checking) than the default domain and range definitions—although they are derived from them using relational calculus—because most of the

$$\delta(\Phi \circ \mathsf{id}) = \Phi$$
$$\delta(\Phi \circ \underline{b}) = \begin{cases} \mathsf{id} & \text{if } \langle b, b\rangle \in \Phi \\ \bot & \text{otherwise} \end{cases}$$
$$\delta([R] \circ (f \vartriangle g)) = (\delta g \circ f^\circ \circ R^\circ \circ g \circ \delta f) \cap \mathsf{id}$$
$$\delta(\Phi \circ \pi_1) = \Phi \times \mathsf{id}$$
$$\delta(\Phi \circ \pi_2) = \mathsf{id} \times \Phi$$
$$\delta(\Phi \circ (\Theta \mathbin{?} f : g)) = (\delta(\Phi \circ f) \cap \Theta) \cup (\delta(\Phi \circ g) \setminus \Theta)$$

Figure 4.5: Domain of restricted expressions.

$$\rho(\mathsf{id} \circ \Phi) = \Phi$$
$$\rho(\underline{b} \circ \Phi) = \begin{cases} \bot & \text{if } \Phi = \bot \\ \underline{b} \circ \underline{b}^\circ & \text{if } \Phi \neq \bot \\ \underline{b} \circ \Phi \circ \underline{b}^\circ & \text{otherwise} \end{cases}$$
$$\rho((f \vartriangle g) \circ \Phi) = [f \circ \Phi \circ g^\circ]$$
$$\rho(\pi_1 \circ [R]) = \delta R$$
$$\rho(\pi_2 \circ [R]) = \rho R$$
$$\rho((\Theta \mathbin{?} f : g) \circ \Phi) = \rho(f \circ (\Phi \cap \Theta)) \cup \rho(g \circ (\Phi \setminus \Theta))$$

Figure 4.6: Range of restricted expressions.

compositions are eliminated. The domain of fork expressions falls in the special case $f^\circ \circ R \circ g$, which can be efficiently evaluated (in fact, if f and g are total it can simply be evaluated as $a \in (f^\circ \circ R \circ g)\ b \equiv (f\ a) \in R\ (g\ b)$ (Oliveira, 2007)). Thus, type-checking shall be decidable except for particular ranges of fork combinators—those where $\langle a, b\rangle \in \mathsf{get}_f \circ \mathsf{get}_g^\circ$ is undecidable due to the entailed existential quantification.

After applying the laws of Figures 4.4 to 4.6, the resulting expression is further simplified by the rewrite system already mentioned. The rewrite system is also used to perform the equality test $\Phi = \bot$ that occurs in the range of the constant combinator in Figure 4.6. However, such test may not be decidable and the rewrite system may not be able to reduce into $\bot$ an expression that is semantically equivalent to $\bot$. If it cannot be shown that Φ is effectively empty, the default definitions of range and domain are applied instead. Still, for some cases when it can be shown that $\Phi \neq \bot$, allowing the range definition to be further simplified.

4.2.2 Executing Constrained Relational Expressions

In the context of calculus of binary relations, executing an expression $R \rhd \phi$ amount to simply executing $\Phi \circ R$. This section explores how such execution can be operationalized in an efficient way.

Executing binary relations We have shown in Section 2.3 that a binary relation R can be trivially given alternative semantics as a multi-valued function $[\![R]\!]$, through a highly inefficient and impractical procedure. Figure 4.7 presents an alternative optimized definition that avoids the exhaustive search over the universe for a large set of combinators that can be used to build putbacks. This semantics can still be directly implemented in standard functional languages using the non-determinism monad. However, given a value a, even if we are only interested in one of the results of $[\![R]\!]\ a$, there are still several concerns for efficiency (besides the inverse) that may lead to infinite runs without returning a single value, rendering such definition impractical. For example, in the left-biased implementation of intersection we still need to iterate over all results of R until a suitable value that also satisfies S is found. However, this results in an exhaustive constraint-oblivious procedure as ineffective as those from the exhaustive combinatorial approach: the combinators comprising R may be generating elements that will simply be disregarded by Φ.

Consider as an example the expression $(\mathsf{id} \vartriangle \mathsf{id})^\circ \circ (\mathsf{length}^\circ \vartriangle \mathsf{head}^\circ) : \mathbb{N} \leftrightarrow [\mathbb{N}]$, that given a natural n calculates lists with length n whose first element is also n. Multi-valued operations head° and length° generate all lists with n at its head and all lists with length n, respectively; $(\mathsf{id} \vartriangle \mathsf{id})^\circ$ represents the merging operation, defined as the inverse of the duplication, a partial function that takes as input pairs with two copies of the same element, and returns such element. In an unbounded execution following the semantics from Figure 4.7, length° and head° execute freely until they both return the same list, so that it can be consumed by $(\mathsf{id} \vartriangle \mathsf{id})^\circ$. Such execution may not even terminate, since, for instance, head° could be generating all possible lists by increasing length before iterating through their content (including the elements at their heads). The problem persists even when assuming that the user is expecting a single return value, and relying on lazy evaluation.

Executing constrained binary relations The most problematic component when executing the putback derived from Algorithm 1 is the expression $f^\circ \rhd \phi$, which

$$
\begin{array}{lll}
\llbracket R \circ S \rrbracket\ a & = \{\, b \mid c \leftarrow \llbracket S \rrbracket\ a, b \leftarrow \llbracket R \rrbracket\ c \,\} & \llbracket \mathsf{id} \rrbracket\ a = \{\, a \,\} \\
\llbracket R \cap S \rrbracket\ a & = \{\, b \mid b \leftarrow \llbracket R \rrbracket\ a, [\![\langle a, b \rangle \in S]\!] \,\} & \llbracket \pi_1 \rrbracket\ (a, b) = \{\, a \,\} \\
\llbracket R \cup S \rrbracket\ a & = \llbracket R \rrbracket\ a \cup \llbracket S \rrbracket\ a & \llbracket \pi_2 \rrbracket\ (a, b) = \{\, b \,\} \\
\llbracket R \vartriangle S \rrbracket\ a & = \{\, (b, c) \mid b \leftarrow \llbracket R \rrbracket\ a, c \leftarrow \llbracket S \rrbracket\ a \,\} & \llbracket i_1 \rrbracket\ a = \{\, i_1\ a \,\} \\
\llbracket R \triangledown S \rrbracket\ (i_1\ b) & = \llbracket R \rrbracket\ b & \llbracket i_2 \rrbracket\ b = \{\, i_2\ b \,\} \\
\llbracket R \triangledown S \rrbracket\ (i_2\ c) & = \llbracket S \rrbracket\ c & \llbracket \underline{b} \rrbracket\ a = \{\, b \,\} \\
\llbracket R^{\mathrm{o}} \rrbracket\ b & = \{\, a \mid a \leftarrow A, [\![\langle b, a \rangle \in R^{\mathrm{o}}]\!] \,\} & \llbracket \bot \rrbracket\ a = \{\,\} \\
\llbracket \Theta? \rrbracket\ a & = \{\, i_1\ a' \mid a' \leftarrow \llbracket \Theta \rrbracket\ a \,\} \cup \{\, i_2\ a' \mid a' \leftarrow \llbracket A \setminus \Theta \rrbracket\ a \,\} &
\end{array}
$$

Figure 4.7: Unconstrained execution of binary relations.

contains the inverted transformation. Executing such expression amounts to executing $\Phi \circ f^{\circ}$, which could be done through the definitions from Figure 4.7. Yet, equipped with the invariant ϕ over outputs, such execution semantics can be optimized by propagating the invariants over the outer expressions down to the inner expressions, in order to avoid backtracking and the computation of intermediate values that are valid for sub-expressions but rejected by the global expression. In this case, due to the domain of $(\mathsf{id} \vartriangle \mathsf{id})^{\circ}$, $\mathsf{length}^{\circ} \vartriangle \mathsf{head}^{\circ}$ will be aware that length° and head° must generate the same list; this information may in turn be propagated down to length° and head°, narrowing their executions. In particular, given an input n, either the values generated by length° can be restricted to lists with head n or, dually, those produced by head° to lists with length n. This will result in an efficient and complete (in the sense that all acceptable values will eventually be produced if desired) multi-valued execution.

Figure 4.8 shows how this propagation can be performed. For instance, in the evaluation of $R \circ S$ we can narrow the evaluation of S to return only values in the domain of R (and vice-versa), thus avoiding generation of values that would be discarded by R; since the fork $R \vartriangle S$ is only defined for values in the domain of both R and S, the source invariant of each branch is restricted by the source invariant of the other. The inverse of expressions is presented in Figure 4.9, where each case is analyzed individually to achieve better efficiency. We omit the evaluation of the inverse of idem-potent combinators ($\mathsf{id}, \Theta, \top, \bot$) and of combinators whose inverse can be easily distributed ($R \circ S, R \cap S, R \cup S$) and thus can be executed by the definitions in Figure 4.8.

The evaluation of the primitive combinators is, for most cases, fairly obvious, since it consists in their standard definition, with a membership test for the desired invariant. It is important to note that all invariant tests occur at the primitives, meaning that

$$
\begin{array}{ll}
\llbracket \Phi \circ \mathsf{id} \rrbracket\ a & = \llbracket \Phi \rrbracket\ a \\
\llbracket \Phi \circ ((R \circ S)) \rrbracket\ a & = \{\, c \mid b \leftarrow \llbracket \delta(\Phi \circ R) \circ S \rrbracket\ a, \\
 & \qquad\quad c \leftarrow \llbracket \Phi \circ R \rrbracket\ b \,\} \\
\llbracket \Phi \circ (R \cap S) \rrbracket\ a & = \{\, b \mid b \leftarrow \llbracket \delta S \lessdot R \gtrdot \rho(\Phi \circ S \circ \underline{a}) \rrbracket\ a \,\} \\
\llbracket \Phi \circ (R \cup S) \rrbracket\ a & = \llbracket \Phi \circ R \rrbracket\ a \cup \llbracket \Phi \circ S \rrbracket\ a \\
\llbracket [U] \circ (R \vartriangle S) \rrbracket\ a & = \{\, (b, c) \mid b \leftarrow \llbracket \delta S \lessdot R \gtrdot \rho(U^\circ \circ S \circ \underline{a}) \rrbracket\ a, \\
 & \qquad\quad\ c \leftarrow \llbracket \delta R \lessdot S \gtrdot \rho(U \circ \underline{b}) \rrbracket\ a \,\} \\
\llbracket \Phi \circ \pi_1 \rrbracket\ (a, b) & = \llbracket \Phi \rrbracket\ a \\
\llbracket \Phi \circ \pi_2 \rrbracket\ (a, b) & = \llbracket \Phi \rrbracket\ b \\
\llbracket \Phi \circ (R \triangledown S) \rrbracket\ (i_1\ a) & = \llbracket \Phi \circ R \rrbracket\ a \\
\llbracket \Phi \circ (R \triangledown S) \rrbracket\ (i_2\ b) & = \llbracket \Phi \circ R \rrbracket\ b \\
\llbracket (\Phi + \Psi) \circ i_1 \rrbracket\ a & = \{\, i_1\ a' \mid a' \leftarrow \llbracket \Phi \rrbracket\ a \,\} \\
\llbracket (\Phi + \Psi) \circ i_2 \rrbracket\ b & = \{\, i_2\ b' \mid b' \leftarrow \llbracket \Psi \rrbracket\ b \,\} \\
\llbracket (\Phi + \Psi) \circ (\Theta?) \rrbracket & = \{\, i_1\ a' \mid a' \leftarrow \llbracket \Theta \rrbracket\ a, [\![\langle a', a' \rangle \in \Phi]\!] \,\} \cup \\
 & \quad \{\, i_2\ a' \mid a' \leftarrow \llbracket A \setminus \Theta \rrbracket\ a, [\![\langle a', a' \rangle \in \Psi]\!] \,\} \\
\llbracket \Phi \circ \bot \rrbracket\ a & = \{\,\} \\
\llbracket \Phi \circ \top \rrbracket\ a & = \{\, b \mid b \leftarrow B, [\![\langle b, b \rangle \in \Phi]\!] \,\} \\
\llbracket \Phi \circ \underline{b} \rrbracket\ a & = \llbracket \Phi \rrbracket\ b
\end{array}
$$

Figure 4.8: Constrained execution of binary relations.

infeasible values are never passed through higher-order combinators. Nevertheless, redundant values can still be generated, even if they produce a valid output. For instance, in an expression $\underline{b} \circ \top$, where b is a valid output, $\top$ will generate all possible values, even though they will all be transformed into the same acceptable value by $\underline{b}$. In many of such cases, the rewrite system already presented can be used to remove redundant value generation—in this case, $\underline{b} \circ \top$ would be reduced to $\underline{b}$.

The most interesting narrowing cases are those of the intersection and the inverse of fork (which is actually also an intersection). For these cases, with the default definition from Figure 4.7, the R branch would execute independently of the invariants of S and its output would be tested in S. Naturally, the unconstrained evaluation of R can be very inefficient and may process and generate infeasible values that are not in the domain or range invariant of S, respectively. Using invariants, we restrict R to the domain of S and constrain the values generated by R to only those that would also be produced by S. For instance, in the execution of the inverse of the fork $\llbracket (R \vartriangle S)^\circ \rrbracket\ (b, c)$, instead

$$
\begin{aligned}
&\llbracket \Phi \circ \underline{b}^{\circ} \rrbracket\ b && = \{\, a \mid a \leftarrow A, [\![\langle a, a\rangle \in \Phi]\!] \,\} \\
&\llbracket \Phi \circ (R \vartriangle S)^{\circ} \rrbracket\ (a, b) && = \{\, c \mid c \leftarrow \llbracket \rho(\psi \circ S^{\circ} \circ \underline{b}) \circ R^{\circ} \rrbracket\ a \,\} \\
&\llbracket [U] \circ \pi_1^{\circ} \rrbracket\ a && = \{\, (a, b) \mid b \leftarrow \llbracket\ U \rrbracket\ a \,\} \\
&\llbracket [U] \circ \pi_2^{\circ} \rrbracket\ b && = \{\, (a, b) \mid a \leftarrow \llbracket\ U^{\circ} \rrbracket\ b \,\} \\
&\llbracket (\Phi + \Psi) \circ (R \triangledown S)^{\circ} \rrbracket\ c && = \llbracket \Phi \circ ((R^{\circ} \circ i_1)) \rrbracket\ c \cup \llbracket \Psi \circ ((S^{\circ} \circ i_2)) \rrbracket\ c \\
&\llbracket \Phi \circ {i_1}^{\circ} \rrbracket\ (i_1\ a) && = \llbracket \Phi \rrbracket\ a \\
&\llbracket \Phi \circ {i_2}^{\circ} \rrbracket\ (i_2\ b) && = \llbracket \Phi \rrbracket\ b
\end{aligned}
$$

Figure 4.9: Constrained exhaustive execution of inverted relations.

$$
\begin{aligned}
&\llbracket \Phi \circ \top \rrbracket\ a && = \{\, a \,\S\, \phi \,\} \\
&\llbracket \Phi \circ \underline{b}^{\circ} \rrbracket\ b && = \{\, b \,\S\, \phi \,\} \\
&\llbracket [U] \circ \pi_1^{\circ} \rrbracket\ a && = \{\, (a, a \,\S\, (U\ a)) \,\} \\
&\llbracket [U] \circ \pi_2^{\circ} \rrbracket\ b && = \{\, (a, a \,\S\, (U^{\circ}\ a)) \,\}
\end{aligned}
$$

Figure 4.10: Constrained selective execution of inverted relations.

of having $\llbracket R^{\circ} \rrbracket\ b$ running freely, it is restricted to produce values that would also be produced by $\llbracket S^{\circ} \rrbracket\ c$, as specified by its post-condition $\rho(\Psi \circ S^{\circ} \circ \underline{c})$. A right-biased implementation would be equivalent. This renders this optimization procedure as not purely syntactic, as only when the concrete value a is known may the exact range of the fork branches be known.

The execution procedure just presented is multi-valued: primitives return every valid value within the provided invariant. However, since the invariants propagated through the combinators are exact, whatever the value chosen in the execution of the primitives will be consumed by the succeeding combinators. As a consequence, using bias selectors $a \,\S\, \phi$ at the primitive combinator level would give rise to a selective execution that is still guaranteed to succeed, thus preserving totality. Figure 4.10 shows such selective procedures for the multi-valued primitives, in a version that will only return a single value, and that would give rise to the selective total ic-lens $\lfloor f \rfloor : A_{\phi \cap \delta f} \trianglerighteq B_{\rho(f \circ \phi)}$.

Example execution To give an example of the proposed techniques, consider the transformation $f = \pi_1 \vartriangle \mathsf{id} : A \times B \to A \times (A \times B)$ and its syntactic bidirectionalization $\lfloor \pi_1 \vartriangle \mathsf{id} \rfloor$ following Algorithm 1. Following the approach presented in Section 4.2.1, we can infer its exact domain (id) and range ($[\pi_1^{\circ}]$) and lift it to an ic-lens that only accepts

views $(a_1, (a_2, b))$ where $a_1 = a_2$. Should the view value be updated, the optimized backward transformation would execute as follows:

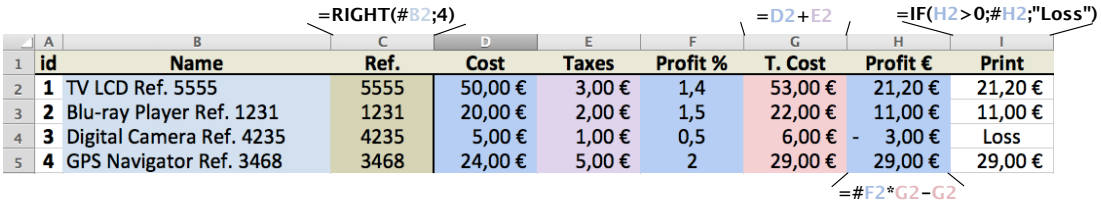
$$
\begin{aligned}
&\llbracket \mathsf{id} \circ ([f] \mathbin{?} \pi_1 : (f^\circ \circ \pi_2)) \rrbracket\ ((a_0, b_0), (a, (a, b))) \\
&\quad = \quad \{\text{-Def. } \Theta \mathbin{?} R : S;\ \Phi \circ ((R \circ S)) \text{ (Fig. 4.8); Domain/Range (Fig. 4.4) -}\} \\
&\{\, y \mid x \leftarrow \llbracket ((\mathsf{id} + (\mathsf{id} \times [\pi_1^\circ]))) \circ ([f]?) \rrbracket\ ((a_0, b_0), (a, (a, b))) \\
&\qquad y \leftarrow \llbracket \mathsf{id} \circ (\pi_1 \triangledown (f^\circ \circ \pi_2)) \rrbracket\ x \,\} \\
&\quad = \quad \{\text{-}((\Phi + \Psi)) \circ (\Theta?) \text{ (Fig. 4.8); } \langle (a_0, b_0), (a, (a, b)) \rangle \notin [f] \text{ -}\} \\
&\{\, y \mid y \leftarrow \llbracket \mathsf{id} \circ (\pi_1 \triangledown (f^\circ \circ \pi_2)) \rrbracket\ i_2\ ((a_0, b_0), ((a, (a, b)))) \,\} \\
&\quad = \quad \{\text{-}\Phi \circ (R \triangledown S) \text{ (Fig. 4.8) -}\} \\
&\{\, y \mid y \leftarrow \llbracket \mathsf{id} \circ (f^\circ \circ \pi_2) \rrbracket\ ((a_0, b_0), (a, (a, b))) \,\} \\
&\quad = \quad \{\text{-}\Phi \circ ((R \circ S)),\ \Phi \circ \pi_1 \text{ (Fig. 4.8); Domain/Range (Fig. 4.4) -}\} \\
&\{\, y \mid w \leftarrow \llbracket [\pi_1^\circ] \rrbracket\ (a, (a, b)), y \leftarrow \llbracket \mathsf{id} \circ f^\circ \rrbracket\ w \,\} \\
&\quad = \quad \{\text{-}[\pi_1^\circ]\ (a, (a, b)) \text{ -}\} \\
&\{\, y \mid y \leftarrow \llbracket \mathsf{id} \circ (\pi_1 \vartriangle \mathsf{id})^\circ \rrbracket\ (a, (a, b)) \,\} \\
&\quad = \quad \{\text{-}\Phi \circ (R \vartriangle S)^\circ \text{ (Fig. 4.8); Domain/Range (Fig. 4.4) -}\} \\
&\{\, y \mid y \leftarrow \llbracket \rho\underline{(a, b)} \circ \pi_1^\circ \rrbracket\ a \,\} \\
&\quad = \quad \{\text{-}\rho\underline{(a, y)} = \rho\underline{a} \times \rho\underline{y} = [\rho\underline{b} \circ \top \circ \rho\underline{a}];\ [R] \circ \pi_1^\circ \text{ (Fig. 4.8) -}\} \\
&\{\, (a, z) \mid z \leftarrow \llbracket \rho\underline{b} \circ \top \circ \rho\underline{a} \rrbracket\ a \,\} \\
&\quad = \quad \{\text{-Simplifications: Appendix A; } R \circ S \text{ (Fig. 4.7); } \langle a \rangle \in \rho\underline{a} \text{ -}\} \\
&\{\, (a, b) \,\}
\end{aligned}
$$

Note how the coreflexive invariants only need to be evaluated at the primitive level. In fact, the semantics of π_1° is multi-valued but the invariant propagated down to its execution forces the generation of a single result. Simplifications are applied to convert the invariant over pairs into the normalized form, which are omitted.

4.3 Spreadsheet Framework

The framework presented in the previous section was built upon arbitrary relational expressions. While extremely expressive, the tradeoff is an unquantifiable complexity. In this section a different approach is explored, considering a more manageable invariant language that emerged in the bidirectionalization of a domain-specific language—that of spreadsheet formulas.

In this context, the subjects of bidirectionalization are ordinary spreadsheet formulas, written in a standard spreadsheet formula language. Each formula is processed as an



	A	B	C	D	E	F	G	H	I
1	id	Name	Ref.	Cost	Taxes	Profit %	T. Cost	Profit €	Print
2	1	TV LCD Ref. 5555	5555	50,00 €	3,00 €	1,4	53,00 €	21,20 €	21,20 €
3	2	Blu-ray Player Ref. 1231	1231	20,00 €	2,00 €	1,5	22,00 €	11,00 €	11,00 €
4	3	Digital Camera Ref. 4235	4235	5,00 €	1,00 €	0,5	6,00 €	- 3,00 €	Loss
5	4	GPS Navigator Ref. 3468	3468	24,00 €	5,00 €	2	29,00 €	29,00 €	29,00 €

Figure 4.11: Bidirectional spreadsheet formula example.

independent bidirectional transformation, what allows us to keep a simple design while handling the necessary environment information. A formula f on a cell B that depends on input cells $A_1, ..., A_n$ will be denoted as $f : A_1 \times ... \times A_n \to B$ (for readability, $A_1 \times .. \times A_n$ will often be abbreviated to A). In general, updates can be reflected back in more than one way to the input cells. In order to keep the system predictable, we adopt a conservative updating principle and ask users to explicitly indicate, for each formula, which cells can be updated, by marking them with the special symbol $\#$. This ensures no modifications occur unless authorized by the user—formulas without $\#$ marks behave as ordinary unidirectional ones.

Consider the spreadsheet for the forecast of profits depicted in Figure 4.11. Each row represents a *product* whose first column defines its identifier, the second and third its name and reference (extracted using the RIGHT function from the name), and the next three its production cost, taxes cost and profit margin. The last column presents a summary report of the total expected profit (calculated in column H), and is processed with an IF statement: profits are simply presented as numerical values, and losses are alerted with the string `"Loss"`. Thus, a modification to the cost of a product will trigger the recalculation of the resulting profit, but the opposite is not possible: one can not simply modify the profit and trigger a recalculation of the costs. Bidirectionalizing such spreadsheet in our system amounts to introducing $\#$ marks denoting how to propagate view updates backwards, as depicted in column H: updates on the total profit are expected to be propagated to the profit margin rather than to the production cost.

It is easy to see that the output of the formula in column I is either a positive number or the string `"Loss"`. Thus, user updates in these bidirectional cells must be somehow restricted: a negative value has no source values that output it (breaking PUTGET). Similarly, in the references column C calculated by a RIGHT function, any string with length higher than 4 is outside its range. Since spreadsheet formulas are not surjective in general[1], we need a way to check whether an updated value is within the domain of a

[1]In fact, spreadsheets, as a lightweight programming language, lack a type system, thus the precise

bidirectional formula. This problem is aggravated by chains of bidirectional formulas: the putback of the second formula must generate values within the range of the first one. Now imagine that the user inserted a constraint in column F stating that the profit margin can never exceed 200% (using, for instance, the 'Data Validation' feature of Excel). The domain of allowed values in the I cells must be coherently restricted, to forbid the insertion of profit values that will exceed the 200% margin. For instance, any value higher than 53 in the column I of product 1 would result in an invalid update. These two scenarios (non-surjective formulas and user-defined constraints) motivated the introduction of invariants in this specific scenario, that filter out invalid values. We will assume user-defined constraints to be applied only at input cells[2], which must be propagated through chains of formulas in order to restrict the range of valid view updates. To contribute to the seamless integration of the bidirectional transformation system, invariants are not propagated backwards to the input cells, since this would affect the pre-existing behavior of the spreadsheet (meaning that forward evaluation may fail, much like in the unidirectional spreadsheet scenario).

Roughly, spreadsheet formulas consist of the sequential composition of primitive functions. This composition may be performed in two ways: either through *formula nesting* (by defining complex formulas $g\ (f\ (A))$) or through *formula chaining* (by having $C = g\ (B)$, where B is itself a formula $B = f\ (A)$). For simplicity, we focus on the latter, since nested formulas can be decomposed into chains of formulas[3]. Data duplication amounts to duplicate cell references, but supporting such formulas would require a deep analysis of the dependency graph and jeopardize our localized approach, and thus are forbidden (see Macedo et al. (2014c) for the technical discussion). Conditional combinators are supported through IF statements, and are addressed independently. Thus, the bidirectionalization of a spreadsheet formula must take into consideration both the chain of formulas that precede it and generate a putback for each one of them. This procedure is resumed in Algorithm 2. Essentially, each formula is bidirectionalized taking into consideration the range of the preceding formula (or the cell invariant, if it is the first formula of the chain) by procedure SYNTH. Its range, calculated by procedure EVAL is then propagated to the next function in the chain. Since the synthesis procedure is constraint-aware and the procedure calculates the exact

invariants are considered to describe the "types" for which formulas are total and surjective.

[2]The evaluation of formulas in systems like Excel actually ignores invariants in output cells, outputting any value produced by a formula regardless of the cell's constraint.

[3]For instance, $B = g\ (f\ (A))$ can be rewritten to $B = g\ (X)$ and $X = f\ (A)$, with a fresh auxiliary cell X.

Algorithm 2: Syntactic bidirectionalization through formula synthesis.

input : formulas $f_1 : A_1 \rightharpoonup A_2, ..., f_n : A_n \rightharpoonup A_{n+1}$ and initial invariant $\phi_1 : \mathbb{P}\, A_1$
output : invariant-constrained lenses $f_1 : A_{1\phi_1} \rhd A_{2\phi_2}, ..., f_n : A_{n\phi_n} \rhd A_{n-1\phi_{n+1}}$
for $i \leftarrow 1$ **to** n **do**
  $\mathsf{put}_{f_i} \leftarrow$ SYNTH (f_i, ϕ_i);
  $\phi_{i+1} \leftarrow$ EVAL (f_i, ϕ_i);
end

range of the functions, the resulting putback will never fail. The remainder of this section explores the development of these procedures. Since the technique is to be integrated in a spreadsheet system, the execution of transformations is to be handle by that same system. Thus, unlike the previous approach, this technique has only a synthesis stage.

4.3.1 Domain-specific Invariants

Language Following the design decisions made regarding the bidirectional spreadsheet system, the invariant language should be intuitive to the typical spreadsheet user and expressible as standard spreadsheet formulas so that type-checking could be performed by the data validation functionalities offered by standard spreadsheet systems.

Formally, an invariant ϕ will consist of a set of $C \in \mathcal{C}$ *clauses* that denote sets of values. The set of all values in an invariant is defined as the union of all values in its comprising clauses. Inspired by typical spreadsheet formulas, clauses over strings, reals, integers or booleans are supported, according to the following abstract syntax:

$$
\begin{array}{lcl}
\mathcal{C} & \in & \mathit{Number} \mid \mathit{Integer} \mid \mathit{Text} \mid \mathbb{B} \\
\mathit{Number} & \in & \langle \mathbb{R}..\mathbb{R} \rangle \mid \langle \mathbb{R}..\mathbb{R}\langle|\rangle\mathbb{R}..\mathbb{R} \rangle \mid \langle|\rangle..\mathbb{R}\rangle \mid \langle \mathbb{R}..\langle| \top_{\mathbb{R}} \\
\mathit{Integer} & \in & [\mathbb{Z}..\mathbb{Z}] \mid [\mathbb{Z}..[|]..\mathbb{Z}] \mid \top_{\mathbb{Z}} \\
\mathit{Text} & \in & \Sigma^* \mid \mathsf{len}_{\mathit{Integer}} \mid \top_{\Sigma^*} \\
\mathit{Boolean} & \in & \mathsf{True} \mid \mathsf{False} \mid \top_{\mathbb{B}}
\end{array}
$$

Here, the clause $\top_c$ contains all values in c (e.g., $\top_{\mathbb{R}}$ contains all numbers), while the clause Σ^* is a string constant. Numeric constants $[x..x]$ or $\langle x..x \rangle$ are often written as x. Integer intervals $[x..y-1]$ and $[x+1..y]$ are typically denoted as open intervals $[x..y[$ and $]x..y]$, respectively. Note that for $x, y \in \mathbb{Z}$, $\langle x..y \rangle$ is a continuous interval

while $[x..y]$ is a discrete one. Most of these clauses are inspired by the data validation features of Excel and the constraint languages of logical spreadsheet systems (Adachi, 2001; Kassoff et al., 2005), that usually support numerical intervals. The notable exception is the $\mathsf{len}_{Integer}$ clause that denotes the set of strings whose length belongs to the integer parameter. To be manageable, invariants are processed in a normalized form, such that their comprising clauses are disjoint. Our invariants can deal with numerical formulas whose range is representable by a finite set of intervals—e.g., the square $A\text{^}2$ over integers is not definable, as it would require an infinite union of singleton sets (though the square of real numbers works fine). Regarding string manipulation, only invariants which are oblivious to the string's content are supported (with the exception of constants)—we cannot give precise invariants for the range of functions like UPPER. Even so, this simple language of invariants is already sufficient to solve interesting bidirectional transformation examples.

Operations The membership test $\langle a \rangle \in \phi$ boils down to testing whether a belongs to any of the comprising clauses, i.e., $\langle a \rangle \in \phi \equiv \bigvee_{C \in \phi} (a \in C)$; since the language was designed to preserve the standard spreadsheet formula language, this operation can be deployed using the underlying spreadsheet system. Invariant manipulation operations must produce themselves normalized invariants, e.g.:

$$\begin{aligned}
&\{[x..y]\} \cup \{[a..b]\} = \mathbf{if}\ a \leqslant y \wedge x \leqslant b \\
&\quad \mathbf{then}\ \{[\mathsf{min}\ (x, a)..\mathsf{max}\ (y, b)]\}\ \mathbf{else}\ \{[x..y], [a..b]\} \\
&\{[x..y]\} \cap \{[a..b]\} = \mathbf{if}\ a \leqslant y \wedge x \leqslant b \\
&\quad \mathbf{then}\ \{[\mathsf{max}\ (x, a)..\mathsf{min}\ (y, b)]\}\ \mathbf{else}\ \{\,\} \\
&\{[x..y]\} \setminus \{[a..b]\} = \\
&\quad \mathbf{if}\ x < a\ \mathbf{then}\ \{[x..\mathsf{min}\ (y, a-1)]\}\ \mathbf{else}\ \{\,\} \cup \\
&\quad \mathbf{if}\ b < y\ \mathbf{then}\ \{[\mathsf{max}\ (x, b+1)..y]\}\ \mathbf{else}\ \{\,\}
\end{aligned}$$

The difference operation on invariants is required for some function primitives, like IF statements; however, for particular cases over strings the resulting invariant may not be expressible in a normalized form (e.g., $\{\top_{\Sigma^*}\} \setminus \{s\}$, representing all strings except s).

Regarding the domain and range of restricted expressions, as already mentioned, only $R \circ \phi$ will be required in this context, which amounts to the EVAL (R, ϕ) procedure. Since nested formulas are assumed to be decomposed into formula chains, the forward transformations are always primitive functions f. Thus, such range can be calculated by

defining the symbolic execution of every supported primitive function over normalized invariants. Since invariants consist of clauses, $\textsc{Eval}\ (f, \phi) = \bigcup_{C \in \phi} (\textsc{Eval}\ (f, C))$, decomposing the view invariant calculation into execution of f over individual invariant clauses.

As an example, consider the LEN primitive function, that calculates the length of a string. Its symbolic execution over normalized invariants is specified as:

$$\begin{aligned}
&\textsc{Eval}\ (\mathsf{LEN}\ (\#A), x \in \Sigma^*) = \{\mathsf{LEN}\ (x)\} \\
&\textsc{Eval}\ (\mathsf{LEN}\ (\#A), \mathsf{len}_x) \quad = \{x\} \\
&\textsc{Eval}\ (\mathsf{LEN}\ (\#A), \top_{\Sigma^*}) \quad = \{\top_{\mathbb{Z}}\} \\
&\textsc{Eval}\ (\mathsf{LEN}\ (\#A), _) \quad = \{\}
\end{aligned}$$

Note that the range calculation depends on the formulas' #-marked cells. For instance, while the range of $\#A + \#B$ contains the case

$$\textsc{Eval}\ (\#A + \#B, ([x..y], [a..b])) = \{[x + a..y + b]\}$$

for $\#A + B$ the same case takes instead the shape

$$\textsc{Eval}\ (\#A + B, ([x..y], [a..b])) = \{[x + B..y + B]\}$$

since B cannot be updated.

The $\mathsf{IF}\ (C, \#A, \#B)$ logical statement is an interesting spreadsheet formula, that affects the flow of update propagation. Without nested formulas, its putback simply needs to decide whether to propagate the update to A or B depending on the condition and the cell invariants on A and B. To make it manageable, predicate C (without # marks) is interpreted as a normalized invariant (ψ_A, ψ_B, K), with ψ_A and ψ_B normalized invariants on A and B, respectively, and K a constant predicate. Also, let ϕ_A and ϕ_B denote the invariants existing in A and B respectively. The range of acceptable values is conditioned by the acceptable updates on the branches (ϕ_A and ϕ_B) and the conditions that trigger the selection of the branches (ψ_A and ψ_B). For instance, values on ϕ_A may not be acceptable if there is no respective acceptable B value that renders condition C true. Range calculation is thus defined by (omitting K for the sake of simplicity):

$$\begin{aligned}
&\textsc{Eval}\ (\mathsf{IF}\ (C, \#A, \#B), ((\psi_A, \psi_B), \phi_A, \phi_B)) = \\
&\quad \textbf{if}\ (\phi_A \setminus \psi_A = \emptyset)\ \textbf{then}\ \phi_B \setminus \psi_B\ \textbf{else}\ \{\phi_B\}\ \cup \\
&\quad \textbf{if}\ (\phi_B \cap \psi_B = \emptyset)\ \textbf{then}\ \emptyset \qquad\quad \textbf{else}\ \phi_A \cap \psi_A
\end{aligned}$$

$c \in$	$c \in$	$\leftarrow c$
$\phi_A \cap \psi_A$	$\phi_B \cap \psi_B$	A
$\phi_A \cap \psi_A$	$\phi_B \setminus \psi_B$	B
$\phi_A \setminus \psi_A$	$\phi_B \cap \psi_B$	B
$\phi_A \setminus \psi_A$	$\phi_B \setminus \psi_B$	B

Table 4.1: IF statement update propagation cases.

The rationale is the following: if all acceptable A values render the condition on A true ($\phi_A \setminus \psi_A = \emptyset$), then no B values that render the condition true are valid ($A_\phi \cap B_\phi$), because when the condition holds, the updated is propagated through A; if all B values render the condition on B false ($\phi_B \cap \psi_B = \emptyset$), then no A values may be produced, since the A branch will never be selected. Table 4.1 helps understand this reasoning by presenting all update propagation possibilities.

This IF statement from spreadsheets differs from the relational conditional combinator because the condition is defined over the environment of the system rather than over a single input value. In fact, the IF statement amounts to conditional expressions $\Theta \,?\, \pi_1 : \pi_2 : A \times A \to A$, with Θ an invariant over pairs $A \times A$.

4.3.2 Executing Constrained Domain-specific Expressions

One of the main focus of the bidirectional spreadsheet system was to provide a seamless integration, aimed at typical spreadsheet users. Following this goal, in order to allow the user to understand the backward semantics of a formula (and eventually parameterize it), we follow a *white-box* approach, specifying all backward transformations as spreadsheet formulas themselves[4].

Preliminaries As has been introduced, the user is able to control which input cells are allowed to be updated through # marks on cell references. Thus, the putback of a bidirectional formulas must assign updated values to each #-marked cell, calculated by independent putback components. The overall putback of a formula may be seen as the tupling of those components (with constant values for unmarked cells). To characterize the different scenarios, each of these components is indexed with extra

[4]For presentation purposes transformations are expressed in an "intermediary" language that can be easily translated to the standard spreadsheet formula language.

marks: ■ denotes the cell for which the putback is being defined; □ denotes another #-marked cell; and cell names denote a non-marked constant parameter. E.g., a formula $C = \#A + \#B$ where both A and B are #-marked gives rise to the lens

$$
\begin{aligned}
&C = \#A + \#B : A \times B \ \triangleright \ C \\
&\mathsf{get}_{C=\#A+\#B}\ (a, b) \quad\ = a + b \\
&\mathsf{put}_{C=\#A+\#B}\ ((a, b), c) = (\mathsf{put}_{■+□}\ ((a, b), c), \mathsf{put}_{□+■}\ ((a, b), c))
\end{aligned}
$$

updating the values of A and B when C is updated, while a formula $C = \#A + B$ gives rise to the lens

$$
\begin{aligned}
&C = \#A + B : A \times B \ \triangleright \ C \\
&\mathsf{get}_{C=\#A+B}\ (a, b) \quad\ = a + b \\
&\mathsf{put}_{C=\#A+B}\ ((a, b), c) = (\mathsf{put}_{■+B}\ ((a, b), c), b)
\end{aligned}
$$

where only A is updated. Note that the definition of $\mathsf{put}_{■+B}$ is not the same as $\mathsf{put}_{■+□}$. Since $C = \#A + B$ is injective, there is only a single valid putback:

$$\mathsf{put}_{■+B}\ ((a, b), c) = c - b$$

For $C = \#A + \#B$ however, c can be divided into A and B in any way as long as they sum up to c and preserve stability. One option is to prioritize A as:

$$
\begin{aligned}
&\mathsf{put}_{■+□}\ ((a, b), c) = c - b \\
&\mathsf{put}_{□+■}\ ((a, b), c) = b
\end{aligned}
$$

Another example, a reasonable putback formula for the LEN function would retrieve as much as possible from the original string, while appending random suffixes when the length increases:

$$
\begin{aligned}
&\mathsf{put}_{\mathsf{LEN}\ (■)}\ (a, b) = \\
&\quad \textbf{if}\ b \leqslant \mathsf{LEN}\ (a)\ \textbf{then}\ \mathsf{LEFT}\ (a, b) \\
&\quad \textbf{else} \qquad\qquad\qquad a\ \&\ \mathsf{REPEAT}\ (\texttt{"A"}, b - \mathsf{LEN}\ (a))
\end{aligned}
$$

The backward evaluation of formula chains exploits the reactive nature of spreadsheets: updating an intermediate formula cell will trigger the backward evaluation of its own formula (recall that nested formulas are assumed to have been decomposed). For instance, for cells $C = g\ (\#B)$, $B = f\ (\#A)$ and $A = a$, update $C \leftarrow c$ triggers an

Algorithm 3: Putback synthesis procedure SYNTH.

input : function $f : A_1 \times ...A_n \rightharpoonup B$ and source domain $(\phi_{A_1}, ..., \phi_{A_n})$
output: $\mathsf{put}_f : A_1 \times ...A_n \times B \rightharpoonup A_1 \times ...A_n$ and view domain ψ_B
$\phi_B \leftarrow \bigcup_{C \in \phi_A} (\text{EVAL } (f, C))$;
for $\#A_i$ **do**
  init $\mathsf{put}_{f\ (...,\blacksquare_i,...)}$;
  for $(\psi_B, (\psi_{A_1}, ..., \psi_{A_n})) \leftarrow f\ \overbrace{(\phi_{A_1}, ..., \phi_{A_n})}$ **do**
    append to $\mathsf{put}_{f\ (...,\blacksquare_i,...)}$ *condition* **if** $b : \psi_B$ **then** $\overleftarrow{f\ (..., \blacksquare_i, ...)\ \psi_{A_i}}$;
  end
end
$\mathsf{put}_f \leftarrow (\mathsf{put}_{f\ (...,\blacksquare_1,...)}, ..., \mathsf{put}_{f\ (...,\blacksquare_n,...)})$;

update on B with $\mathsf{put}_g\ (B, c) = \mathsf{put}_g\ (f\ (A), c)$, which in turn triggers the update on A with $\mathsf{put}_f\ (A, \mathsf{put}_g\ (f\ (A), c)) = \mathsf{put}_f\ (a, \mathsf{put}_g\ (f\ (a), c))$—which amounts exactly to the regular definition of lens composition (Section 3.2). Such localized chaining updates require a careful analysis of the formula dependencies in order to be sound. Roughly, the dependency graph regarding #-marked cells must form a tree whose leaves are value cells (for the in-depth technical discussion the reader is redirected to (Macedo et al., 2014c)). A major consequence is that duplication (embodied by multiple references to the same cell) is disallowed[5].

Synthesis procedure Algorithm 2 split formula bidirectionalization into a local synthesis procedure that at each formula cell, statically bidirectionalizes its primitive function f under current cell constraint $(\phi_{A_1}, ..., \phi_{A_n})$. This putback, needs to be able to, given an updated view b, find one possible source value a that is consistent with b and satisfies the existing source invariant. The simple putback definitions just presented are not sufficiently flexible to that purpose. Instead, we developed a constraint-aware synthesis procedure that generates such putback formulas, relying on specific synthesis procedures for each supported primitive function. This procedure is summarized in Algorithm 3.

The first step calculates the range ϕ_B through the EVAL operation explored in Section 4.3.1. The remainder steps deal with the generation of the putback. Without

[5]Since bidirectionalization is performed at the cell level, duplication is inherently supported to a certain degree (for formulas with disjoint dependency graphs) even with these restrictions. For example, consider $B = f(\#A)$ and $C = g(A)$; updating the value B assigns a new value to A through f, which in turn triggers a forward evaluation of g, restoring the consistency between A and C.

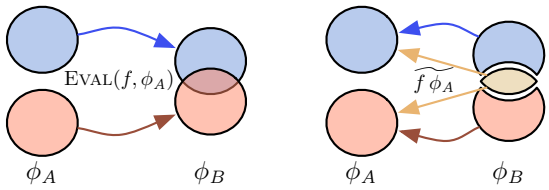


Figure 4.12: Representation of the traceability link $\widetilde{f\ \phi_A}$.

going into much detail, $f\ \widetilde{(\phi_{A_1}, ..., \phi_{A_n})}$ denotes the normalized traceability between source and view invariants, and is required to ease the putback component synthesis. Its left projection must form a partition of the range invariant ϕ_B.

For instance, consider a function $f(A) = A\text{^}2$, with a pre-existing source invariant $\phi_A = \{\langle 0..10\rangle, \langle -6.. -5\rangle\}$. Applying EVAL $(A\text{^}2 \circ \phi)$ yields:

$$\begin{aligned}\langle 0..10\rangle &\mapsto \langle 0..100\rangle\\ \langle -6.. -5\rangle &\mapsto \langle 25..36\rangle\end{aligned}$$

With view invariant $\phi_B = \{\langle 0..100\rangle\}$. The clauses of ϕ_B overlap, as values $\langle 25..36\rangle$ are produced by both $\langle 0..10\rangle$ and $\langle -6.. -5\rangle$. Since the B component of the normalized traceability must form a partition, these must be split, and thus $\widetilde{A\text{^}2\ \phi}$ becomes:

$$\begin{aligned}\{\langle 0..25\langle, \rangle 36..100\rangle\} &\mapsto \{\langle 0..10\rangle\}\\ \{\langle 25..36\rangle\} &\mapsto \{\langle -6.. -5\rangle, \langle 0..10\rangle\}\end{aligned}$$

Figure 4.12 depicts this process graphically, by comparing EVAL (f, ϕ_A) with the produced $\widetilde{f\ \phi_A}$. By keeping this traceability in a normalized form, the source invariant into which the updated a' source must fall for each view b can be easily found. It is worth noting that calculating this normalized traceability is only feasible in this context of semantic invariants: in the syntactic invariants from Section 4.2.1 such task would be impossible.

Operation $\overleftarrow{f\ (..., \blacksquare_i, ...)\ \psi_{A_i}} : A \times B \to A_i$, for an updated view $b : \psi_B$ and an original source $(a_1, ..., a_n)$, produces a consistent update $a_{i'} : \psi_{A_i}$ for the cell A_i. These operations shall be statically synthesized, for a particular source invariant $(\phi_{A_1}, ..., \phi_{A_n})$ and primitive function f, and are the components that actually describe the behavior of each putback formula. The $\mathsf{put}_{f\ (..., \blacksquare_i, ...)}$ transformation for a cell $\#A_i$ simply assembles these components together: based on the traceability, $\mathsf{put}_{f\ (..., \blacksquare_i, ...)}$ tests to which ψ_B invariant the updated b belongs and applies the corresponding $\overleftarrow{f\ (..., \blacksquare_i, ...)\ \psi_{A_i}}$. The overall put_f simply applies the putback synthesized for each $\#A_i$ in parallel.

Consider as an example an invariant $\phi_A = \{\texttt{"abc"}, \texttt{"xyz"}, \mathsf{len}_{[4..10]}\}$ over a cell A. For a cell $B = \mathsf{LEN}\ (\#A)$ calculating the traceability $\widetilde{\mathsf{LEN}\ (A)\ \phi_A}$ results in:

$$\begin{aligned}\{3\} &\mapsto \{\texttt{"abc"}, \texttt{"xyz"}\}\\ \{[4..10]\} &\mapsto \{\mathsf{len}_{[4..10]}\}\end{aligned}$$

with view invariant $\phi_B = \{[3..10]\}$. Now, putback components $\overleftarrow{\mathsf{LEN}\ (A)\ \{3\}}$ and $\overleftarrow{\mathsf{LEN}\ (A)\ \{[4..10]\}}$ must be synthesized. After being composed, the overall $\mathsf{put}_{\mathsf{LEN}\ (\blacksquare)}$ for the particular source invariant ϕ_A is then:

$$\begin{array}{l}
\mathsf{put}_{\mathsf{LEN}\ (\blacksquare)}\ (a, b) = \\
\quad \textbf{if}\ b = 3\ \textbf{then} \\
\quad\quad \textbf{if}\ b \leqslant \mathsf{LEN}\ (a)\ \textbf{then} \\
\quad\quad\quad \textbf{if}\ \mathsf{LEFT}\ (b, a) = \texttt{"abc"}\ \textbf{then} \quad \texttt{"abc"} \\
\quad\quad\quad \textbf{else if}\ \mathsf{LEFT}\ (b, a) = \texttt{"xyz"}\ \textbf{then}\ \texttt{"xyz"} \\
\quad\quad\quad \textbf{else} \quad \mathsf{LEFT}\ (b, a) \S \{\texttt{"abc"}, \texttt{"xyz"}\} \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{if}\ \mathsf{LEFT}\ (b, \texttt{"abc"}) = a\ \textbf{then} \quad \texttt{"abc"} \\
\quad\quad\quad \textbf{else if}\ \mathsf{LEFT}\ (b, \texttt{"xyz"}) = a\ \textbf{then}\ \texttt{"xyz"} \\
\quad\quad\quad \textbf{else} \quad a \S \{\texttt{"abc"}, \texttt{"xyz"}\} \\
\quad \textbf{if}\ 4 \leqslant b \leqslant 10\ \textbf{then} \\
\quad\quad \textbf{if}\ b \leqslant \mathsf{LEN}\ (a)\ \textbf{then}\ \mathsf{LEFT}\ (b, a) \\
\quad\quad \textbf{else} \quad a\ \&\ a \S \{\mathsf{len}_{b - \mathsf{LEN}\ a}\}
\end{array}$$

If the updated view b is 3, it searches the constants for the closest string; otherwise the $\mathsf{len}_{[4..10]}$ clause allows it to freely generate the closest solution. This putback is automatically synthesized and could eventually be simplified: if $b : \{3\}$ is true, $b \leqslant \mathsf{LEN}\ (a)$ always holds.

Note the use of $a \S \phi$ operations in the putback specification. Likewise the previous approach, the allows the biased selection of values when there are multiple valid solutions. In the spreadsheet context, at implementation time, the $a \S \phi$ operations can either be given a default value or remain a placeholder which the user is able to parametrize within ϕ.

4.4 Discussion

This chapter addressed an open problem in view-update frameworks that arises in the presence of (explicit or implicit) constraints over the transformation domains and is responsible for the latent partiality found in most practical lens frameworks. We have proposed to alleviate this problem by regarding datatype invariants as first class entities, which allowed us to reason about the bidirectional transformations properties within their scope. The result is an expressive bidirectional transformation scheme that is able to support expressive combinators like duplication or conditional choices. The proposed bidirectional transformation framework can be seen as a bidirectional transformation language over algebraic types similar to the one for lenses over generalized trees first developed by Foster et al. (2007). They devise a complex set-based type system with invariants to precisely define the domains for which their combinators are well-behaved. However, combining lenses requires matching on invariants rather than on types, which is too restrictive. A dual approach is followed by Foster et al. (2008), where composition requires matching on equivalence relations that relax the lens domains.

The potential of a purely combinatorial approach was analyzed, but the overhead of defining exhaustive backward transformations counterbalances its main advantages, which led to the exploration of context-aware syntactic bidirectionalization procedures. This was explored in two fronts: one relied on the relational calculus as a general-purpose language for the specification of both transformations and invariants, the other addressed the issue in the particular context of spreadsheet formulas.

The former approach builds up on previous work by Pacheco and Cunha (2010, 2011) on the development of a language of functional point-free combinators allowing only surjective transformations. In this chapter such language was extended to support typical non-surjective combinators such as forks. The presentation is a simplified version of a more thorough exploration presented in (Macedo et al., 2012), where generalized constrained relational expressions are effectively evaluated. While including more combinators (like sums and injections), the fundamental idea is the same presented in the chapter. In fact, the technique proved to be independent of the bidirectional transformation context and applicable to relational expressions in general. Unlike the data abstraction approach by Wang et al. (2010), our lens language allows arbitrary type constructors and deconstructors without extending the language with *ad hoc* primitives and surjectivity tests. Although omitted here, in the full paper (Macedo et al., 2012) we have shown that this technique can be extended to support typical recursion patterns,

in particular folds and unfolds. Much like the normalized invariants for products, a normalized shape for recursive types also proved to be useful in the calculation of the domain and range of recursive transformations.

Similar to our syntactic bidirectionalization technique, some proposed frameworks derive the backward transformations by calculation, but are less expressive than ours. Mu et al. (2004) propose a technique to derive the putback by inverting *injective* forward transformations through algebraic reasoning, while Matsuda et al. (2007) bidirectionalize a restricted first-order language (namely, without duplication) based on a notion of view-update under constant complement. They also calculate an automata that matches the exact domain of the transformations, and acts similarly to our invariants. The lens language for graph transformations proposed by Hidaka et al. (2010) processes view insertions using the universal resolving algorithm (URA), exploring all possible right inverses for the forward transformation. URA (Abramov and Glück, 2000) has been developed to compute the inverses of functional programs. Like our evaluation algorithm, it is complete (it lazily enumerates all possible values) but not always terminating (since recursive types may admit infinitely many values). Nevertheless, unlike in URA, we are able to optimize expressions before evaluation using the relational calculus. This allows to cut many intermediate infeasible values, making value generation for most invariants much more efficient.

The latter approach is also a simplified presentation of the work presented (Macedo et al., 2014c), by omission of most technical details. The result was the deployment of a functional add-in for Microsoft Excel, deployed in an *online* setting, benefiting from the reactive nature of spreadsheet systems. In this sense, it is similar to the interactive bidirectional XML editor proposed by Hu et al. (2008), which reacts immediately to one operation at a time, although to ensure that after each update the editor converges into a consistent state, transformations only obey one-and-a-half round-tripping laws. To the best of our knowledge, existing work on the application of bidirectional transformation techniques to spreadsheets has not yet considered the bidirectionalization of spreadsheet formulas. Cunha et al. (2012) developed an OpenOffice plugin that tackles the bidirectional synchronization of spreadsheet models (modeling their business logic) and conforming instances.

Significant work has been done in extending spreadsheets with solving capabilities (Fylstra et al., 1998; Adachi, 2001; Konopasek and Jayaraman, 1984; Cervesato, 2013; Kassoff et al., 2005) that could to some extent provide bidirectional behavior.

However, at the user-interface level, solving is more tailored for search and optimization problems described through a set of global constraints over the spreadsheet, the spreadsheet acting as a mere interface to the solver. Typically, for every backward computation, the user must set up a new constraint-solving problem using a specialized interface. The user is expected to reason about bidirectionalization in a new language and/or interface—not using plain spreadsheet formulas. Our "solver" is not explicit and upfront but a backend: constraints are not first class entities that the user manipulates, but integrated seamlessly into the bidirectionalization approach. Besides, constraint-solvers typically support arithmetic constraints, but not high-level combinators like conditionals, table lookups or string manipulation, at which bidirectional transformations excel. GoalDebug (Abraham and Erwig, 2007) allows users to fix incorrect formulas by defining (numeric) constraints over their outputs: if the constraints are broken, the system proposes changes on the *formula* in order to restore consistency. These changes may involve modifying the formula or one of its parameters, being propagated until value cells are reached. While at first sight this technique resembles our own, they are fundamentally different: bidirectional transformations are meant to propagate updates between the transformation domains, leaving formulas unchanged.

It is widely accepted that non-deterministic transformations are undesirable due to their inherent predictability, since repeated executions may yield different results (Stevens, 2010). It should however be emphasized that exhaustive bidirectional transformations that return all valid elements are not non-deterministic, as they always return the same set of solutions. In fact multi-valued specifications could promote predictability by allowing users to control the particular semantics of the transformations that the bidirectional transformation language is not able to capture. In fact, the selective frameworks proposed in Section 4.2 and Section 4.3 where achieved through the insertion of biased selections at primitive level, that could be parametrized by the used. The next chapter proposes a different technique to address this issue: to refine the behavior of loosely specified backward transformations through the definition of an order of preference over the source updates.

Chapter 5

Least-change Lenses

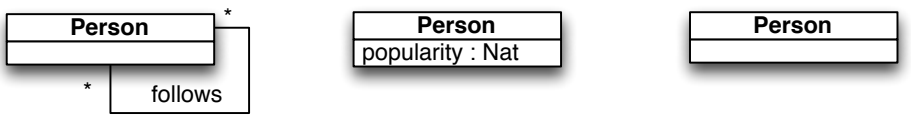
Recall the PUTGET and GETPUT round-tripping laws from the regular lens framework, presented at Section 3.2.1 that render a (partial) lens $f : A \rhd B$ well-behaved:

$$a \in \mathsf{put}_f\ (a_0, b) \Rightarrow b \in \mathsf{get}_f\ a \qquad \text{PUTGET}$$

$$b_0 \in \mathsf{get}_f\ a_0 \Rightarrow a_0 \in \mathsf{put}_f\ (a_0, b_0) \qquad \text{GETPUT}$$

Throughout this dissertation we have been interpreting these laws as upper- and lower-bounds for the behavior of the backward transformation: PUTGET entails that view updates are translated exactly by put_f, and thus imposes an *upper* bound on its behavior (admissible behaviors); GETPUT entails that if the view is not changed, then it must be "put back" to the same source, and thus imposes a *lower* bound on the behavior of put_f (mandatory behaviors). Thus, in general, these laws do not entail a unique backward transformation but instead define a range of valid backward propagation procedures.

Consider as an example the class diagram $TW1$ in Figure 5.1a that models a simplified Twitter: models consist of persons who may follow other persons (persons are allowed to follow themselves for simplicity purposes). A model transformation $\mathtt{tw}_1 : TW1 \rightharpoonup TW2$ that converts the "follows" links to the popularity of each person could be defined, resulting in models conforming to $TW2$ in Figure 5.1b (with the additional constraint maxpop stating that popularity cannot be higher than the number of existing persons). To satisfy GETPUT, a backward transformation for $\mathtt{tw}_1$ must return the same source when the view is not changed. If the view is changed, then it must satisfy PUTGET, which essentially only requires popularity preservation. For example, should somebody be added to a $TW2$ view with popularity n, the putback has



(a) Meta-model $TW1$. (b) Meta-model $TW2$ with $\texttt{Person}_{\texttt{maxpop}}$. (c) Meta-model $TW3$.

Figure 5.1: Meta-models for different views of a simplified Twitter.

freedom to determine the n persons that should follow her or him. Unfortunately, it is also free to rearrange the followers of all *other* persons, as long as their total popularity remains the same.

As already discussed in Section 3.2.4, the lens laws should be taken as first principles: for example, GETPUT "only provides a relatively loose constraint on the behavior of lenses", as originally remarked by the authors of the lens framework (Foster, 2009). Thus, most modern combinatorial bidirectional transformation languages (see references (Foster et al., 2007; Bohannon et al., 2008; Pacheco and Cunha, 2010; Hidaka et al., 2010; Hermann et al., 2011) among others) select a fixed putback from among those deemed well-behaved at design time, in an attempt to specify the expected backward behavior of each combinator. Others propose interactive algorithms that dialog either with the language designer (Keller, 1986) or with the user (Larson and Sheth, 1991) in order to disambiguate the selection of updates. A more drastic approach advocates that the only way to fully control the behavior of update propagation is to allow the user to directly write the putback and have the forward transformation derived from it instead (Pacheco et al., 2014). Refining the transformation domains as proposed in Chapter 4 is another mechanism that may reduce such unpredictability, by restricting the values among which the putback is able to choose. However, while in the end these approaches may provide sensible backward transformations, to understand the behavior of a lens, a user cannot rely solely on the laws and must directly inspect the definition of the backward transformation. To provide a better account of its behavior to users, the putback's update selection criteria should be embodied in the round-tripping properties of the bidirectional transformation framework.

This led to the emergence of frameworks that incorporate the notion of "optimal" update in the formalization of well-behavedness. These are typically attained by the definition of a preference order over the domain elements, that guide the selection of the source update. Some approaches propose the introduction of "absolute" orders

$\preceq : A \leftrightarrow A$ and $\sqsubseteq : B \leftrightarrow B$ over the transformation domains (Hegner, 2004; Johnson et al., 2010), and then force the putbacks to preserve the order of the updates as:

$$\mathsf{get}_f\ a \sqsubseteq b \Rightarrow a \preceq \mathsf{put}_f\ (a, b)$$

This kind of absolute orders are suitable in operation-based frameworks, like in the database context where the insertion/deletion of tuples establishes an order over updates: if a view b' results from the insertion of a tuple in b—rendering b' "greater" than b—then it should be reflected on the source also as an insertion—rendering the update source "greater" than the original one. Yet, in a state-based setting it is not as easy to envision what this order might represent. Consider a left-projection lens $\pi_1 : \mathbb{N} \times \mathbb{N} \trianglerighteq \mathbb{N}$ and an update propagation $(a', b') = \mathsf{put}_{\pi_1}\ ((a_0, b_0), a)$. From GETPUT, a' is necessarily the updated view a, thus put_{π_1} must only assign a value to b' such that $a_0 \leqslant a \Rightarrow (a_0, b_0) \preceq (a, b')$. Assuming the product order $(a, b) \preceq (a', b') \equiv a \leqslant a' \wedge b \leqslant b'$, the above law entails that $a_0 \leqslant a \Rightarrow b_0 \leqslant b'$. Although this does further document the behavior of the backward transformation, it is not clear why the value of b_0 may not be decremented when the view a_0 is incremented. Rather than orders on *values*, in state-based frameworks one must reason about orders on *updates*.

This leads to the principle of least-change proposed by Meertens (1998) for the bidirectional transformation framework of constraint maintainers and already presented in Section 3.2.4. Here, backward propagation is guided by a "relative" order $\preceq : A \to (A \leftrightarrow A)$ on sources, that compares the acceptable sources in relation to the distance to the original one. When applied to lenses, this principle will allow us to tighten the bounds imposed by the traditional laws, thus making the behavior of the backward transformation more predictable, by enforcing:

$$a \in \mathsf{put}_f\ (a_0, b_0) \Rightarrow b_0 \in \mathsf{get}_f\ a \wedge (\forall\ a' : A \mid b_0 \in \mathsf{get}_f\ a' \Rightarrow a \preceq_{a_0} a')$$

Now, in the π_1 lens example mentioned above, assuming a reasonable order on pairs (like $(a, b) \preceq_{(a_0,b_0)} (a', b') \equiv a \leqslant_{a_0} a' \wedge b \leqslant_{b_0} b'$, where $a \leqslant_{a_0} a'$ compares the absolute value of the difference between a/a' and a_0), the selected source would be (a', b), that is at minimal distance from (a, b). Returning to the Twitter example, if one defines an order $\preceq^{TW1} : TW1 \leftrightarrow TW1$ where a model is "closer" than another if it shares with the original source model more persons and "follows" links, the problematic putback for $\mathtt{tw}_1$ that rearranged the previously existing followers will no longer be

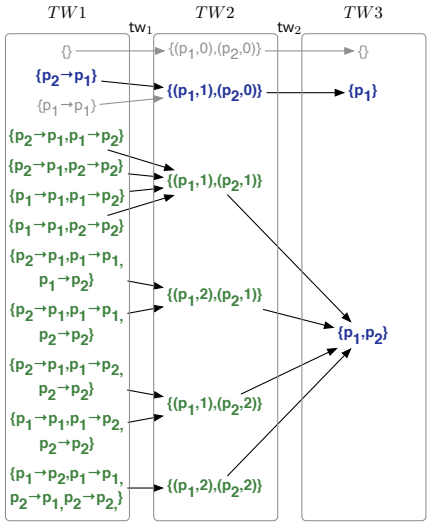


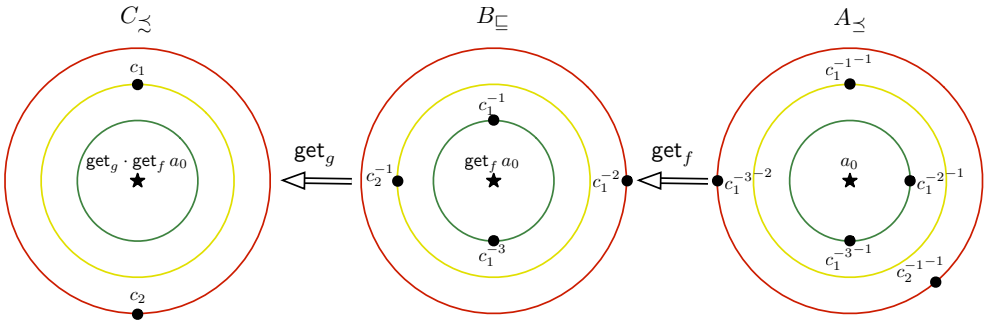
Figure 5.2: Instantiation of lens $\mathtt{tw}_2 \circ \mathtt{tw}_1 : TW1 \triangleright TW3$.

acceptable—although different backward transformations can still be defined, reflecting the different follow arrangements for the newly inserted persons.

While the least-change property is intuitive to formulate, deploying an effective combinatorial system that enforces it is not straight-forward. Indeed, Meertens did not investigate the composition of least-change constraint maintainer (since maintainers are by nature not suitable for compositional reasoning (Meertens, 1998, p. 42)) and many authors (Stevens, 2010; Diskin, 2008; Hofmann et al., 2011) view the lack of compositionality as a drawback. This results in frameworks where propagating updates backwards through composition may generate elements with little resemblance with the original ones. Back to our example, consider a second transformation $\mathtt{tw}_2 : TW2 \rightharpoonup TW3$ that filters out persons with no followers, removing the popularity of each person in the process, ending up with meta-model $TW3$ from Figure 5.1c. Figure 5.2 depicts update propagation alternatives over the unrestricted composition $\mathtt{tw}_2 \circ \mathtt{tw}_1$ for a fixed universe with two persons p_1 and p_2 (the set of persons is omitted from the $TW1$ and $TW2$ instantiations, having only the "follows" and "popularity" relations depicted, respectively), for a scenario where p_2 is introduced in the view of $\{p_2 \rightarrow p_1\}$ (blue instances denote the input values of the putback). The putbacks of $\mathtt{tw}_1$ and $\mathtt{tw}_2$ are able to generate any acceptable source (here denoted by green instances), resulting in updates that are no distinguishable by the lens but not perceived as minimal by the user, including those that disregard the original follower relation $p_2 \rightarrow p_1$.

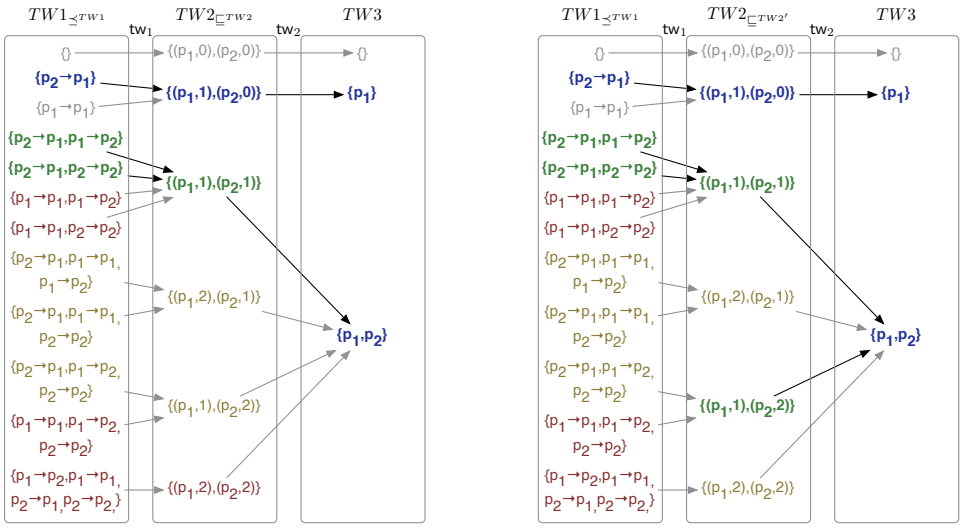
Developing complex least-change bidirectional transformations requires a careful

analysis of the relation between the transformations and the preference orders defined over the transformation domains. Our fundamental (and novel) research question in this chapter is: *under which conditions may two least-change lens be sequentially composed such that the resulting lens is also a least-change lens?* Concretely, when does $f \circ g : A_{\preceq} \rhd C_{\precsim}$ hold? Consider the tentative sequential composition of two *least-change lenses* $f : A_{\preceq} \rhd B_{\sqsubseteq}$ and $g : B_{\sqsubseteq} \rhd C_{\precsim}$, where the putback of f is known to produce minimal source updates according to $\preceq$ and that of g minimal source updates according to $\sqsubseteq$, as depicted in the following diagram:



On the left subfigure, C views spread over concentric circles according to the distance entailed by $\precsim$ to a hypothetical center denoting $\mathsf{get}_g \circ \mathsf{get}_f\ a_0$ (represented by a star). Similarly for the center and right subfigures, for B and A elements with center $\mathsf{get}_f\ a_0$ and a_0 and under orders $\sqsubseteq$ and $\preceq$, respectively. Elements c^{-i} denote the B pre-images of element c under get_f (which are not unique, since forward transformations need not be injective), while elements c^{-i-j} denote the A pre-images of element c^{-i} under get_g.

It is easy to envision why least-change composition is not trivial: the selection of "closest" elements by the putback of g does not necessarily entail "closest" elements are produced by the putback of f. For instance, if $(\mathsf{get}_g \circ \mathsf{get}_f)\ (a_0)$ is updated to c_1, and the putback of g selects the minimal update ${c_1}^{-1}$, the putback of f would necessarily return its only view ${c_1}^{-1^{-1}}$, which is not the minimal update in relation a_0; yet, the selection of the non-minimal update ${c_1}^{-2}$ by the putback of g would allow the putback of f to select the minimal update in relation to a_0, ${c_1}^{-2^{-1}}$. Moreover, in a combinatorial context, well-behaved transformations are expected to be combined into transformations that are also well-behaved without inspecting the particular behavior of the sub-lenses. Thus, the ability to be composable should be intrinsic to well-behaved least-change lenses, rather than to specific pairs. Suppose, for instance, that $\mathtt{tw}_1$ is a least-change lens minimizing updates according to the order $\preceq^{TW1}$ presented above. Can it be safely composed with another (arbitrary) least-change lens? The answer depends on



(a) $\preceq = \preceq^{TW1}$ and $\sqsubseteq = \sqsubseteq^{TW2}$. (b) $\preceq = \preceq^{TW1}$ and $\sqsubseteq = \sqsubseteq^{TW2'}$.

Figure 5.3: Instantiations of least-change lens $\mathtt{tw}_2 \circ \mathtt{tw}_1 : TW1_{\preceq} \rhd TW3_{\precsim}$.

the metrics followed by $\mathtt{tw}_2$: if an order $\sqsubseteq^{TW2'}$ on $TW2$ ignores the popularity of new persons, it will obviously lead to problematic executions of $\mathtt{tw}_1$. This is depicted in Figure 5.3b: if the $TW2$ model $\{(p_1, 1), (p_2, 2)\}$ is selected—a minimal update in the perspective of the putback of $\mathtt{tw}_2$—the putback of $\mathtt{tw}_1$ is not able to produce a minimal repair. What if $\mathtt{tw}_2$ performs under an order $\sqsubseteq^{TW2}$ on $TW2$ that considers the popularity of new persons, as depicted in Figure 5.3a, will the putback of $\mathtt{tw}_1$ be able to always produce minimal updates? While that does seem to be the case, intuitive criteria must be defined so that each lens can be easily tested for its least-change properties.

Like in invariant-constrained lenses, we will show that allowing putbacks to be exhaustive will somehow tame these issues: the putback of the first transformation would be able to consider the different alternatives, and maybe return the desired least-changed sources. For instance, in Figure 5.3b, if the putback of $\mathtt{tw}_2$ returns both $\{(p_1, 1), (p_2, 2)\}$ and $\{(p_1, 1), (p_2, 1)\}$, the putback of $\mathtt{tw}_1$ will be able to produce a minimal update from the latter. While multi-valued composition is not ideal—as the putback of f needs to search which intermediate candidates produced by the putback of g will allow it to generate minimal sources updates—it may still be better than trying to discover the least-changed sources of the composed transformation directly, since for reasonable orders the guidance from the inner transformations will greatly limit the search space.

The contributions of this chapter, as well as its structure, are the following:

- we introduce the notion of *least-change lenses* by applying the principle of least-change to regular lenses (Section 5.1) under two dual formalizations: one based on selective single-valued putbacks and another on exhaustive multi-valued ones that correspond to the tightening of the upper- and lower-bounds imposed by the round-tripping laws, respectively.
- we explore the viability of a combinatorial approach under least-change for the two formalizations, in particular under sequential composition (Section 5.2), ending up with a set of criteria under which least-change is indeed preserved by composition.

Section 5.3 closes the chapter with an overview and discussion of the contributions.

5.1 Least-change Lens Framework

In Section 3.2.4 the principle of least-change as proposed by Meertens (1998) for the constraint maintainer framework was presented. In this section we adapt this property to the asymmetrical lens setting and explore the consequences of adopting it.

5.1.1 Defining Least-change Lenses

Adapting the least-change property to the lens setting is rather straight-forward. In this asymmetric context of least-change lenses, or *lc-lenses*, the forward transformation is assumed to be a concrete, user-defined transformation, and thus only the putback needs be further controlled by least-change. To that purpose there is the need to define a family of total preorders over source elements $\preceq : A \to (A \leftrightarrow A)$ that are stable in regard to its input element, i.e., a_0 is the minimum of $\preceq_{a_0}$ (Definition 3.8). These stable preorders could be derived from distance functions over sources $\Delta : A \to A \to \mathbb{N}$ that satisfies the identity of indiscernibles (Equation (DISTSTABLE)), which is denoted by $[\Delta]$. In our example a $TW1$ model instance consists of a pair (W, F), with W a set of persons and F a "follows" relation from persons to persons (a set of pairs $F \subseteq W \times W$). Thus, a suitable metric is the size of their symmetric differences:

$$\Delta^{TW1}\ (W, F)\ (W', F') = \\ |W \ominus W'| + |F \ominus F'|$$

$$\textbf{where } X \ominus Y = (X - Y) \cup (Y - X)$$

Lifting this distance to a family of preorders results in the $\preceq^{TW1}$ previously presented.

A model instance of $TW2$ can instead be seen as a pair (W, P), where W is again a set of persons and P is seen as a multi-set of persons P denoting their popularity—a person occurs as many times in P as its popularity. Membership test $X\ x$ shall denote the set indicator function, returning the number of occurrences of x in X (generalizing that over regular sets that returns either 0 or 1). Again, a suitable metric could be the size of their symmetric differences (generalized to multi-sets with the obvious definition):

$$\begin{aligned}&\Delta^{TW2}\ (W, P)\ (W', P') = \\ &\quad |W \ominus W'| + |P \ominus P'| \\ &\qquad \textbf{where } X \ominus Y = (X - Y) \cup (Y - X)\end{aligned}$$

Lifting Δ^{TW2} results in the family of preorders $\sqsubseteq^{TW2}$, and thus pairing metrics Δ^{TW1} and Δ^{TW2} results in behavior like the one depicted in Figure 5.3a[1]. The alternative order $\sqsubseteq^{TW2'}$ that ignored the popularity of persons not contained in both instances could be defined by the following distance function:

$$\begin{aligned}&\Delta^{TW2'}\ (W, P)\ (W', P') = \\ &\quad |W \ominus W'| + |(P \otimes (W \cap W')) \ominus (P' \otimes (W \cap W'))| \\ &\qquad \textbf{where } (X \otimes Y)\ x = \textbf{if } x \in Y \textbf{ then } X\ x \textbf{ else } 0\end{aligned}$$

Pairing metric $\Delta^{TW2'}$ with Δ^{TW1} results instead in behavior like the one depicted in Figure 5.3b.

Selective Least-change Lenses By inspecting the above defined least-change law definition, we see that it acts as an upper-bound for the putback: it removes values that are farther away from the original source from the range of valid updates. This gives rise to a selective version of least-change lenses: one in which, if the putback is defined, it returns least-changed sources.

Definition 5.1 (selective lc-lens)**.** *Given a family of stable total preorders* $\preceq : A \to (A \leftrightarrow A)$*, a* well-behaved selective least-change lens $f : A_{\preceq} \rhd B$ *consists of transformations* $\mathsf{get}_f : A \rightharpoonup B$ *and* $\mathsf{put}_f : A \times B \rightharpoonup A$ *such that* GETPUT *and the following property hold:*

$$a \in \mathsf{put}_f\ (a_0, b_0) \Rightarrow b_0 \in \mathsf{get}_f\ a \wedge (\forall\ a' : A \mid b_0 \in \mathsf{get}_f\ a' \Rightarrow a \preceq_{a_0} a')\ \text{LC-PUTGET}$$

[1] These amount to the graph-edit distance, that will be automatically inferred from the meta-models in succeeding chapters.

LC-PUTGET is a refinement of PUTGET, in the sense that the resulting source value is required to be not only acceptable, but also one of closest to the original a_0 among the sources that share the same view b, according to the preorders $\preceq$. Since the preorder over elements B does not affect the definition of least-change, it is usually omitted from the type declarations.

Although this definition further restricts the backward transformation, there may still be more than one valid $\mathtt{put}_f$ for the same $\mathtt{get}_f$ and preorders $\preceq$, since there may be multiple source elements at the same distance from the original source. Consider, from our running example, transformation $\mathtt{tw}_1$ with preorders $[\Delta^{TW1}]$ over $TW1$. If we insert a person p in the view with popularity n, $\mathtt{put}_{\mathtt{tw}_1}$ is free to choose who will be her followers, since any instance with n followers for p is at the same distance from the original source (although it is no longer free to change the followers of other persons, since this would result in more distant instances). This is depicted in the left-hand side of Figure 5.3a: with this formulation the putback would have to select a single source from those colored green. Regarding transformation $\mathtt{tw}_2$, assuming the preorders $[\Delta^{TW2}]$, when a new person is inserted to the view, $\mathtt{put}_{\mathtt{tw}_2}$ must assign him or her popularity 0, leaving only one acceptable source update (cf. right-hand side of Figure 5.3a). However, assuming preorders $\sqsubseteq^{TW2'}$ instead, $\mathtt{put}_{\mathtt{tw}_2}$ is free to assign the new person an arbitrary popularity (cf. right-hand side of Figure 5.3b). In both cases, the refined putback must preserve the popularity of the previously existing persons.

Obviously, the preorder may not discriminate much, and at one end of the spectrum we may have a preorder induced by the metric that considers all different values at the same distance[2]:

$$\Delta^0\ s\ s' = \mathbf{if}\ s = s'\ \mathbf{then}\ 0\ \mathbf{else}\ 1$$

In this case, LC-PUTGET degenerates into the regular PUTGET, allowing the same backward transformations as the regular lens laws, only recovering the original source when the view update is null, bringing us back to the scenario depicted in Figure 5.2. The other extreme case occurs when the preorder is refined to the point where minimal values are unique and there is a single valid putback. This will occur when, for any $a_0 \in A$, $\Delta\ a_0 : A \to \mathbb{N}$ is injective (which when lifted results in a total order), since the minimal sources will always be unique.

[2]In metric space terminology, this is known as the *discrete metric* on a set.

Exhaustive Least-change Lenses For many pragmatic examples a single-valued putback that commits to a particular minimal update will be too restrictive to allow compositionality. The least-change composition example depicted at Figure 5.3b clearly fails for selective putbacks. Therefore, we will introduce a variant of least-change lenses where the putback is allowed to be multi-valued, enumerating all possible minimal updates. As usual, to be distinguished from the single-valued versions, these putbacks will be denoted as Put_f.

Definition 5.2 (exhaustive lc-lens)**.** *Given a family of stable total preorders* $\preceq : A \to (A \leftrightarrow A)$*, a* well-behaved exhaustive least-change lens $f : A_{\preceq} \blacktriangleright B$ *consists of a partial forward transformation* $\mathsf{get}_f : A \rightharpoonup B$*, and a multi-valued backward transformation* $\mathsf{Put}_f : (A \times B) \leftrightarrow A$ *such that* PUTGET *and the following property hold:*

$$b_0 \in \mathsf{get}_f\ a \wedge (\forall\ a' : A \mid b_0 \in \mathsf{get}_f\ a' \Rightarrow a \preceq_{a_0} a') \Rightarrow a \in \mathsf{Put}_f\ (a_0, b_0) \quad \text{LC-GETPUT}$$

In exhaustive least-change lenses, the LC-GETPUT law replaces GETPUT as the lower-bound of the putback and PUTGET is kept as the upper-bound. In fact, LC-GETPUT is the dual of LC-PUTGET, as it states that, if a source a (with view b) is one of the closest to a_0, then it must be returned by $\mathsf{Put}_f\ (a_0, b)$ (possibly among other non-minimal updates). This duality will become clear in the next section. It is also worth noting that LC-GETPUT forces Put_f to be defined for every pair (a_0, b) such that b is the view of some source a, i.e., to be safe (Section 3.2.2).

Returning to transformation $\mathtt{tw}_2$ under preorders $\sqsubseteq^{TW2}$ (cf. Figure 5.3b), when a new person p_2 is added to the $TW3$ view, only an exhaustive well-behaved $\mathsf{Put}_{\mathtt{tw}_2}$ that returns models with all possible popularities for p_2 may guarantee that the subsequent $\mathsf{Put}_{\mathtt{tw}_1}$ will be able to generate minimal source updates by selecting the one that assigns popularity 0 to p_2.

5.1.2 Reasoning about Least-change Lenses

In the sequel we use the point-free subset of relational logic to formally specify and reason about the two least-change lens formalizations, paving the way to easily transcribing properties found about one of them to the other.

To ease calculations, we use a *curried* version of the putback transformations $\widehat{\mathsf{put}}\ f : A \to (B \leftrightarrow A)$. Transcribing PUTGET and GETPUT to this version makes it even more clear that they establish lower- and upper-bounds for the putback for every

original source $a_0 \in A$:

$$\widehat{\mathsf{put}_f}\ a_0 \subseteq \mathsf{get}_f^\circ \qquad \text{PUTGET}$$
$$\rho\underline{a_0} \circ \mathsf{get}_f^\circ \subseteq \widehat{\mathsf{put}_f}\ a_0 \qquad \text{GETPUT}$$

where $\rho\underline{a_0} = \{(a_0, a_0)\}$ filters out values other than a_0, restricting the output of get_f° to a_0. As $\rho\underline{a_0}$ is at most the identity, such bounds are consistent—the lower-bound $\rho\underline{a_0} \circ \mathsf{get}_f^\circ$ is smaller than the upper-bound get_f°.

The principle of least-change, formalized by the LC-PUTGET and LC-GETPUT laws, can be encoded in terms of the so-called *shrink* operator $R \upharpoonright S : A \leftrightarrow B$ proposed by Mu and Oliveira (2011) that minimizes the output of a binary relation $R : A \leftrightarrow B$ in regard to another binary relation $S : B \leftrightarrow B$, and defined as follows:

$$\langle a, b\rangle \in (R \upharpoonright S) \equiv \langle a, b\rangle \in R \wedge (\forall\ b' : B \mid \langle a, b'\rangle \in R \Rightarrow \langle b, b'\rangle \in S)$$

In relational notation this equals defining $R \upharpoonright S = R \cap S \mathbin{/} R^\circ$, where $b \in S \mathbin{/} R\ a$ denotes relational *division* $\forall\ c : C \mid a \in R\ c \Rightarrow b \in S\ c$. So $R \upharpoonright S$ is *at most* R and its output is a *minimum* in regard to S. Using the shrink operator, the least-change laws proposed for selective and exhaustive lc-lenses can be specified as follows, for every $a_0 \in A$:

$$\widehat{\mathsf{put}_f}\ a_0 \subseteq \mathsf{get}_f^\circ \upharpoonright \preceq_{a_0} \qquad \text{LC-PUTGET}$$
$$\mathsf{get}_f^\circ \upharpoonright \preceq_{a_0} \subseteq \widehat{\mathsf{Put}_f}\ a_0 \qquad \text{LC-GETPUT}$$

Clearly, these laws are dual of each other. Moreover, it can be shown that they refine the regular laws in the sense that LC-PUTGET lowers the upper-bound imposed by PUTGET and LC-GETPUT raises the lower-bound imposed by GETPUT.

Proposition 5.1. *Given a family of stable total preorders* $\preceq : A \to (A \leftrightarrow A)$*, for every* $a_0 \in A$*,*

$$\rho\underline{a_0} \circ \mathsf{get}_f^\circ \subseteq \mathsf{get}_f^\circ \upharpoonright \preceq_{a_0} \subseteq \mathsf{get}_f^\circ$$

Proof. The upper-bound is trivial since $\mathsf{get}_f^\circ \cap \preceq_{a_0} \mathbin{/} \mathsf{get}_f \subseteq \mathsf{get}_f^\circ$; for the lower-bound we have

$$\begin{aligned} &\rho\underline{a_0} \circ \mathsf{get}_f^\circ \subseteq \mathsf{get}_f^\circ \upharpoonright \preceq_{a_0} \\ \equiv\ &\{\text{-shrink definition -}\} \end{aligned}$$

$$
\begin{aligned}
&\rho\underline{a_0} \circ \mathsf{get}_f^\circ \subseteq \mathsf{get}_f^\circ \cap \preceq_{a_0} / \mathsf{get}. \\
&\equiv \quad \{\text{-}\cap\text{-UNIVERSAL ; } /\text{-DEF -}\} \\
&\rho\underline{a_0} \circ \mathsf{get}_f^\circ \subseteq \mathsf{get}_f^\circ \wedge \rho\underline{a_0} \circ \mathsf{get}_f^\circ \circ \mathsf{get}_f \subseteq \preceq_{a_0} \\
&\equiv \quad \{\text{-}\rho\underline{a_0} = \underline{a_0} \circ \underline{a_0}^\circ \text{ ; SHUNTING -}\} \\
&\underline{a_0}^\circ \circ \mathsf{get}_f^\circ \circ \mathsf{get}_f \subseteq \underline{a_0}^\circ \circ \preceq_{a_0} \\
&\equiv \quad \{\text{-ORDSTABLE thus } \underline{a_0}^\circ \circ \preceq_{a_0} = \top \text{ -}\} \\
&\underline{a_0}^\circ \circ \mathsf{get}_f^\circ \circ \mathsf{get}_f \subseteq \top
\end{aligned}
$$

□

Note how ORDSTABLE is required for this proof. This was expected, since the lower-bound imposed by GETPUT restricts the result of put_f for the original source a_0, and ORDSTABLE states precisely that a_0 is a minimum of the preorder. This proposition also shows that LC-GETPUT and LC-PUTGET are consistent bounds.

Whenever $\Delta\ a_0$ is injective, for any $a_0 \in A$, (resulting in an antisymmetric $\preceq_{a_0} = [\Delta\ a_0]$, and thus a total order) or get_f is injective, $\mathsf{get}_f^\circ \upharpoonright \preceq_{a_0}$ is simple[3] and so will $\widehat{\mathsf{put}_f}\ a_0$. Moreover:

Proposition 5.2. *If* $\mathsf{get}_f^\circ \upharpoonright \preceq_{a_0}$ *is simple for every* $a_0 \in A$, $\widehat{\mathsf{put}_f}\ a_0 = \mathsf{get}_f^\circ \upharpoonright \preceq_{a_0}$ *is the only total backward transformation that gives rise to a well-behaved selective lc-lens. It is also the smallest* Put_f *that gives rise to a well-behaved exhaustive lc-lens.*

Proof. Trivial, since $\preceq_{a_0}$ is reflexive and shrinking simple relations by reflexive orderings has no effect (Mu and Oliveira, 2011). □

Since GETPUT is a requirement for lc-lenses and PUTGET follows from Proposition 5.1, every well-behaved selective lc-lens is a well-behaved regular lens, as should be expected:

$$\frac{\Gamma \vdash f : A_{\preceq}\ \rhd\ B}{\Gamma \vdash f : A\ \rhd\ B}$$

For every well-behaved regular lens $f : A\ \rhd\ B$ there is at least one family of stable total preorders $\preceq : A \to (A \leftrightarrow A)$ under which f can be interpreted as a well-behaved selective lc-lens: that which emerges from the discrete metric Δ^0, under which LC-PUTGET degenerates into PUTGET. Thus:

$$\frac{\Gamma \vdash f : A\ \rhd\ B}{\Gamma \vdash f : A_{[\Delta^0]}\ \rhd\ B} \tag{5.1}$$

[3] Since antisymmetric shrinking criteria ensure simplicity (Mu and Oliveira, 2011).

Finally, any well-behaved exhaustive regular lens can also be seen as a well-behaved exhaustive lc-lens by assigning a preorder such that $\mathsf{get}_f^\circ \restriction \preceq_{a_0}$ is simple and equal to Put_f. For instance, for any natural k, the distance function

$$\Delta^f\ (a, a_0) = \begin{cases} 0 & \text{if } a = \mathsf{Put}_f\ a_0\ (\mathsf{get}_f\ a) \\ k & \text{otherwise} \end{cases}$$

results, for every $a_0 \in A$, in a total preorder such that

$$a'\ [\Delta^f\ a_0]\ a \equiv (a \in \widehat{\mathsf{Put}_f}\ a_0\ (\mathsf{get}_f\ a) \Rightarrow a' \in \widehat{\mathsf{Put}_f}\ a_0\ (\mathsf{get}_f\ a'))$$

LC-GETPUT follows easily, thus:

$$\frac{\Gamma \vdash f : A \blacktriangleright B}{\Gamma \vdash f : A_{[\Delta^f]} \blacktriangleright B}$$

5.2 Criteria for Composing Least-change Lenses

This section explores under which conditions does the composition of two well-behaved least-change lenses $f : A_{\preceq} \rhd B$ and $g : B_{\sqsubseteq} \rhd C$ result in another well-behaved least-change lens $g \circ f : A_{\preceq} \rhd C$. In the curried putback version, the transformations comprising the regular lens composition combinator ($\circ$) (Foster et al., 2007) take the shape, for any original source $a_0 \in A$:

$$\begin{aligned} \mathsf{get}_{g \circ f} &= \mathsf{get}_g \circ \mathsf{get}_f \\ \widehat{\mathsf{put}_{g \circ f}}\ a_0 &= (\widehat{\mathsf{put}_f}\ a_0) \circ (\widehat{\mathsf{put}_g}\ (\mathsf{get}_f\ a_0)) \end{aligned}$$

In general least-change is not preserved by composition, since $\mathsf{put}_{g \circ f}$ is not guaranteed to return minimal values. This has already been demonstrated for our running example, under the update propagation example depicted in Figure 5.3b with preorders $\sqsubseteq^{TW2'}$ over $TW2$. Consider the depicted update on $TW3$ that inserts a new person p_2. Clearly, minimal updates on $TW1$ regarding $\mathsf{put}_{\mathsf{tw}_2 \circ \mathsf{tw}_1}$ are those that insert person p_2 in $TW1$ with a single follower (it would not show up in $TW3$ otherwise). However, with $\sqsubseteq^{TW2'}$, $\mathsf{put}_{\mathsf{tw}_2}$ may assign an arbitrary popularity to p_2 and still be regarded as minimal, denying $\mathsf{put}_{\mathsf{tw}_1}$ the ability to generate minimal $TW1$ updates in the process.

5.2.1 Selective Composition

We will see that in order to preserve LC-PUTGET in the selective scenario, the forward transformations will need to somehow preserve the preorders between view and source elements. To be more precise, the goal of this section is to analyze under which conditions LC-PUTGET is preserved by composition, i.e.:

$$(\forall\ a : A \mid \widehat{\mathsf{put}_f}\ a \subseteq \mathsf{get}_f^\circ \upharpoonright \preceq_a) \wedge (\forall\ b : B \mid \widehat{\mathsf{put}_g}\ b \subseteq \mathsf{get}_g^\circ \upharpoonright \sqsubseteq_b) \Rightarrow \\ (\forall\ a : A \mid \widehat{\mathsf{put}_{g \circ f}}\ a \subseteq \mathsf{get}_g \circ \mathsf{get}_f^\circ \upharpoonright \preceq_a)$$

In other words, if for every execution, put_f and put_g either fail or produce a single minimal update, will $\mathsf{put}_{g \circ f}$ be able to do the same? It is worth recalling that in a combinatorial frameworks, the well-behavedness of the composed should emerge solely from the well-behavedness of the opaque "sub-lenses" and thus the compositionality criteria should be intrinsic to each "sub-lens", rather than depend on their relationship.

The simplest condition for this to hold is for get_g to be injective: since get_g° will be simple, there will be a single valid put_g, and thus there is no ambiguity in the choice of propagated updates.

Proposition 5.3. *If* $f : A_\preceq \rhd B$ *and* $g : B_\sqsubseteq \rhd C$ *are well-behaved selective lc-lenses and* get_g *is* injective*, then* $g \circ f : A_\preceq \rhd C$ *is a* well-behaved *selective lc-lens.*

Having an injective get_f, however, is not enough to preserve LC-PUTGET. Still, a sufficient condition on get_f is for it to be *strictly increasing* between the preorders $\preceq_{a_0}$ and $\sqsubseteq_{(\mathsf{get}_f\ a_0)}$, for any $a_0 \in A$:

$$a_1 \prec_{a_0} a_2 \Rightarrow (\mathsf{get}_f\ a_1) \sqsubset_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_2)$$

We will abuse the notation and say that such transformations are strictly increasing between the families of preorders $\preceq$ and $\sqsubseteq$.

A strictly increasing get_f reads: if a source a_1 is smaller than another source a_2 (regarding the distance to the original source a_0), then its view $\mathsf{get}_f\ a_1$ shall be smaller than the view of a_2 (regarding the distance to the original view $\mathsf{get}_f\ a_0$). In point-free notation this can be expressed as follows:

$$\prec_{a_0} \subseteq \mathsf{get}_f^\circ \circ \sqsubset_{(\mathsf{get}_f\ a_0)} \circ \mathsf{get}_f \qquad \text{STRICTINC}$$

In total preorders, we have $a_1 \prec_{a_0} a_2 \equiv \neg(a_2 \preceq_{a_0} a_1)$, thus an equivalent formulation is:

$$(\mathsf{get}_f\ a_1) \sqsubseteq_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_2) \Rightarrow a_1 \preceq_{a_0} a_2$$

This property is similar to that enforced by "order-based" frameworks (Hegner, 2004), but over relative, rather than absolute, orders.

Proposition 5.4. *If* $f : A_{\preceq} \rhd B$ *and* $g : B_{\sqsubseteq} \rhd C$ *are well-behaved selective lc-lenses and* get_f *is* strictly increasing *between the families of preorders* $\preceq$ *and* $\sqsubseteq$*, then* $g \circ f : A_{\preceq} \rhd C$ *is a* well-behaved *selective lc-lens.*

Unfortunately this requirement is too restrictive, as it forces get_f to be injective, meaning that it cannot abstract information from the source.

To help understand why composition succeeds when get_f is strictly increasing, Figure 5.4 shows a possible configuration under this assumption. On the left subfigure, several B views spread over concentric circles according to their distance to a hypothetical center denoting $\mathsf{get}_f\ a_0$ (and represented by a star). Similarly for the right subfigure, for A sources and center a_0. Elements a^{-1} denote the pre-image of an element a under get_f. In this case, we have an one-to-one correspondence since get_f, being strictly increasing, must be injective. Another consequence of this property is that all B views that are at the same distance from $\mathsf{get}_f\ a_0$ (e.g., b_1, b_2 and b_3) must have their pre-images at the same distance from a_0 (b_1^{-1}, b_2^{-1} and b_3^{-1}, respectively). Moreover, the relative distances to the center are preserved, in the sense that, for example, since $b_1 \sqsubseteq_{(\mathsf{get}_f\ a_0)} b_4$ then $b_1^{-1} \preceq_{a_0} b_4^{-1}$. Now, consider the backward propagation performed by a hypothetical lc-lens g, when putting back a given view to the original source $\mathsf{get}_f\ a_0$ such that all the named elements are acceptable source updates. Being well-behaved, put_g will return one of the values that is closest to $\mathsf{get}_f\ a_0$, in this case either b_1, b_2 or b_3; put_f must return one of their pre-images which, due to the above assumptions, is guaranteed to be one of the closest to a_0, whatever choice is made by put_g. This renders the composed lens $g \circ f$ well-behaved under least-change.

While a strictly increasing get_f suffices to guarantee that any composition $g \circ f$ is also a well-behaved least-change lens, in order to attain a proper compositional language and be able to compose any sequence of transformations, this property must also be preserved by composition.

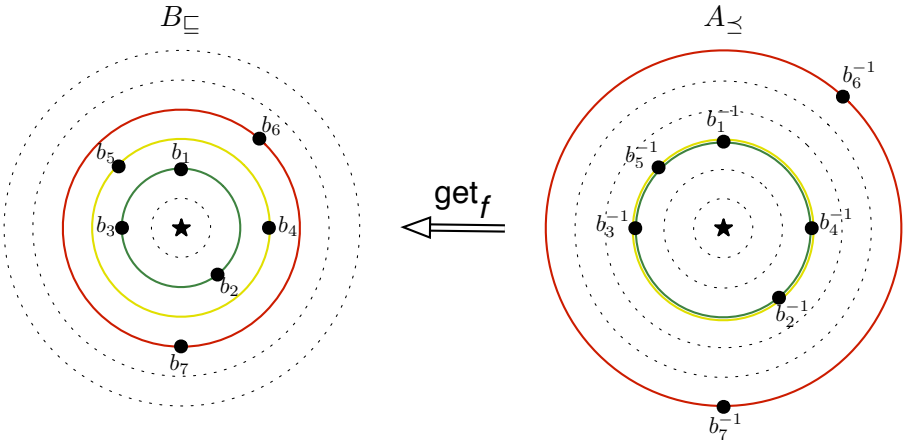


Figure 5.4: Strictly increasing transformation.

Proposition 5.5. *For two transformation* $\mathsf{get}_f : A_{\preceq} \rightharpoonup B_{\sqsubseteq}$ *and* $\mathsf{get}_g : B_{\sqsubseteq} \rightharpoonup C_{\precsim}$ *such that* get_f *is* strictly increasing *between the families of preorders* $\preceq$ *and* $\sqsubseteq$ *and* get_g *is* strictly increasing *between the families of preorders* $\sqsubseteq$ *and* $\precsim$*, then* $\mathsf{get}_g \circ \mathsf{get}_f : A_{\preceq} \rightharpoonup C_{\precsim}$ *is* strictly increasing *between* $\preceq$ *and* $\precsim$.

A more interesting requirement is the following more relaxed version of StrictInc, that does not imply injectivity:

$$(\forall\ a_2' : A \mid \mathsf{get}_f\ a_2 = \mathsf{get}_f\ a_2' \Rightarrow a_1 \prec_{a_0} a_2') \Rightarrow (\mathsf{get}_f\ a_1) \sqsubset_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_2)$$

In point-free notation this is written

$$\prec_{a_0} / (\mathsf{get}_f^{\circ} \circ \mathsf{get}_f) \subseteq \mathsf{get}_f^{\circ} \circ \sqsubset_{(\mathsf{get}_f\ a_0)} \circ \mathsf{get}_f \qquad \text{QuasiStrictInc}$$

Under quasi strictly increasing transformations, a view $\mathsf{get}_f\ a_1$ is required to be closer to $\mathsf{get}_f\ a_0$ than another view $\mathsf{get}_f\ a_2$ only when a_1 is closer to a_0 than all other sources a_2' that also map to the view of a_2. An alternative formulation using the "less than or equal" preorders is

$$(\mathsf{get}_f\ a_2) \sqsubseteq_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_1) \Rightarrow (\exists\ a_2' : A \mid \mathsf{get}_f\ a_2 = \mathsf{get}_f\ a_2' \wedge a_2' \preceq_{a_0} a_1)$$

or $\rho(\mathsf{get}_f) \circ \sqsubseteq_{(\mathsf{get}_f\ a_0)} \circ \mathsf{get}_f \subseteq \mathsf{get}_f \circ \preceq_{a_0}$ in the point-free notation. In other words, when a view b_2 is smaller than or equal to another view b_1 (regarding the distance to a view $\mathsf{get}_f\ a_0$), then at least one source with view b_2 is smaller than or equal to all the sources with view b_1 (regarding the distance to a source a_0). The difference

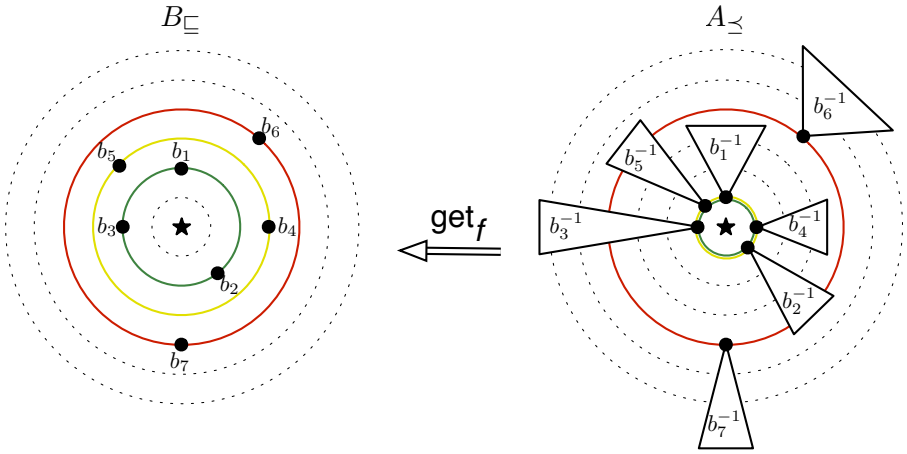


Figure 5.5: Quasi strictly increasing transformation.

to StrictInc can easily be seen in Figure 5.5. Now the pre-image of a view value can contain several sources at different distances from the center. Still, for views at the same distance to $\mathsf{get}_f\ a_0$ (e.g., b_1, b_2 and b_3), the minimum values of their pre-images must all be at the same distance from a_0 (minimal values of b_1^{-1}, b_2^{-1} and b_3^{-1}, respectively).

Proposition 5.6. *If* $f : A_{\preceq} \rhd B$ *and* $g : B_{\sqsubseteq} \rhd C$ *are well-behaved selective lc-lenses and* get_f *is* quasi strictly increasing *between the families of preorders* $\preceq$ *and* $\sqsubseteq$*, then* $g \circ f : A_{\preceq} \rhd C$ *is a* well-behaved *selective lc-lens.*

Figure 5.5 also helps understanding why this proposition holds. Again, when propagating an update whose acceptable sources are the named elements, put_g is free to choose any value closest to $\mathsf{get}_f\ a_0$ and, whatever the choice made, put_f will always be able to return one of the sources closest to a_0 from their pre-images. An example of a transformation that is not strictly increasing but is quasi strictly increasing is determining the size of a set, with the obvious metrics (size of symmetric difference in the source and absolute value of the difference in the view).

Unlike with strictly increasing transformations, the composition of two quasi strictly increasing transformations only remains so if get_f is surjective.

Proposition 5.7. *For two transformation* $\mathsf{get}_f : A_{\preceq} \rightharpoonup B_{\sqsubseteq}$ *and* $\mathsf{get}_g : B_{\sqsubseteq} \rightharpoonup C_{\precsim}$ *such that* get_f *is* surjective *and* quasi strictly increasing *between the families of preorders* $\preceq$ *and* $\sqsubseteq$ *and* get_g *is* quasi strictly increasing *between the families of preorders* $\sqsubseteq$ *and* $\precsim$*, then* $\mathsf{get}_g \circ \mathsf{get}_f : A_{\preceq} \rightharpoonup C_{\precsim}$ *is* quasi strictly increasing *between* $\preceq$ *and* $\precsim$*.*

In a lens framework, requiring surjectivity is a typical restriction, as already extensively discussed in Chapter 4.

An interesting class of lenses is the one obtained from transformations with a *perfect complement*. A function cpl_f is said to be a complement of get_f if $\mathsf{get}_f \vartriangle \mathsf{cpl}_f$ is injective, i.e., any source can be losslessly represented as a view/complement pair. For view update translation under a constant complement, there is a unique source update for each view update (Bancilhon and Spyratos, 1981). A complement is said to be *independent* (Keller and Ullman, 1984) when $\mathsf{get}_f \vartriangle \mathsf{cpl}_f$ is bijective, i.e., any view/complement pair corresponds to a source state. If a get_f has an independent complement (we call it *perfect*), then there exists a lens with a unique optimal put_f with perfect updatability, since it is possible to translate all view updates onto source updates while keeping the complement constant. Typical examples of transformations that fall under this category are tuple projections, whose perfect complement is exactly the information projected out.

Having a perfect complement is not sufficient for a least-change lens to be composable. However, if get_f is strictly increasing among sources with the same complement, then it is also quasi strictly increasing, and thus can safely be composed.

Proposition 5.8. *If* $\mathsf{get}_f : A \rightharpoonup B$ *has* perfect complement *and*

$$(\mathsf{get}_f\ a_1) \sqsubseteq_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_2) \wedge \mathsf{cpl}_f\ a_1 = \mathsf{cpl}_f\ a_2 \Rightarrow a_1 \preceq_{a_0} a_2$$

for every $a_0, a_1, a_2 \in A$*, then* get_f *is* quasi strictly increasing *between the families of preorders* $\preceq$ *and* $\sqsubseteq$*.*

Proving that a transformation is quasi strictly increasing is not trivial. This is an example of a useful class of transformations where such proof can be much simplified.

5.2.2 Exhaustive Composition

Dually to the selective case, the necessary condition for two exhaustive least-change lenses to be composable is the following:

$$\begin{aligned}&(\forall\ a : A \mid \mathsf{get}_f^\circ \upharpoonright \preceq_a \subseteq \widehat{\mathsf{Put}_f}\ a) \wedge (\forall\ b : B \mid \mathsf{get}_g^\circ \upharpoonright \preceq_b \subseteq \widehat{\mathsf{Put}_g}\ b) \Rightarrow \\ &\quad (\forall\ a : A \mid \mathsf{get}_g \circ \mathsf{get}_f^\circ \upharpoonright \preceq_a \subseteq \widehat{\mathsf{Put}_{g \circ f}}\ a)\end{aligned}$$

That is, if Put_f and Put_g produce at least all minimal updates, will $\mathsf{Put}_{g \circ f}$ be able to do the same?

Once again, an injective get_g guarantees least-change compositionality. However, it requires get_f to be total, in order to guarantee that $\mathsf{Put}_{g \circ f}$ is safe.

Proposition 5.9. *If* $f : A_{\preceq} \blacktriangleright B$ *and* $g : B_{\sqsubseteq} \blacktriangleright C$ *are well-behaved exhaustive lc-lenses,* get_f *is* total *and* get_g *is* injective*, then* $g \circ f : A_{\preceq} \blacktriangleright C$ *is a* well-behaved *exhaustive lc-lens.*

Interestingly, the remaining laws that ensure exhaustive compositionality are dual to those for the selective case: we can either reverse every law from $R \subseteq S$ to $S \subseteq R$ or, equivalently, replace every $\prec_a$ by $\preceq_a$. For example, instead of requiring get_f to be strictly increasing it suffices to require it to be *monotonic*:

$$a_1 \preceq_{a_0} a_2 \Rightarrow (\mathsf{get}_f\ a_1) \sqsubseteq_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_2)$$

Remember that monotonicity reads the same way as the strictly increasing property, by simply replacing "smaller than" by "smaller than or equal to" in the text. An equivalent formulation in the opposite direction is:

$$(\mathsf{get}_f\ a_1) \sqsubset_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_2) \Rightarrow a_1 \prec_{a_0} a_2$$

In the point-free notation we have

$$\preceq_{a_0} \subseteq \mathsf{get}_f^{\circ} \circ \sqsubseteq_{(\mathsf{get}_f\ a_0)} \circ \mathsf{get}_f \qquad \textsc{Monot}$$

Unlike the strictly increasing property, monotonicity no longer implies that get_f is injective; instead, it implies the dual property, that get_f is total (entailing a safe $\mathsf{Put}_{g \circ f}$).

Proposition 5.10. *If* $f : A_{\preceq} \blacktriangleright B$ *and* $g : B_{\sqsubseteq} \blacktriangleright C$ *are well-behaved exhaustive lc-lenses and* get_f *is* surjective *and* monotonic *between the families of preorders* $\preceq$ *and* $\sqsubseteq$*, then* $g \circ f : A_{\preceq} \blacktriangleright C$ *is a* well-behaved *exhaustive lc-lens.*

Unlike in Proposition 5.4, get_f is now required to be surjective (note that surjectivity is the dual property of simplicity, which get_f always satisfies). Figure 5.6 demonstrates the differences between being strictly increasing and monotonic, and why this property ensures a well-behaved exhaustive composition. Unlike in Figure 5.4, the pre-images of the B views at the same distance from $\mathsf{get}_f\ a_0$ (e.g., b_1 and b_2) are allowed to be at different distances from a_0 (elements from b_1^{-1} and b_2^{-1}, respectively). However, we have the restriction that, for example, when $b_1 \sqsubset_{(\mathsf{get}_f\ a_0)} b_4$ then $b_1^{-1} \prec_{a_0} b_4^{-1}$, forcing the pre-images of views at different distances from $\mathsf{get}_f\ a_0$ to also be at different distances from a_0 (respecting the relative orders). This ensures that when Put_g returns all values

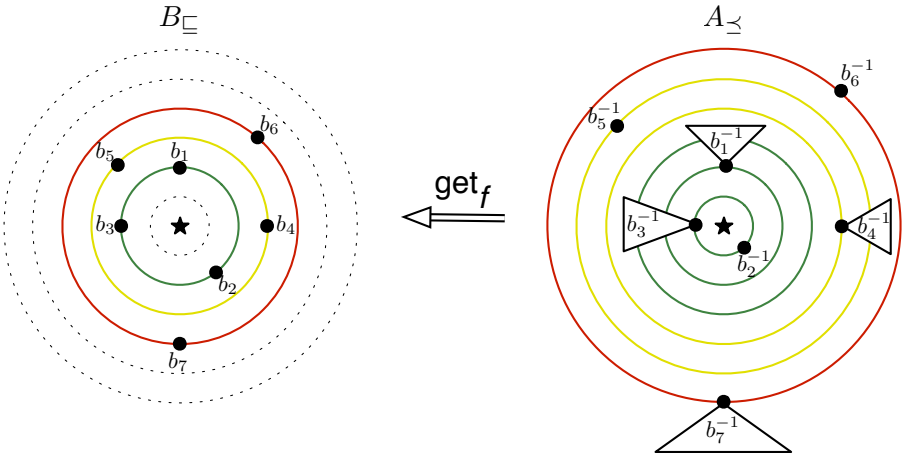


Figure 5.6: Monotonic transformation.

at the minimum distance from $\mathsf{get}_f\ a_0$, all sources at the minimum distance from a_0 are contained in their pre-image. This example also helps understanding why exhaustive lc-lenses were defined to return *at least* all minimal values rather than *exactly* all minimal values: even if the "sub-lenses" return exactly the minimal values (b_1, b_2 and b_3), the composed lens will produce values other than the minimals (those at the pre-images of b_1^{-1} and b_3^{-1}).

An interesting fact is that, unlike in total partial-orders, a strictly increasing transformation is not necessarily monotonic. This means that we can can have transformations whose composition is well-behaved in the selective scenario, but not well-behaved in the exhaustive one. Figure 5.4 helps understanding why: even if Put_g returns a value closest to $\mathsf{get}_f\ a_0$, (e.g., b_3) there is no chance for Put_f to return all values closest to a_0, since some of the pre-images closest to a_0 originate from views at minimal distance from $\mathsf{get}_f\ a_0$ (namely b_2).

As with strictly increasing transformations, monotonicity is preserved by composition.

Proposition 5.11. *For two transformations* $\mathsf{get}_f : A \rightharpoonup B$ *and* $\mathsf{get}_g : B \rightharpoonup C$ *such that* get_f *is* monotonic *between the families of preorders* $\preceq$ *and* $\sqsubseteq$ *and* get_g *is* monotonic *between the families of preorders* $\sqsubseteq$ *and* $\precsim$*, then* $\mathsf{get}_g \circ \mathsf{get}_f : A \rightharpoonup C$ *is* monotonic *between* $\preceq$ *and* $\precsim$.

Again, monotonicity can be relaxed to

$$(\forall\ a_2' : A \mid \mathsf{get}_f\ a_2 = \mathsf{get}_f\ a_2' \Rightarrow a_1 \preceq_{a_0} a_2') \Rightarrow (\mathsf{get}_f\ a_1) \sqsubseteq_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_2)$$

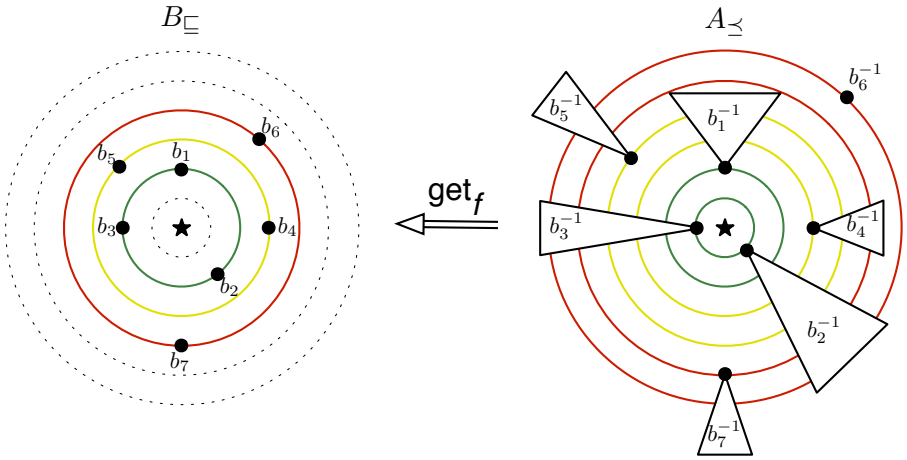


Figure 5.7: Quasi monotonic transformation.

or with the point-free notation:

$$\preceq_{a_0} / (\mathsf{get}_f^\circ \circ \mathsf{get}_f) \subseteq \mathsf{get}_f^\circ \circ \sqsubseteq_{(\mathsf{get}_f\ a_0)} \circ \mathsf{get}_f \qquad \text{QUASIMONOT}$$

In quasi monotonic transformations, if a source a_1 is smaller than or equal to all the sources with view b (regarding the distance to a source a_0), then its view $\mathsf{get}_f\ a_1$ shall be at most b (regarding the distance to a view $\mathsf{get}_f\ a_0$). Alternatively, when a view b_2 is smaller than another view b_1 (regarding the distance to a view $\mathsf{get}_f\ a_0$), then at least one source with view b_2 is smaller than all sources with view b_1 (regarding the distance to a source a_0):

$$(\mathsf{get}_f\ a_2) \sqsubset_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_1) \Rightarrow (\exists\ a_2' : A \mid \mathsf{get}_f\ a_2 = \mathsf{get}_f\ a_2' \Rightarrow a_2' \prec_{a_0} a_1)$$

Like monotonicity, quasi monotonicity entails a total get_f.

Proposition 5.12. *If* $f : A_{\preceq}$ ▶ B *and* $g : B_{\sqsubseteq}$ ▶ C *are well-behaved exhaustive lc-lenses and* get_f *is* surjective *and* quasi monotonic *between the families of preorders* $\preceq$ *and* $\sqsubseteq$*, then* $g \circ f : A_{\preceq}$ ▶ C *is a* well-behaved *exhaustive lc-lens.*

Figure 5.7 illustrates this property. The difference to monotonicity is that only the minimum pre-images of each view are required to respect the relative orders, for instance, even though b_4 is farther from $\mathsf{get}_f\ a_0$ than b_2, some elements from b_4^{-1} may be closer to a_0 than elements from b_1^{-1} other than the minimal. This suffices for Put_f to able to return all closest sources to a_0 given all closest views to $\mathsf{get}_f\ a_0$.

Likewise to quasi strictly increasing transformations, preservation of quasi monotonicity by composition requires the surjectivity of the first transformation.

Proposition 5.13. *For two transformation* $\mathsf{get}_f : A_{\preceq} \rightharpoonup B_{\sqsubseteq}$ *and* $\mathsf{get}_g : B_{\sqsubseteq} \rightharpoonup C_{\precsim}$ *such that* get_f *is* surjective *and* quasi monotonic *between the families of preorders* $\preceq$ *and* $\sqsubseteq$ *and* get_g *is* quasi monotonic *between the families of preorders* $\sqsubseteq$ *and* $\precsim$*, then* $\mathsf{get}_g \circ \mathsf{get}_f : A_{\preceq} \rightharpoonup C_{\precsim}$ *is* quasi monotonic *between* $\preceq$ *and* $\precsim$.

Finally, regarding lenses with perfect complement, exhaustive least-change compositionality is ensured by a monotonic get_f among sources with the same complement, since this ensures that get_f is quasi monotonic.

Proposition 5.14. *If* $\mathsf{get}_f : A \rightharpoonup B$ *has* perfect complement *and*

$$a_1 \preceq_{a_0} a_2 \wedge \mathsf{cpl}_f\ a_1 = \mathsf{cpl}_f\ a_2 \Rightarrow (\mathsf{get}_f\ a_1) \sqsubseteq_{(\mathsf{get}_f\ a_0)} (\mathsf{get}_f\ a_2)$$

for every $a_0, a_1, a_2 \in A$*, then* get_f *is* quasi monotonic *between the families of preorders* $\preceq$ *and* $\sqsubseteq$.

5.3 Discussion

This chapter introduced a framework of least-change lenses that is characterized by two dual scenarios: a single-valued one that only requires transformations to selectively return minimal updates, and a multi-valued one that requires them to exhaustively return all minimal updates. Several sufficient conditions that enable composition of least-change lenses were also presented (summarized in Tables 5.1 and 5.2), an issue not addressed in the pioneering work by Meertens (1998), where no linguistic mechanisms to combine maintainers following least-change, including composition, are proposed. The properties for well-behavedness and for composition arose naturally from the duality between both scenarios.

We have modeled our framework in Alloy, which has proved very useful in the early stages of this research to rapidly explore and verify/discard different propositions[4]. Of course, such automatic verification is necessarily bounded, and full unbounded calculational proofs for all our propositions were then conducted in the Isabelle/HOL (Nipkow

[4]The Alloy model can be downloaded from http://wiki.di.uminho.pt/twiki/pub/Research/FATBIT/Publications/lc-lenses.als.

$f : A_{\preceq} \rhd B$	$g : B_{\sqsubseteq} \rhd C$
	get_g injective
get_f strictly increasing	
get_f quasi strictly increasing	
get_f with perfect complement and strictly increasing over complement	

Table 5.1: Compositionality criteria for selective lc-lenses $g \circ f : A_{\preceq} \rhd C$.

$f : A_{\preceq} \blacktriangleright B$	$g : B_{\sqsubseteq} \blacktriangleright C$
get_f total	get_g injective
get_f surjective and monotonic	
get_f surjective and quasi monotonic	
get_f with perfect complement and monotonic over complement	

Table 5.2: Compositionality criteria for exhaustive lc-lenses $g \circ f : A_{\preceq} \blacktriangleright C$.

et al., 2012) proof assistant[5]. Our criteria are still not general enough to encompass least-change lenses we believe can be composed, although more relaxed criteria will most likely not be independent of the "sub-lenses", denying the advantages of a truly compositional approach. This issue raises the interesting question of completeness: is there a limit on the expressiveness of a least-change lens that can be composed safely with any other such lens? Our (still unproved) conjecture is that such limit is set precisely at quasi strictly increasing transformations for the selective scenario, and quasi monotonic transformations for the exhaustive one.

We advocate the principle of least-change as a tool to trim down the variability inherent in a bidirectional transformation to reasonable bounds while still supporting precise and predictable bidirectional laws. In this sense, least-change lenses and the invariant-constrained lenses explored in Chapter 4 complement each other, both providing means to tame the unpredictability inherent to the regular lens laws: the latter allows the restriction of the acceptable updates, while the former enforces an order on their selection. It remains to be seen however, how the combinators besides sequential composition behave under least-change.

In contrast with traditional bidirectional transformation approaches, whose criteria

[5]The Isabelle/HOL theory can be downloaded from http://wiki.di.uminho.pt/twiki/pub/Research/FATBIT/Publications/lc-lenses.thy.

for update translation (besides the regular well-behavedness laws) are usually vague or non-existent, least-change bidirectional transformations come equipped with an order on sources plus least-change well-behavedness laws that constitute a formal and explicit documentation of the criteria used for update translation. Therefore, reimplementing existing bidirectional transformation approaches in our framework could bring to light their underlying update translation semantics. Under this perspective, the composition of least-change lenses could be more flexible, by allowing orders over the datatypes to differ from lc-lens to lc-lens: one would write $f : A \rhd_{\preceq} B$ to state that lens f is well-behaved for order $\preceq$. By doing so, the order could specify more precisely the behavior of a lens, and given two lc-lenses $f : A \rhd_{\preceq} B$ and $B : B \rhd_{\sqsubseteq} C$, an interesting research direction is to infer an order $\precsim$ such that $g \circ f : A \rhd_{\precsim} C$ is a well-behaved lc-lens. Obviously, one could always default to the discrete metric Δ^0 following 5.1, but the goal would be to derive a more precise (i.e., more predictive) preorder. This resembles the properties explored in Chapter 4 over ic-lenses, where well-behavedness was preserved by "type-changing" combinators. In fact, in ic-lenses—unlike in lc-lenses—assessing whether well-behavedness is preserved by combinators is a rather trivial problem, and has already been explored in the context of regular lens (Foster et al., 2007).

Other approaches to minimal update propagation that do not rely on least-change (Larson and Sheth, 1991; Keller, 1986) consider whole get_f functions and do not allow reasoning by composition. The minimization criteria of the technique by Larson and Sheth (1991) can be seen as an order on sources and their algorithms as an exhaustive least-change lens, whereas the dialog is an additional mechanism to allow choosing a minimal source from the various possible. Buneman et al. (2002) study the complexity of minimizing view updates for a monotonic fragment of relational algebra, for two kinds of deletion and annotation updates, and conclude that this problem is NP-hard for queries including both projection-join or join-union. They consider two metrics for minimality: first the number of changes in the source, and second the number of side effects in the view caused by the view update. Although our notion of order is more natural with the former metric (since the lens laws disallow view side effects), our framework of lc-lenses is general in the sense that it does not assume any particular order on states nor language of updates. Also, while they study the problem of inverting an individual lens in a minimal way, we consider the orthogonal problem of composing "minimized" lenses.

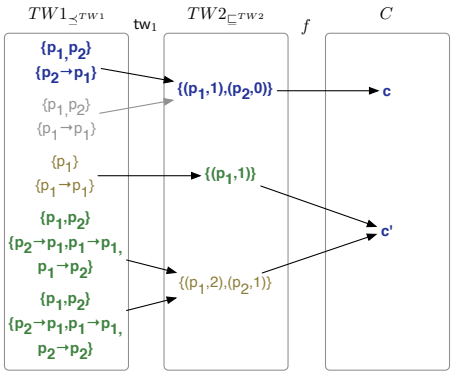


Figure 5.8: $\mathtt{tw}_1 : TW1_{\preceq^{TW1}} \rhd TW2$ failing QUASIMONOT and QUASISTRICTINC for $f : TW2_{\sqsubseteq^{TW2}} \rhd C$.

Like already posited in Section 4.4, allowing putbacks to be multi-valued does not compromise the predictability of the system (as would non-deterministic transformations): users are no longer required to rely on transformations with possibly opaque or contrived semantics, instead being given the chance to control the produced updates. Nonetheless, multi-valued composition rises obvious efficiency issues and it is not clear whether techniques similar to the invariant propagation from Section 4.2 could be developed for least-change. However, having the putback of f searching which intermediate candidates produced by the putback of g will allow it to generate minimal sources, is probably still better than trying to discover the least-changed sources of the composed transformation directly, since for reasonable orders the guidance from the inner transformations will greatly limit the search space (indeed, it allows the construction of a search tree by composition). A more serious issue is that the proposed criteria are rather tedious (and unintuitive) to verify and it is still not clear how better proof methods to perform such task could be developed. For example, we expected our example transformation $\mathtt{tw}_1$ to be quasi monotonic (with distance function Δ^{TW1}), but have found that such is not the case. An example is presented in Figure 5.8[6]. We are currently investigating whether other subtle redefinitions of Δ^{TW1} and Δ^{TW2}, namely using distance functions that give rise to lexicographic orders, would satisfy any of the criteria.

[6]Least-change lens $\mathtt{tw}_1$ fails to compose with a hypothetical f due to not being quasi monotonic nor quasi strictly increasing: when view c is updated to c', neither having $\mathtt{put}_f$ select a single minimal update allows $\mathtt{put}_{\mathtt{tw}_1}$ to select a minimal update—the selective scenario—nor having $\mathtt{Put}_f$ select all minimal updates allows $\mathtt{Put}_{\mathtt{tw}_1}$ to select all minimal updates—the exhaustive scenario.

Part III

Maintainer Framework

Chapter 6

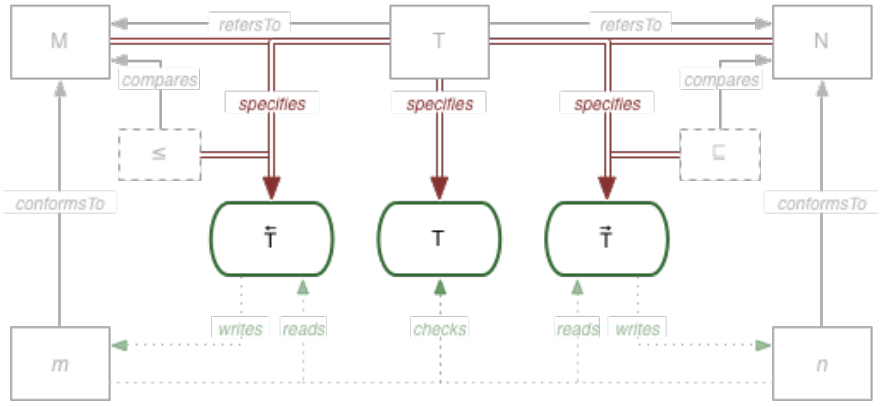
Maintaining Constraints

Throughout this dissertation we have been emphasizing the importance of bidirectional transformations that are aware of both domain constraints and minimality criteria. In Part II we tackled these issues under the combinatorial setting of lenses, with a particular focus on their sequential composition. To render these truly combinatorial—i.e., to guarantee that well-behavedness is preserved by the combinators—complex update propagation mechanisms were proposed that may render such approaches unfeasible: invariant-constrained lenses require type-inference mechanisms that may not be decidable, while exhaustive least-change lenses require putbacks to generate every acceptable element so that the succeeding putbacks may search for an appropriate one. The core concept in combinatorial approaches is the creation of complex, correct-by-construction bidirectional transformations from a set of simple primitives and combinators; if deploying these combinators is impractical, the combinatorial principle is undermined.

In fact, this kind of issues is typical in more general frameworks, like that of constraint maintainers. Compositional reasoning over constraint maintainers was deemed unsuitable by Meertens (1998, p. 42) from the start precisely because testing the composition of consistency relations—which in general are not simple—may yield an undecidable algorithm. The tradeoff is that constraint maintainers are more expressive than lens frameworks, being able to handle the symmetric scenario where neither transformation domain is a view of the other. As a consequence, constraint maintainers typically embody global constraints defined over the system, their comprising transformations being responsible for restoring its consistency when needed. In this part of the dissertation we explore the potential of such perspective, specifying bidirectional transformations by a constraint over the state of the system and disregarding their

combination into more complex artifacts. Since the asymmetric restriction imposed by lenses—of having a functional forward transformation—is no longer relevant, we are also able to focus on bidirectional transformations whose consistency relation is of arbitrary multiplicity.

The main hindrance in the development of constraint maintainer frameworks is that the three components of $T : M \bowtie N$ must be derived from a single specification of T: the consistency checker $T : M \leftrightarrow N$ as well as the two consistency-restoring transformations[1] $\overrightarrow{T} : M \times N \rightharpoonup N$ and $\overleftarrow{T} : M \times N \rightharpoonup M$, taking into consideration both the constraints imposed by the meta-models and the provided preference orders so that least-change may be enforced, as depicted in the following diagram:



This leads to restrictions over the specification language in order to render the derivation procedure feasible, as well as to derived transformations that are not sufficiently flexible to be applicable in many scenarios. As a consequence, very few effective instantiations of the constraint maintainer scheme exist. For instance, the QVT-R specification language by OMG (2011a) is one of the best known instantiation of constraint maintainers where consistency relations are specified as declarative relations that establish when two models are consistent. Yet, in order to attain consistency-restoring transformations that are effective, relations must be written in such way that almost resembles an imperative procedure (OMG, 2011a, p. 18). This jeopardizes the advantages of writing declarative constraints, as the user must think in an imperative paradigm. Most alternative implementations of QVT-R provide transformations that are not really bidirectional (Boronat et al., 2006; de Lara and Guerra, 2009), ignoring the initial models, and even the QVT-R standard (OMG, 2011a) only retrieves such information if explicit keys for the models

[1]These components are not necessarily embodied in artifacts, but may instead represent update procedures.

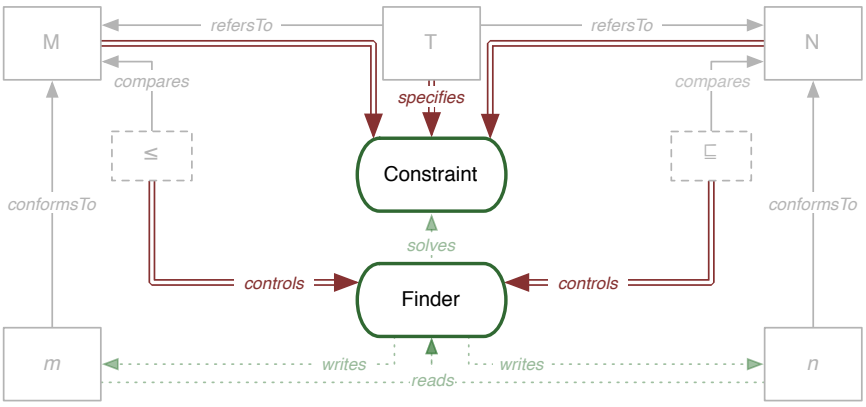
are defined. Introducing an additional complexity layer, transformations should also be aware of the domain constraints, especially in the context of MDE, where meta-model constraints abound. Failing to do so results in procedures that are simply not useful. Such is the case of the semantics proposed for QVT-R (OMG, 2011a) as well as tools that aim at implementing it. As far as we are aware, least-change semantics is not considered by any available implementation. Another widely study instantiation of constraint maintainers is that of *Triple Graph Grammars* (TGGs). However, these focus on the simultaneous application of updates, being more suitable for batch transformation scenarios (although some work has been developed on the incremental execution of TGGs (Ehrig et al., 2007; Giese and Wagner, 2009)).

To avoid these issues, in this chapter we see such frameworks under a different perspective. Recalling the constraint maintainer laws for $\overrightarrow{T}$, we have:

$$n \in \overrightarrow{T}\ (m_0, n_0) \Rightarrow T\ (m_0, n) \qquad \text{CORRECT}$$

$$T\ (m_0, n_0) \Rightarrow n_0 = \overrightarrow{T}\ (m_0, n_0) \qquad \text{HIPPOCRATIC}$$

The first states that transformations should restore consistency between the models, and the second that consistent models should not be updated; in both, the consistency relation is at the core of the transformations' behavior. In fact, the goal of the transformations is *repairing* the inter-model constraint entailed by the maintainer when inconsistencies are introduced, while also taking into consideration intra-model constraints imposed by the meta-models. Under this interpretation, we could deploy constraint maintainers according to the following architecture:



Unlike the above approach where actual transformations must be derived, the burden of this approach is on the embedding of the specification artifacts into a unifying logical

formalism. Consistent solutions could then be generated by relying on *model finding* procedures. The JTL framework proposed by Cicchetti et al. (2010) follows such approach, embedding constraints in a solver and asking the system to find consistent solutions. Yet, it lacks the ability to control the proposed updates, which may result in solutions with little resemblance with the original target model. Consider the following definition of least-change for $\overrightarrow{T}$, given preorders $\sqsubseteq$ over N:

$$n \in \overrightarrow{T}\ (m_0, n_0) \Rightarrow T\ (m_0, n) \wedge (\forall\ n' : N \mid T\ (m_0, n') \Rightarrow n \sqsubseteq_{n_0} n')$$

LC-CORRECT

Essentially, least-change entails a "biased selection" (Meertens, 1998) that selects minimal solutions from among those considered valid by the constraints. This hints that, rather than being embedded in the constraint, least-change should be deployed as a mechanism to control the solutions produced by the model finder.

The main research question of this chapter is: *can least-change constraint maintainers be deployed over model finding procedures?* To attain this, first there is the need to provide an embedding of the intra- and inter-model constraints into a specification language that is amenable to model finding; second, one needs to implement mechanisms that allow that same finder to provide least-change behavior. In the context of this dissertation, the unifying language over which constraints are defined is relational logic. Due to recent developments on *relational model finding*, such relational constraints may be solved by the Kodkod model finder developed by Torlak and Jackson (2007). By providing a higher-level specification language—arising from its support for relational logic and transitive closure—relational model finders have proven to be suitable to address MDE problems (Anastasakis et al., 2010; Kuhlmann and Gogolla, 2012; Cunha et al., 2013). In fact, the potential of Kodkod for intra-model consistency repair has previously been explored (Kleiner et al., 2010; Straeten et al., 2011) although the ability to produce minimal updates has only recently been tackled (Cunha et al., 2014).

Due to the general nature of the model finding problem, once the mechanism to enforce least-change is defined it can be employed to address other MDE tasks. In particular, we show that once the intra- and inter-model constraint is embedded in relational logic, model finding can be used to solve model repair, synchronization or multidirectional transformation problems.

The contributions and structure of this chapter are the following:

- we formalize the notion of constraint maintainers based on *model finding* (Sec-

tion 6.1) where the comprising transformations amount to consistency-restoring procedures;

- we show that such formalization can be generalized to address other MDE tasks (Section 6.2) like *model repair*, *synchronization* and *multidirectional transformation*;
- we explore how *least-change* behavior can be obtained from off-the-shelf model finders (Section 6.3).

Section 6.4 wraps up the chapter and discusses the results.

6.1 Constraint Maintaining with Model Finding

In the context of constraint maintainers, bidirectional transformations are represented by a global *constraint* over the environment, rather than a relational *expression* that consumes and produces elements (as with the lens frameworks from Part II). More specifically, while a lens $f : A \triangleright B$ transforms data between sorts A and B, a constraint maintainer $T : M \bowtie N$ is embodied by a formula that is evaluated for the current state of the transformation domains M and N. Thus, data is instead represented by relational variables occurring in the formula, whose valuations define the state of M and N. In fact, as discussed in Section 3.1, relational data is actually better suited to represent graph-like data structures. In this perspective, maintaining a constraint may be seen as a model finding problem, where the procedures look for variable assignments such that the intra- and inter-model constraints hold.

6.1.1 Model Finding

In the relational setting, free variables $\mathcal{R}$ represent relations, and their values consist of tuple sets drawn from universe $\mathcal{T}$. Much like predicates in standard predicate logic, these relations must have a uniform arity, and thus the tuple set assigned to a variable must be uniform, containing only tuples with the same arity. A transformation domain M introduces a set of relation variables, while models conforming to it consist of a binding for those variables, representing the current state of the model. Relational model finding consists precisely of finding a model binding $m : \mathcal{R} \rightharpoonup \mathcal{T}$ for the free relations for which a formula ϕ holds, i.e., $[\![\phi]\!]^m \equiv \mathsf{True}$. However, this problem is

in general undecidable, and thus, model finders must execute over a restricted search space by considering a bounded universe of atoms $\mathcal{A}$, rather than the whole universe of elements $\mathcal{U}$. To further control the search space, the range of acceptable values for each free variable can usually be constrained. In the relational setting, and in Kodkod in particular, these are embodied by lower- and upper-bounds for the tuple set assigned to each free variable, i.e., tuple sets $R_L, R_U : \mathcal{T}$ for a relation R such that $R_L \subseteq R_U$. In this context, a binding is only valid if the value assigned to each variable is within the given bounds.

Definition 6.1 (valid binding)**.** *A binding $m : \mathcal{R} \rightharpoonup \mathcal{T}$ is said to be* valid *under bound mapping $B : \mathcal{R} \rightharpoonup \mathcal{T} \times \mathcal{T}$ comprised of tuples $R \mapsto (R_L, R_U)$, denoted by $m : B$, if*

$$\forall\ R : \delta B \mid R_L \subseteq (m(R)) \subseteq R_U$$

The upper- and lower-bounds of a relation are also expected to have uniform arity, and thus force the same arity on its valuation. This also entails that, to be valid, a model binding must provide a valuation for each relation bound by B.

Given the interpretation of models as relation assignments, model finding consists of finding such bindings under the provided bounds.

Definition 6.2 (model finding)**.** *A* (relational) model finding *(MF) problem is a tuple $\langle \mathcal{A}, \phi, B \rangle$, where $\mathcal{A}$ is the universe of atoms, ϕ is a formula and $B : \mathcal{R} \rightharpoonup \mathcal{T} \times \mathcal{T}$ is a mapping from relations to bounds $R \mapsto (R_L, R_U)$ over $\mathcal{A}$ such that $\mathsf{fv}(\phi) \subseteq \delta B$. A model finding procedure $\llbracket \mathcal{A}, \phi, B \rrbracket : \mathbb{P}(\mathcal{R} \rightharpoonup \mathcal{T})$ is said to solve the MF problem if*

$$m \in \llbracket \mathcal{A}, \phi, B \rrbracket \equiv m : B \wedge [\![\phi]\!]^m$$

To be a solution of a model finding problem, a binding must be valid under bounds B and render formula ϕ true. The latter can be performed using standard semantics, like the one presented at Section 2.3. This definition entails that a valid solution must assign values to all variables bound by B, even if not occurring in ϕ. Moreover, it also assumes the model finding procedure returns every valid solution.

6.1.2 Embedding Constraints

As shown above, the core procedures of the proposed approach are the embedding of the MDE artifacts into solvable constraints; this section addresses precisely these

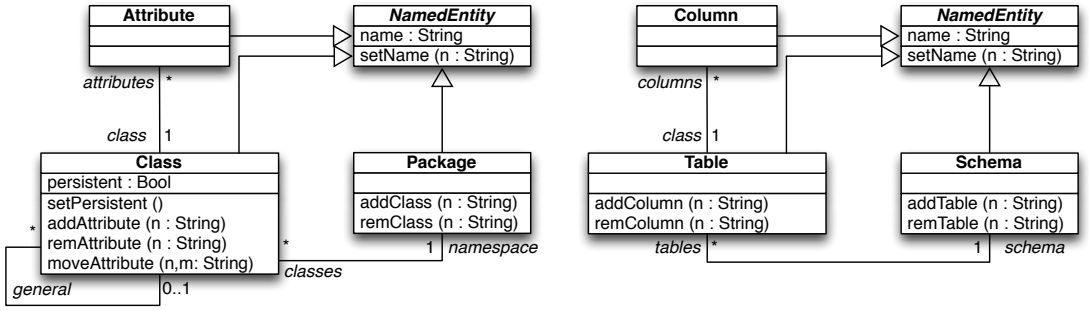
translations.

Intra-model constraints Back to the MDE level, each transformation domain M under meta-model $\mathtt{M}$ embedded in relational logic, introduces a set of sorts $\mathcal{S}_{\boldsymbol{M}}$ (representing the declared types), free relation variables $\mathcal{R}_{\boldsymbol{M}}$ bounded by the $\mathcal{S}_{\boldsymbol{M}}$ sorts (typed properties that represent the state of the model) and a formula $\phi_{\boldsymbol{M}}$ over those $\mathcal{R}_{\boldsymbol{M}}$ relations (the intra-model constraints that restrict the range of acceptable models): a binding $m : \mathcal{R} \rightharpoonup \mathcal{T}$ is said to conform to the transformation domain M if it binds the $\mathcal{R}_{\boldsymbol{M}}$ relations in such a way that $\phi_{\boldsymbol{M}}$ holds, i.e., $m : M \equiv [\![\phi_{\boldsymbol{M}}]\!]^m$. When processed in such a way, a transformation domain M is interpreted as a *typed relational constraint* $\boldsymbol{M}$, according to the following definition.

Definition 6.3 (typed relational constraint)**.** *A* typed relational constraint *(TRC)* $\boldsymbol{C}$ *is a tuple* $\langle \phi_{\boldsymbol{C}}, \mathcal{S}_{\boldsymbol{C}}, D_{\boldsymbol{C}} \rangle$*, where* $\phi_{\boldsymbol{C}}$ *is a formula,* $\mathcal{S}_{\boldsymbol{C}} \subseteq \mathcal{S}$ *a set of sorts and* $D_{\boldsymbol{C}}$ *a mapping from relations to type declarations* $R \mapsto A_1 \leftrightarrow ... \leftrightarrow A_n$*, with* $A_1, ..., A_n \in \mathcal{S}_{\boldsymbol{C}}$ *and* $\mathsf{fv}(\phi_{\boldsymbol{C}}) \subseteq \delta D_{\boldsymbol{C}}$*.*

The set of declared relation variables $\mathcal{R}_{\boldsymbol{C}}$ can be retrieved by $\delta D_{\boldsymbol{C}}$. Given a typed relational constraint $\boldsymbol{M}$, $\boldsymbol{M}^i$ shall denote another typed relational constraint that is equivalent to $\boldsymbol{M}$ modulo alpha-renaming, each variable having the index i appended. In practice, this duplicates relation variables while preserving the declared sorts. This is required to represent different transformation domains over the same meta-model $\mathtt{M}$ or different coexisting states of a transformation domain. A number of techniques have been proposed that allow the embedding of meta-models and their constraints into relational logic (Anastasakis et al., 2010; Kuhlmann and Gogolla, 2012; Cunha et al., 2013). Our embedding is based on the one proposed by Cunha et al. (2013) for UML models annotated with OCL constraints because, unlike other proposals, it covers an expressive OCL subset that includes closure and operation specifications via pre- and post-conditions (more details in Section 9.2).

As an example, consider a simplified version of the class diagrams $\mathtt{CD}$ and relational database schemas $\mathtt{DBS}$ meta-models, depicted at Figure 6.1. Figure 6.2 depicts a typed relational constraint $\boldsymbol{CD}$ that emerges from a transformation domain CD over $\mathtt{CD}$. The set of sorts $\mathcal{S}_{\boldsymbol{CD}}$ introduced denotes the classes declared within $\mathtt{CD}$, as well as the class hierarchy $\sqsubseteq_{\mathcal{S}}$. Mapping $D_{\boldsymbol{CD}}$ declares the free relations required to represent every $\mathtt{CD}$ model. Note the occurrence of relations $A_{\boldsymbol{CD}}$ for every sort A in $\mathcal{S}_{\boldsymbol{CD}}$: while the sort A denotes all elements of its type, $A_{\boldsymbol{CD}}$ represents only A elements present



(a) Meta-model `CD`. (b) Meta-model `DBS`.

Figure 6.1: Simplified class diagrams of the `CD` and `DBS` meta-models.

in each particular state of CD. This allows sorts to be shared among different typed relational constraints derived from the same meta-model, since a duplicated $\boldsymbol{CD}^1$ introduces new relations $A_{\boldsymbol{CD}^1}$ for each sort A. Formula $\phi_{\boldsymbol{CD}}$ introduces additional constraints over the bindings of $\mathcal{R}_{\boldsymbol{CD}}$. The first line represents the restriction on circular inheritance; the second models inheritance under relations $A_{\boldsymbol{CD}}$ (`NamedEntity` is abstract, so all elements within must belong to one of the sub-sorts); formulae from the third block restrict the value of the free relations to be defined within elements present in the particular CD instantiation; finally, the last block defines the multiplicity of the relations, following the classification defined in Section 2.4.

Inter-model constraints To be useful in the MDE context, constraint maintainers must also take into consideration the intra-model constraint imposed by the meta-models. Model finding problems are multi-valued, returning all valid solutions. Thus, deploying constraint maintainers as such procedures will result in an exhaustive framework.

Definition 6.4 (exhaustive invariant-constrained constraint maintainer)**.** *An exhaustive* well-behaved invariant-constrained constraint maintainer $T : M \bowtie N$ *is an exhaustive constraint maintainer such that following properties hold for* $\overrightarrow{T}$ *(and dually for* $\overleftarrow{T}$*):*

$$n \in \overrightarrow{T}\ (m_0, n_0) \Rightarrow\ T\ (m_0, n) \wedge n : N \qquad \text{CORRECT-INV}$$

$$T\ (m_0, n_0) \wedge m_0 : M \wedge n_0 : N \Rightarrow n_0 \in \overrightarrow{T}\ (m_0, n_0) \qquad \text{HIPPOCRATIC-INV}$$

$$T\ (m_0, n) \wedge m_0 : M \wedge n : N \wedge \neg T\ (m_0, n_0) \Rightarrow n \in \overrightarrow{T}\ (m_0, n_0) \qquad \text{EX-CORRECT-INV}$$

Given families of stable preorders $\preceq$ *and* $\sqsubseteq$ *over transformation domains* M *and* N*,*

$$
\begin{aligned}
\mathcal{S}_{\boldsymbol{CD}} = (&\{\mathtt{Package}, \mathtt{Class}, \mathtt{Attribute}, \mathtt{NamedEntity}, \mathtt{String}\},\\
&\{\mathtt{Package} \sqsubseteq_{\mathcal{S}} \mathtt{NamedEntity}, \mathtt{Class} \sqsubseteq_{\mathcal{S}} \mathtt{NamedEntity},\\
&\ \mathtt{Attribute} \sqsubseteq_{\mathcal{S}} \mathtt{NamedEntity}\})\\
D_{\boldsymbol{CD}} = \ &\{\mathtt{Package}_{\boldsymbol{CD}} : \mathbb{P}\,\mathtt{Package}, \mathtt{Class}_{\boldsymbol{CD}} : \mathbb{P}\,\mathtt{Class},\\
&\ \mathtt{Attribute}_{\boldsymbol{CD}} : \mathbb{P}\,\mathtt{Attribute}, \mathtt{NamedEntity}_{\boldsymbol{CD}} : \mathbb{P}\,\mathtt{NamedEntity},\\
&\ \mathtt{name}_{\boldsymbol{CD}} : \mathtt{NamedEntity} \leftrightarrow \mathtt{String}, \mathtt{classes}_{\boldsymbol{CD}} : \mathtt{Package} \leftrightarrow \mathtt{Class},\\
&\ \mathtt{attributes}_{\boldsymbol{CD}} : \mathtt{Class} \leftrightarrow \mathtt{Attribute}, \mathtt{general}_{\boldsymbol{CD}} : \mathtt{Class} \leftrightarrow \mathtt{Class},\\
&\ \mathtt{persistent}_{\boldsymbol{CD}} : \mathbb{P}\,\mathtt{Class}\}\\
\phi_{\boldsymbol{CD}} = \ &(\forall\, c : \mathtt{Class}_{\boldsymbol{CD}} \mid \neg(c \in \mathtt{general}^{+}_{\boldsymbol{CD}}\ c)) \wedge\\
&\mathtt{Package}_{\boldsymbol{CD}} \cup \mathtt{Class}_{\boldsymbol{CD}} \cup \mathtt{Attribute}_{\boldsymbol{CD}} = \mathtt{NamedEntity}_{\boldsymbol{CD}} \wedge\\
&\mathtt{name}_{\boldsymbol{CD}} \subseteq \mathtt{NamedEntity}_{\boldsymbol{CD}} \times \mathtt{String} \wedge\\
&\mathtt{classes}_{\boldsymbol{CD}} \subseteq \mathtt{Package}_{\boldsymbol{CD}} \times \mathtt{Class}_{\boldsymbol{CD}} \wedge\\
&\mathtt{attributes}_{\boldsymbol{CD}} \subseteq \mathtt{Class}_{\boldsymbol{CD}} \times \mathtt{Attribute}_{\boldsymbol{CD}} \wedge\\
&\mathtt{general}_{\boldsymbol{CD}} \subseteq \mathtt{Class}_{\boldsymbol{CD}} \times \mathtt{Class}_{\boldsymbol{CD}} \wedge\\
&\mathtt{persistent}_{\boldsymbol{CD}} \subseteq \mathtt{Class}_{\boldsymbol{CD}}\\
&\mathsf{img}\ \mathtt{name}_{\boldsymbol{CD}} \subseteq \mathsf{id} \wedge \mathsf{id} \subseteq \mathsf{ker}\ \mathtt{name}_{\boldsymbol{CD}}\\
&\mathsf{img}\ \mathtt{general}_{\boldsymbol{CD}} \subseteq \mathsf{id}\\
&\mathsf{ker}\ \mathtt{classes}_{\boldsymbol{CD}} \subseteq \mathsf{id} \wedge \mathsf{id} \subseteq \mathsf{img}\ \mathtt{classes}_{\boldsymbol{CD}}\\
&\mathsf{ker}\ \mathtt{attributes}_{\boldsymbol{CD}} \subseteq \mathsf{id} \wedge \mathsf{id} \subseteq \mathsf{img}\ \mathtt{attributes}_{\boldsymbol{CD}}
\end{aligned}
$$

Figure 6.2: Transformation domain CD as a TRC $\boldsymbol{CD}$.

respectively, it is an invariant-constrained least-change constraint maintainer $T : M_{\preceq} \blacktriangleright\!\blacktriangleleft N_{\sqsubseteq}$ *if* CORRECT-INV *and the following property hold for* $\overrightarrow{T}$ *(and dually for* $\overleftarrow{T}$*):*

$$
\begin{aligned}
&T\ (m_0, n) \wedge m_0 : M \wedge n : N \wedge (\forall\, n' : N \mid T\ (m_0, n') \Rightarrow n \sqsubseteq_{n_0} n') \Rightarrow\\
&\quad n \in \overrightarrow{T}\ (m_0, n_0) \qquad\qquad \text{LC-HIPPOCRATIC-INV}
\end{aligned}
$$

The inter-model constraint entailed by a constraint maintainer $T : M \blacktriangleright\!\blacktriangleleft N$ is also interpreted as a typed relational constraint $\boldsymbol{T}$, since it may introduce new relation variables into the model finding problem. Some of the relations from $D_{\boldsymbol{T}}$ are however expected to overlap with those of $D_{\boldsymbol{M}}$ and $D_{\boldsymbol{N}}$, as T should refer to the state of the transformation domains it is relating.

The deployment of a constraint maintainer $T : M \blacktriangleright\!\blacktriangleleft N$ must be aware of both intra- and inter-model constraints, and thus $\boldsymbol{M}$, $\boldsymbol{N}$ and $\boldsymbol{T}$ must be combined into a single

typed relational constraint[2]. Two typed relational constraints $\boldsymbol{C}_1 = \langle \phi_{\boldsymbol{C}_1}, \mathcal{S}_{\boldsymbol{C}_1}, D_{\boldsymbol{C}_1} \rangle$ and $\boldsymbol{C}_2 = \langle \phi_{\boldsymbol{C}_2}, \mathcal{S}_{\boldsymbol{C}_2}, D_{\boldsymbol{C}_2} \rangle$ can be combined if the shared relation declarations are consistent, i.e., they are assigned the same type ($\forall\ R : \delta D_{\boldsymbol{C}_1} \cap \delta D_{\boldsymbol{C}_2} \mid D_{\boldsymbol{C}_1}(R) = D_{\boldsymbol{C}_2}(R)$). In that case, they may be combined in the following way:

$$\langle \phi_{\boldsymbol{C}_1}, \mathcal{S}_{\boldsymbol{C}_1}, D_{\boldsymbol{C}_1} \rangle \ \curlywedge\ \langle \phi_{\boldsymbol{C}_2}, \mathcal{S}_{\boldsymbol{C}_2}, D_{\boldsymbol{C}_2} \rangle = \langle \phi_{\boldsymbol{C}_1} \wedge \phi_{\boldsymbol{C}_2}, \mathcal{S}_{\boldsymbol{C}_1} \cup \mathcal{S}_{\boldsymbol{C}_2}, D_{\boldsymbol{C}_1} \cup D_{\boldsymbol{C}_2} \rangle$$

Since model finding procedures require upper- and lower-bounds to be defined over each free variable, deploying typed relational constraint requires the derivation of bounds for each relation from their declared types. Since a bounded universe is also required, assigning to each sort in $\mathcal{S}_{\boldsymbol{C}}$ a finite scope will allow the derivation of the universe $\mathcal{A}$ as well as entail concrete upper-bounds for each declared relation. For a typed relational constraint $\boldsymbol{C}$, $\lfloor D_{\boldsymbol{C}} \rfloor^s$ shall denote the procedure that, given a fixed scope $s : \mathcal{S} \rightharpoonup \mathcal{T}$ for the defined sorts $\mathcal{S}_{\boldsymbol{C}}$, converts relation declarations into relation bounds, as

$$\lfloor D_{\boldsymbol{C}} \rfloor^s = \{\, R \mapsto (\emptyset, s(A_1) \times ... \times s(A_n)) \mid (R \mapsto A_1 \leftrightarrow ... \leftrightarrow A_n) \leftarrow D_{\boldsymbol{C}} \,\}$$

Now that the intra- and inter-model constraints are embedded into a model finding problem, given a relational model finder we are able to restore the consistency of the system. However, to embody a bidirectional transformation, an additional restriction over the model finding problem must be introduced, that of preserving the pre-state of the source domain. Recall that operation $\oplus : (A \rightharpoonup B) \to (A \rightharpoonup B) \to (A \rightharpoonup B)$ between mappings overrides shared keys with the value of the second, i.e., $\forall\ a : \delta R \cap \delta S \mid (R \oplus S)(a) = S(a)$. Thus, given a bound B and a binding m, expression $B \oplus (m \triangle m)$ replaces the bound of every relation R fixed by m to be the constant $m(R)$, i.e., $(B \oplus (m \triangle m))(R) = (m(R), m(R))$.

Proposition 6.1 (constraint maintaining as MF)**.** *Let $T : M \bowtie N$ be a well-behaved invariant-constrained constraint maintainer, such that $\langle \phi_{\bowtie}, \mathcal{S}_{\bowtie}, D_{\bowtie} \rangle = \boldsymbol{M} \curlywedge \boldsymbol{N} \curlywedge \boldsymbol{T}$, and $s : \mathcal{S} \rightharpoonup \mathcal{T}$ a concrete scope with $\delta s \subseteq \mathcal{S}_{\bowtie}$. For two valid model bindings $m : \lfloor D_{\boldsymbol{M}} \rfloor^s$ and $n : \lfloor D_{\boldsymbol{N}} \rfloor^s$, T can be deployed as a model finding procedure with:*

$$T\ (m, n) \equiv [\![\rho s, \phi_{\bowtie}, \lfloor D_{\bowtie} \rfloor^s \ \oplus\ ((m \cup n) \triangle (m \cup n))]\!] \neq \emptyset$$

[2]Unlike invariant-constrained lenses from Chapter 4, the transformation domains of invariant-constrained constraint maintainers are not appended with additional invariants because these are entailed by the associated meta-models.

$$\overrightarrow{T}\ (m,n) = \textbf{if}\ T\ (m,n)\ \textbf{then}\ n \\ \textbf{else}\ \mathcal{R}_{\boldsymbol{N}} \lessdot [\![\rho s, \phi_{\blacktriangleright\!\blacktriangleleft}, \lfloor D_{\blacktriangleright\!\blacktriangleleft} \rfloor^{s}\ \oplus\ (m \vartriangle m)]\!]$$

$$\overleftarrow{T}\ (m,n) = \textbf{if}\ T\ (m,n)\ \textbf{then}\ m \\ \textbf{else}\ \mathcal{R}_{\boldsymbol{M}} \lessdot [\![\rho s, \phi_{\blacktriangleright\!\blacktriangleleft}, \lfloor D_{\blacktriangleright\!\blacktriangleleft} \rfloor^{s}\ \oplus\ (n \vartriangle n)]\!]$$

Proof. Formula $\phi_{\blacktriangleright\!\blacktriangleleft}$ results from the combination of the TRCs from the constraint and the transformation domains. Thus, by definition of model finding (Definition 6.2), the defined problems always return solutions that intra- and inter-model consistent. For the $T\ (m,n)$ procedure, the valuation of the models is fixed to the pre-state models, thus it will return solutions if m and n are intra- and inter-model consistent. For the $\overrightarrow{T}\ (m,n)$ and $\overleftarrow{T}\ (m,n)$ procedures, CORRECT-INV holds because $\phi_{\blacktriangleright\!\blacktriangleleft}$ will hold for the solutions; HIPPOCRATIC-INV holds because the procedure explicitly tests if m and n are already consistent; EX-CORRECT-INV holds because model finding procedures return every solution (Definition 6.2). These update the expected model because the valuation of the source model is fixed to the valuation of its pre-state. Since the used formalism is many-sorted, the universal quantification from EX-CORRECT-INV is bounded by the elements of the sorts and thus the scope required by model finding problems does not affect the well-behavedness of the constraint maintainer. □

Checking the consistency also requires a model finding procedure because, although m and n bind every $\mathcal{R}_{\boldsymbol{M}}$ and $\mathcal{R}_{\boldsymbol{N}}$ variable, $\boldsymbol{T}$ may introduce free variables for which an assignment must be found. If no such variables are introduced by $\boldsymbol{T}$, then it can be simply run as the test $[\![\phi_{\blacktriangleright\!\blacktriangleleft}]\!]^{m \cup n}$, since input models m and n are assumed be within the provided scope. As for restoring the consistency, a model finding procedure is deployed with constant bounds for the variables of the source domain. Correctness is obviously guaranteed by this procedure, since the goal of model finding is precisely to solve $\phi_{\blacktriangleright\!\blacktriangleleft}$ and stability is achieved artificially since the finding procedure has no knowledge of the pre-state of target domain. Exhaustiveness is achieved due to the nature of model finding procedures. Since model finding assigns values to every relation, including those fixed, the solution must be filtered for the relations regarding the expected output model (e.g., filtering a binding with $\mathcal{R}_{\boldsymbol{M}} \lessdot m$ retrieves the assignments of variables related to $\boldsymbol{M}$).

6.1.3 Target-oriented Model Finding

Although constraint maintainers can be deployed over standard model finding procedures, these do not provide any minimality guarantees. Elsewhere (Cunha et al., 2014) we studied an extension to model finding that enables the definition of *targets* for each free relation variable, that denote the optimal solution for the model finding problem: solutions generated by target-oriented model finders must approximate the defined targets.

Definition 6.5 (target-oriented model finding)**.** *A* target-oriented (relational) model finding *(TO-MF) problem is a tuple* $\langle \mathcal{A}, \phi, B, T, \preceq \rangle$*, where* $\langle \mathcal{A}, \phi, B \rangle$ *is a model finding problem,* $T : \mathcal{R} \rightharpoonup \mathcal{T}$ *is a mapping from relations to targets* $R \mapsto R_T$ *within* $\mathcal{A}$ *with* $R_L \subseteq R_T \subseteq R_U$ *and* $\delta\, T \subseteq \delta B$*, and* $\preceq : (\mathcal{R} \rightharpoonup \mathcal{T}) \to ((\mathcal{R} \rightharpoonup \mathcal{T}) \leftrightarrow (\mathcal{R} \rightharpoonup \mathcal{T}))$ *a family of total preorders over tuple sets. A model finding procedure* $[\![\mathcal{A}, \phi, B, T]\!]_{\preceq} : \mathbb{P}(\mathcal{R} \rightharpoonup \mathcal{T})$ *is said to solve the problem if*

$$m \in [\![\mathcal{A}, \phi, B, T]\!]_{\preceq} \equiv m \in [\![\mathcal{A}, \phi, B]\!] \wedge (\forall\, m' : [\![\mathcal{A}, \phi, B]\!] \mid m \preceq_T m')$$

Like least-change, target-oriented model finding does not necessarily entail a single solution: depending on the metric, there might be multiple solutions at a minimal distance. We say that a solution is *optimal* if it matches all defined targets: if the preorders are stable (Definition 3.8), these are the only ones returned when valid. Note that not every relation variable bound by B must have a target assigned by T, in which case their valuation does not affect the target-oriented procedure. In this case there are multiple optimal solutions.

Least-change constraint maintainers can be deployed as target-oriented model finders by, besides fixing the state of the source domain, setting the pre-state of the target domain as the target of the model finding problem.

Proposition 6.2 (least-change constraint maintaining as TO-MF)**.** *Let* $T : M_{\preceq} \bowtie N_{\sqsubseteq}$ *be a well-behaved invariant-constrained least-change constraint maintainer, such that* $\langle \phi_{\bowtie}, \mathcal{S}_{\bowtie}, D_{\bowtie} \rangle = \boldsymbol{M} \curlywedge \boldsymbol{N} \curlywedge \boldsymbol{T}$ *and* $s : \mathcal{S} \rightharpoonup \mathcal{T}$ *a concrete scope with* $\delta s \subseteq \mathcal{S}_{\bowtie}$*. For two valid model bindings* $m : \lfloor D_{\boldsymbol{M}} \rfloor^s$ *and* $n : \lfloor D_{\boldsymbol{N}} \rfloor^s$*,* T *can be deployed as a target-oriented model finding procedure with:*

$$T\ (m, n) \equiv [\![\rho s, \phi_{\bowtie}, \lfloor D_{\bowtie} \rfloor^s \oplus ((m \cup n) \vartriangle (m \cup n))]\!] \neq \emptyset$$
$$\overrightarrow{T}\ (m, n) = \mathcal{R}_{\boldsymbol{N}} \lhd [\![\rho s, \phi_{\bowtie}, \lfloor D_{\bowtie} \rfloor^s \oplus (m \vartriangle m), n]\!]_{\sqsubseteq}$$

$$\overleftarrow{T}\ (m, n) = \mathcal{R}_{\boldsymbol{M}} \lessdot [\![\rho s, \phi_{\blacktriangleright\!\blacktriangleleft}, \lfloor D_{\blacktriangleright\!\blacktriangleleft} \rfloor^{s} \ \oplus\ (n \vartriangle n), m]\!]_{\preceq}$$

Proof. The procedures follows the same rationale as Proposition 6.1, but considering the pre-state of the target domain as the target of the problem. HIPPOCRATIC-INV no longer needs the explicit test since for stable preorders (as required by Definition 6.4), if the target is a solution, it will be the only one. LC-HIPPOCRATIC-INV holds due to the nature of target-oriented model finding (Definition 6.5). Again, the bounded nature of model finding does not affect LC-HIPPOCRATIC-INV because the universal quantification is itself bound by the valuation of the sorts. □

Now, besides fixing the bounds of the source model, the model finding procedure is given the bindings of the original target model as the target of the problem. Since the preorders are assumed to be stable, the stability of the transformation no longer has to be achieved in an artificial way: if m and n are already consistent, the model finding procedure will only return optimal solutions, whose $\mathcal{R}_{\boldsymbol{M}}$ and $\mathcal{R}_{\boldsymbol{N}}$ components are exactly m and n (only the valuation of the variables introduced by $\boldsymbol{T}$ may vary). LC-HIPPOCRATIC-INV holds because the model finding procedure will generate all valid solutions at minimal distance.

6.2 Beyond Bidirectional Transformation

While the focus of this dissertation is bidirectional transformation, deploying constraint maintainers as target-oriented procedures allows us to address other MDE tasks in a straight-forward manner. For instance, model repair, one of the most essential MDE tasks, can be directly deployed over model finding procedures. As presented in Section 6.1.2, a transformation domain M is essentially a typed relational constraint $\boldsymbol{M}$, where testing whether a model m is well-formed $m : M$ amounts to checking whether $[\![\phi_{\boldsymbol{M}}]\!]^{m}$ holds. From this interpretation, given a family of stable preorders $\preceq$ over M, an exhaustive *correct* and *least-change model repair* operation $M\uparrow : M \leftrightarrow M$ can be easily defined, for which the following laws shall hold:

$$m \in M\uparrow m_0 \Rightarrow m : M \qquad \text{M-CORRECT-INV}$$

$$m_0 : M \Rightarrow m_0 \in M\uparrow m_0 \qquad \text{M-HIPPOCRATIC-INV}$$

$$m : M \wedge \neg m_0 : M \Rightarrow m \in M\uparrow m_0 \qquad \text{M-EX-CORRECT-INV}$$

$$m : M \wedge (\forall\ m' : M \mid m \preceq_{m_0} m') \Rightarrow m \in M\uparrow m_0 \qquad \text{M-LC-HIPPOCRATIC-INV}$$

These laws are essentially those of constraint maintainers but restricted to a single model: repairs must generate consistent models, models that are already consistent shall not be modified, and whenever the initial model is inconsistent, all consistent repair models shall be returned. Model repair without least-change concerns would be even more problematic than constraint maintainers: in the latter the consistency with the opposite model must still be enforced, but in the former arbitrary conforming models are simply produced.

Assuming the need for least-change, deploying model repair as a target-oriented model finding procedure can be performed as follows.

Proposition 6.3 (least-change model repair as TO-MF)**.** *For a concrete scope $s : \mathcal{S} \rightharpoonup \mathcal{T}$ with $\delta s \subseteq \mathcal{S}_{\boldsymbol{M}}$, a valid model binding $m : \lfloor D_{\boldsymbol{M}} \rfloor^s$, can be repaired by the following target-oriented model finding procedure:*

$$M \uparrow m = \llbracket \rho s, \phi_{\boldsymbol{M}}, \lfloor D_{\boldsymbol{M}} \rfloor^s, m \rrbracket_{\preceq}$$

Proof. Formula $\phi_{\boldsymbol{M}}$ embodies the intra-model constraint, so the rationale is the same as Proposition 6.2 for a single transformation domain. □

The bindings of relations occurring in the original model m are set as the targets of the model finding problem, so that the generated models are as close as possible to m. Since the only relation variables in this problem are introduced by $\boldsymbol{M}$, the binding produced by the repair needs not be filtered. Figure 6.3 depicts a model repair problem that resulted from the embedding of the $\boldsymbol{CD}$ typed relational constraint from Figure 6.2 for the concrete scope depicted in Figure 6.4, given an initial model that contained cyclic inheritance (the tuples (C_1, C_2) and (C_2, C_1)). Figure 6.5 depicts the two solutions calculated by the target-oriented model finding procedure.

Model repair is a more concrete application scenario than bidirectional transformation. However, more general MDE tasks can also be formalized in a similar way. For instance, a problem related to bidirectional transformation is that of *model synchronization*, where, unlike in bidirectional transformation, both models are allowed to be updated in order to restore consistency.

Definition 6.6 (exhaustive synchronizer)**.** *An exhaustive* synchronizer *is comprised of a predicate $T : M \leftrightarrow N$ and a transformation $\overleftrightarrow{T} : M \times N \leftrightarrow M \times N$. It is said to*

$$
\begin{aligned}
\mathcal{A} &= \{P_1, C_1, C_2, C_3, A_1, A_2, \texttt{"Company"}, \texttt{"Employee"}, \texttt{"Person"}, \texttt{"Salary"}\} \\
B &= \{\texttt{NamedEntity}_{\boldsymbol{CD}} \mapsto (\emptyset, \{P_1, C_1, C_2, C_3, A_1, A_2\}), \\
&\quad \texttt{Package}_{\boldsymbol{CD}} \mapsto (\emptyset, \{P_1\}), \\
&\quad \texttt{Class}_{\boldsymbol{CD}} \mapsto (\emptyset, \{C_1, C_2, C_3\}), \\
&\quad \texttt{Attribute}_{\boldsymbol{CD}} \mapsto (\emptyset, \{A_1, A_2\}), \\
&\quad \texttt{name}_{\boldsymbol{CD}} \mapsto (\emptyset, \{(P_1, \texttt{"Company"}), (P_1, \texttt{"Employee"}), \\
&\qquad\qquad (P_1, \texttt{"Person"}), (P_1, \texttt{"Salary"}), \\
&\qquad\qquad (C_1, \texttt{"Company"}), (C_1, \texttt{"Employee"}), \ldots\}), \\
&\quad \texttt{classes}_{\boldsymbol{CD}} \mapsto (\emptyset, \{(P_1, C_1), (P_1, C_2), (P_1, C_3)\}), \\
&\quad \texttt{attributes}_{\boldsymbol{CD}} \mapsto (\emptyset, \{(C_1, A_1), (C_1, A_2), (C_2, A_1), \\
&\qquad\qquad (C_2, A_2), (C_3, A_1), (C_3, A_2)\}), \\
&\quad \texttt{general}_{\boldsymbol{CD}} \mapsto (\emptyset, \{(C_1, C_1), (C_1, C_2), (C_1, C_3), \\
&\qquad\qquad (C_2, C_1), (C_2, C_2), (C_2, C_3), \ldots\}), \\
&\quad \texttt{persistent}_{\boldsymbol{CD}} \mapsto (\emptyset, \{C_1, C_2, C_3\})\} \\
T &= \{\texttt{NamedEntity}_{\boldsymbol{CD}} \mapsto \{P_1, C_1, C_2, A_1\}, \\
&\quad \texttt{Package}_{\boldsymbol{CD}} \mapsto \{P_1\}, \\
&\quad \texttt{Class}_{\boldsymbol{CD}} \mapsto \{C_1, C_2\}, \\
&\quad \texttt{Attribute}_{\boldsymbol{CD}} \mapsto \{A_1\}, \\
&\quad \texttt{name}_{\boldsymbol{CD}} \mapsto \{(P_1, \texttt{"Company"}), (C_1, \texttt{"Person"}), \\
&\qquad\qquad (C_2, \texttt{"Employee"}), (A_1, \texttt{"salary"})\}, \\
&\quad \texttt{classes}_{\boldsymbol{CD}} \mapsto \{(P_1, C_1), (P_1, C_2)\}, \\
&\quad \texttt{attributes}_{\boldsymbol{CD}} \mapsto \{(C_2, A_1)\}, \\
&\quad \texttt{general}_{\boldsymbol{CD}} \mapsto \{(C_2, C_1), (C_1, C_2)\}, \\
&\quad \texttt{persistent}_{\boldsymbol{CD}} \mapsto \{C_2\}\} \\
\phi &= \phi_{\boldsymbol{CD}}
\end{aligned}
$$

Figure 6.3: TO-MF problem for a CD domain under the scope from Figure 6.4.

$$
\begin{aligned}
&\texttt{String} \mapsto \{\texttt{"Company"}, \texttt{"Employee"}, \texttt{"Person"}, \texttt{"Salary"}\} \\
&\texttt{NamedEntity} \mapsto \{P_1, C_1, C_2, C_3, A_1, A_2\} \\
&\texttt{Package} \mapsto \{P_1\} \\
&\texttt{Class} \mapsto \{C_1, C_2, C_3\} \\
&\texttt{Attribute} \mapsto \{A_1, A_2\}
\end{aligned}
$$

Figure 6.4: Concrete scope for a CD transformation domain.

$$
\begin{aligned}
m'_1 = \{&\texttt{NamedEntity}_{CD} \mapsto \{P_1, C_1, C_2, A_1\},\\
&\texttt{Package}_{CD} \mapsto \{P_1\},\\
&\texttt{Class}_{CD} \mapsto \{C_1, C_2\},\\
&\texttt{Attribute}_{CD} \mapsto \{A_1\},\\
&\texttt{name}_{CD} \mapsto \{(P_1, \texttt{"Company"}), (C_1, \texttt{"Person"}),\\
&\qquad (C_2, \texttt{"Employee"}), (A_1, \texttt{"salary"})\},\\
&\texttt{classes}_{CD} \mapsto \{(P_1, C_1), (P_1, C_2)\},\\
&\texttt{attributes}_{CD} \mapsto \{(C_2, A_1)\},\\
&\texttt{general}_{CD} \mapsto \{(C_2, C_1)\},\\
&\texttt{persistent}_{CD} \mapsto \{C_2\}\}\\
m'_2 = \{&\texttt{NamedEntity}_{CD} \mapsto \{P_1, C_1, C_2, A_1\},\\
&\texttt{Package}_{CD} \mapsto \{P_1\},\\
&\texttt{Class}_{CD} \mapsto \{C_1, C_2\},\\
&\texttt{Attribute}_{CD} \mapsto \{A_1\},\\
&\texttt{name}_{CD} \mapsto \{(P_1, \texttt{"Company"}), (C_1, \texttt{"Person"}),\\
&\qquad (C_2, \texttt{"Employee"}), (A_1, \texttt{"salary"})\},\\
&\texttt{classes}_{CD} \mapsto \{(P_1, C_1), (P_1, C_2)\},\\
&\texttt{attributes}_{CD} \mapsto \{(C_2, A_1)\},\\
&\texttt{general}_{CD} \mapsto \{(C_1, C_2)\},\\
&\texttt{persistent}_{CD} \mapsto \{C_2\}\}
\end{aligned}
$$

Figure 6.5: Solutions of the TO-MF problem from Figure 6.3.

be an exhaustive well-behaved invariant-constrained synchronizer $T : M \Leftrightarrow N$ *if the following properties hold:*

$$(m, n) \in \overleftrightarrow{T}\ (m_0, n_0) \Rightarrow T\ (m, n) \wedge m : M \wedge n : N \qquad \text{S-CORRECT-INV}$$

$$T\ (m_0, n_0) \wedge m_0 : M \wedge n_0 : N \Rightarrow (m_0, n_0) \in \overleftrightarrow{T}\ (m_0, n_0) \qquad \text{S-HIPPOCRATIC-INV}$$

$$T\ (m, n) \wedge m : M \wedge n : N \wedge \neg T\ (m_0, n_0) \Rightarrow (m, n) \in \overleftrightarrow{T}\ (m_0, n_0) \qquad \text{S-EX-CORRECT-INV}$$

Given families of stable preorders $\preceq$ and $\sqsubseteq$ over transformation domains M and N, respectively, it is an invariant-constrained least-change synchronizer $T : M_{\preceq} \Leftrightarrow N_{\sqsubseteq}$ if S-CORRECT-INV *and the following property hold:*

$$T\ (m, n) \wedge m : M \wedge n : N \wedge$$

$$(\forall\ m' : M, n' : N \mid T\ (m', n') \Rightarrow (m, n) \precsim_{(m_0, n_0)} (m', n')) \Rightarrow$$
$$(m, n) \in \overleftrightarrow{T}\ (m_0, n_0) \qquad \text{S-LC-HIPPOCRATIC-INV}$$

Where $\precsim$ results from the combination of the preorders $\preceq$ and $\sqsubseteq$ into a preorder over pairs $M \times N$.

The derivation of the product preorder $\precsim$ will be explored in Section 6.3. Unlike constraint maintainers, these are deployed as model finding procedures without fixed relations, but having the pre-state of both transformation domains set as the problem's targets.

Proposition 6.4 (least-change synchronization as TO-MF)**.** *Let $T : M_{\preceq} \Leftrightarrow N_{\sqsubseteq}$ be an invariant-constrained least-change synchronizer, such that $\langle \phi_{\Leftrightarrow}, \mathcal{S}_{\Leftrightarrow}, D_{\Leftrightarrow} \rangle = \boldsymbol{M} \curlywedge \boldsymbol{N} \curlywedge \boldsymbol{T}$ and $s : \mathcal{S} \rightharpoonup \mathcal{T}$ a concrete scope with $\delta s \subseteq \mathcal{S}_{\Leftrightarrow}$. For two valid model bindings $m : \lfloor D_{\boldsymbol{M}} \rfloor^s$ and $n : \lfloor D_{\boldsymbol{N}} \rfloor^s$, T can be deployed as a target-oriented model finding procedure as:*

$$T\ (m, n) \equiv [\![\rho s, \phi_{\Leftrightarrow}, \lfloor D_{\Leftrightarrow} \rfloor^s \oplus ((m \cup n) \vartriangle (m \cup n))]\!] \neq \emptyset$$
$$\overleftrightarrow{T}\ (m, n) = \textbf{let}\ mn' = [\![\rho s, \phi_{\Leftrightarrow}, \lfloor D_{\Leftrightarrow} \rfloor^s, m \cup n]\!]_{\precsim}$$
$$\textbf{in}\ (\mathcal{R}_{\boldsymbol{M}} \lessdot mn', \mathcal{R}_{\boldsymbol{M}} \lessdot mn')$$

Proof. Formula $\phi_{\Leftrightarrow}$ embodies the intra- and inter-model constraint, so the rationale is the same as Proposition 6.2, provided that no domain has fixed valuations and both pre-state models are set as targets of the problem. □

Finally, we can also generalize maintainers to relate multiple models rather than exactly two. This allows us to tackle the problem of *multidirectional transformation*, which thus far has been widely disregarded. Let $\sigma\, T$ denote the set of n transformation domains $\{ M_1, ..., M_n \}$ related by a multidirectional constraint maintainer and $\boldsymbol{m}$ denote the respective n models $m_1 \cup ... \cup m_n$. For a tuple of k selectors $i = (i_1, ..., i_k)$ with $i_j \in [1..n]$, $\sigma\, T_i$ shall denote the (ordered) selection of transformation domains $(M_{i_1}, ..., M_{i_k})$ and $\boldsymbol{m}_i$ the respective models $m_{i_1} \cup ... \cup m_{i_k}$.

Definition 6.7 (exhaustive multidirectional constraint maintainer)**.** *An exhaustive* well-behaved invariant-constrained multidirectional constraint maintainer $T : M_1 \bowtie ... \bowtie$

M_n is comprised of a predicate $T : M_1 \leftrightarrow ... \leftrightarrow M_n$ and a set of transformations $\overrightarrow{T}_{\sigma T_t} : M_1 \times ... \times M_n \leftrightarrow M_{t_1} \times ... \times M_{t_k}$, for any tuple $t = (t_1, ..., t_k)$ of domain indexes, such that the following properties hold:

$$\boldsymbol{m}_t \in \overrightarrow{T}_{\sigma T_t}\ \boldsymbol{m}_0 \Rightarrow \bigwedge_{i \in t} \boldsymbol{m}_{ti} : M_i \wedge T\ (\boldsymbol{m}_0\ \oplus\ \boldsymbol{m}_t) \qquad \text{X-CORRECT-INV}$$

$$T\ \boldsymbol{m}_0 \wedge \bigwedge_{i \in [1..n]} \boldsymbol{m}_{0i} : M_i \Rightarrow \boldsymbol{m}_0 \in \boldsymbol{m}_0\ \oplus\ (\overrightarrow{T}_{\sigma T_t}\ \boldsymbol{m}_0) \qquad \text{X-HIPPOCRATIC-INV}$$

$$T\ (\boldsymbol{m}_0\ \oplus\ \boldsymbol{m}_t) \wedge \bigwedge_{i \in [1..n]} (\boldsymbol{m}_0\ \oplus\ \boldsymbol{m}_t)_i : M_i \wedge \neg T\ \boldsymbol{m}_0 \Rightarrow \boldsymbol{m}_t \in \overrightarrow{T}_{\sigma T_t}\ \boldsymbol{m}_0$$

X-EX-CORRECT-INV

Given families of stable preorders $\preceq^1, ..., \preceq^n$ over transformation domains $M_1, ..., M_n$, respectively, it is an invariant-constrained least-change multidirectional constraint maintainer *$T : M_{1\preceq^1} \bowtie ... \bowtie M_{n\preceq^n}$ if* X-CORRECT-INV *and the following property hold:*

$$T\ (\boldsymbol{m}_0\ \oplus\ \boldsymbol{m}_t) \wedge \bigwedge_{i \in [1..n]} (\boldsymbol{m}_0\ \oplus\ \boldsymbol{m}_t)_i : M_i \wedge$$
$$(\forall\ \boldsymbol{m}'_t : \sigma\, T_t \mid T\ (\boldsymbol{m}_0\ \oplus\ \boldsymbol{m}'_t) \Rightarrow \boldsymbol{m}_t \preceq^{\sigma T_i}_{\boldsymbol{m}_0} \boldsymbol{m}'_t) \Rightarrow$$
$$\boldsymbol{m}_t \in \overrightarrow{T}_{\sigma T_t}\ \boldsymbol{m}_0 \qquad \text{X-LC-HIPPOCRATIC-INV}$$

Where $\preceq^{\sigma T_i}$ results from the combination of the preorders $\preceq^i$ into a preorder over tuples $\sigma\, T_i$.

The derivation of the tuple preorder $\preceq^{\sigma T_i}$ will be explored in Section 6.3. These laws extend those of bidirectional constraint maintainers. For transformations over models t, the updated $\boldsymbol{m}_t$ combined with the source models must be consistent; if the original models $\boldsymbol{m}_0$ are already consistent, then updating them should produce null updates.

Proposition 6.5 (least-change multidirectional constraint maintaining as TO-MF)**.** *Let $T : M_{1\preceq^1} \bowtie ... \bowtie M_{n\preceq^n}$ be an invariant-constrained least-change multidirectional constraint maintainer, such that $\langle \phi_{\bowtie}, \mathcal{S}_{\bowtie}, D_{\bowtie} \rangle = \boldsymbol{M}_1 \curlywedge ... \curlywedge \boldsymbol{M}_n \curlywedge \boldsymbol{T}$ and $s : \mathcal{S} \rightharpoonup \mathcal{T}$ a concrete scope with $\delta s \subseteq \mathcal{S}_{\bowtie}$. For n valid models bindings $m_i : \lfloor D_{\boldsymbol{M}_i} \rfloor^s$, T can be deployed as a target-oriented model finding procedure for any tuple $t = (t_1, ..., t_k)$ of domain indexes, as:*

$$
\begin{aligned}
T\ \boldsymbol{m} \quad &\equiv [\![\rho s, \phi_{\blacktriangleright\!\blacktriangleleft}, \lfloor D_{\blacktriangleright\!\blacktriangleleft} \rfloor^{s} \ \oplus\ (\boldsymbol{m} \vartriangle \boldsymbol{m})]\!] \neq \emptyset \\
\overrightarrow{T}_{\sigma T_t}\ \boldsymbol{m} &= \textbf{let}\ \boldsymbol{m}' = [\![\rho s, \phi_{\blacktriangleright\!\blacktriangleleft}, \lfloor D_{\blacktriangleright\!\blacktriangleleft} \rfloor^{s} \ \oplus\ (\boldsymbol{m}_{\bar{t}} \vartriangle \boldsymbol{m}_{\bar{t}}), \boldsymbol{m}_t]\!]_{(\preceq^{M_1 \times \ldots \times M_n})} \\
&\qquad \textbf{in}\ (\mathcal{R}_{\boldsymbol{M}_{t_1}} \lessdot \boldsymbol{m}', ..., \mathcal{R}_{\boldsymbol{M}_{t_k}} \lessdot \boldsymbol{m}')
\end{aligned}
$$

where $\boldsymbol{m}_{\bar{t}} = \boldsymbol{m} \setminus \boldsymbol{m}_t$.

Proof. Formula $\phi_{\blacktriangleright\!\blacktriangleleft}$ embodies the intra- and inter-model constraint, so the rationale is the same as Proposition 6.2, provided that the source domains are fixed to their pre-state valuations and the pre-state of the target domains are set as the target of the problem. □

Here, there is a set of source models—that must remain constant—and a set of k target models m_{t_i}—that must approximate the original ones.

6.3 Deploying Preference Orders

Our approach to constraint maintaining as model finding considers the definition of the model finding problem and the ability to produce minimal solutions as orthogonal concerns. Section 6.1 addressed the former; this section will address the latter. We start by exploring how to approximate targets under arbitrary metrics using off-the-shelf relational model finders (Kodkod in particular); then we show that for particular preference orders, least-change can be achieved by directly exploiting the functionalities of the SAT solvers that are behind relational model finding.

As initially presented in Chapter 3 (and extensively explored in Chapter 5 in the context of lens frameworks), the family of total stable preorders $\preceq : M \to (M \leftrightarrow M)$ can be attained by lifting a distance function $\Delta : M \times M \to \mathbb{N}$ that is also stable (Section 3.2.4). In this section we follow the same approach and focus on orders derived from distance functions. However, in the MDE constraint maintainer context, metamodels are first-class artifacts that are processed and embedded into the bidirectional transformation; this allows us to rely on automatically inferred distance functions, rather than defining them in an *ad hoc* way (as in Chapter 5).

6.3.1 Least-change as Iterative MF

In general, since relational model finders do not have native support for targets, target-oriented procedures must be defined by external means. The most direct method is to

iteratively call standard model finders to search for models at an increasing distance from the original one. Under this approach, for a distance d, the model finding procedure shall be called $d + 1$ times (the first run tests whether the models are already consistent, at distance 0).

Embedding distance functions In this context, a given distance function Δ over transformation domain M must be embedded in a typed relational constraint in order to be used in the model finding procedure. Let $\boldsymbol{M}^s$ and $\boldsymbol{M}^t$ be two copies of the $\boldsymbol{M}$ typed relational constraint modulo variable renaming. The embedding of Δ as a typed relational constraint, for a particular $d : \mathbb{N}$, is the typed relational constraint $\boldsymbol{\Delta}_d = \langle \phi^d_{\boldsymbol{\Delta}}, \mathcal{S}_{\boldsymbol{M}}, D_{\boldsymbol{M}^s} \cup D_{\boldsymbol{M}^t} \cup D^d_{\boldsymbol{\Delta}} \rangle$. If $\boldsymbol{\Delta}_d$ holds, then model $m' : M^t$ is at most at distance[3] d from $m : M^s$, i.e., for a scope $s : \mathcal{S} \rightharpoonup \mathcal{T}$,

$$\Delta\ (m, m')\ \leqslant\ d \Leftarrow \left[\!\left[\rho s, \phi^d_{\boldsymbol{\Delta}}, \lfloor D^d_{\boldsymbol{\Delta}} \rfloor^s\ \oplus\ ((m \cup m') \vartriangle (m \cup m')) \right]\!\right] \neq \emptyset$$

The metric may introduce free variables $D^d_{\boldsymbol{\Delta}}$, in which case m and m' will bind the variables concerning $\boldsymbol{M}^s$ and $\boldsymbol{M}^t$, while the remainder are left free for the finder to solve. It is important to note that the embedding of the distance function is *bounded*: even if two models are at distance d according to Δ, within a fixed scope it might not be possible to calculate that distance since variables $D^d_{\boldsymbol{\Delta}}$ are bounded (thus the implication on the formula).

Given this formulation, an iterative procedure over model finding can be deployed that iteratively calls $\boldsymbol{\Delta}_d$ until a solution is found. For model repair, this can be achieved by adapting Proposition 6.3 in the following way.

Proposition 6.6 (least-change model repair as iterative MF)**.** *For a concrete scope $s : \mathcal{S} \rightharpoonup \mathcal{T}$ with $\delta s \subseteq \mathcal{S}_{\boldsymbol{M}}$, a valid model binding $m : \lfloor D_{\boldsymbol{M}} \rfloor^s$, can be repaired by iteratively calling a model finding procedure*

$$M \uparrow m = \mathcal{R}_{\boldsymbol{M}} \lessdot \left[\!\left[\rho s, \phi_{\boldsymbol{M}} \wedge \phi^d_{\boldsymbol{\Delta}}\ [R_t \Rightarrow R], \lfloor D_{\boldsymbol{M}} \cup D^d_{\boldsymbol{\Delta}} \rfloor^s\ \oplus\ (m^s \vartriangle m^s) \right]\!\right]$$

from $d = 0$ until SAT, where $R_x \Rightarrow R_y$ replaces every variable $R_x \in \mathcal{R}_{\boldsymbol{M}^x}$ by the matching variable $R_y \in \mathcal{R}_{\boldsymbol{M}^y}$ and m^s renames every binding of R in m by one over R_s.

[3] Recall that distance functions are not necessarily symmetric.

Proof. Since $\phi_{\boldsymbol{M}}$ is still expected to hold, M-CORRECT-INV still holds; if the order is stable, then $\phi^0_{\boldsymbol{\Delta}}$ will only hold when the targets are solutions of the problem, guaranteeing M-HIPPOCRATIC-INV; M-HIPPOCRATIC-INV is guaranteed by the iterative nature of the process, that searches for solutions at increasing distance from the target. □

Essentially, the $\boldsymbol{M}^t$ variables from $\boldsymbol{\Delta}$ are replaced by the respective free variables that shall denote the solution of the problem, while those regarding $\boldsymbol{M}^s$ are assigned constant bounds that correspond to the valuation provided by the original model: this effectively tests the distance between the valuation calculated by the finder and the provided pre-state. The binding produced by the model finding procedure is filtered to retain only the variables related to the original problem, filtering out the portion of the binding concerning the variables introduced by $\boldsymbol{\Delta}_d$. The same technique could be applied to achieve least-change constraint maintainers by adapting Proposition 6.2 in a similar way.

It is important to note that this procedure is only least-change modulo the provided scope: the generated solutions represent the models closest to the target that are representable under the provided scope. Mechanisms can be developed to tame this issue, namely by increasing the universe of atoms at each iteration. This is further explored in Section 9.3.

Operation-based distance Instantiating the procedure just presented requires the definition of $\phi^d_{\boldsymbol{\Delta}}$ and any additional declarations $D^d_{\boldsymbol{\Delta}}$ for any distance d. One standard way to define the distance between two models is to define the set of valid edit operations and measure the number of required steps to reach one model from the other (Diskin et al., 2011). Let ϕ_U be a formula denoting the set of valid model updates between two typed relational constraint copies $\boldsymbol{M}^s$ and $\boldsymbol{M}^t$, i.e., $[\![\phi_U]\!]^{m \cup m'}$ holds if there is a valid update between models $m : M^s$ and $m' : M^t$. This formula can be lifted to the following distance function:

$$\Delta^U\ (m, m') = \begin{cases} 0 & \text{if } m = m' \\ d & \text{if } d \in U^*\ (m, m') \\ \infty & \text{otherwise} \end{cases}$$

where

$$d \in U^*\ (m, m') \equiv \exists\ m_1 : M^1, ..., m_{d-1} : M^{d-1}\ |$$

$$[\![(\phi_U\ [R_t \Mapsto R_1])]\!]^{m \cup m_1} \wedge ... \wedge$$
$$[\![(\phi_U\ [R_s \Mapsto R_i, R_t \Mapsto R_{i+1}])]\!]^{m_i \cup m_{i+1}} \wedge ... \wedge$$
$$[\![(\phi_U\ [R_s \Mapsto R_{d-1}])]\!]^{m_{d-1} \cup m'}$$

which denotes the fact there is an update path of length d from m to m', where each ϕ_U has its free variables replaced by those introduced by the intermediate states, except for R_s variables in the first step that denote the initial model $m : M^s$ and R_t in the last, denoting the target model M^t. If two models are at infinite distance, then there is no path of whatever length that connects them. This metric is stable and total, and thus gives rise to stable total preorders (although, it may not be symmetric, since updates are not necessarily undoable). This definition forces null updates to be always valid, otherwise the model finder would not be able to produce optimal solutions.

In this context, target-oriented model finding may be attained by minimizing the number of update steps from the target of the problem to a consistent model. However, since this metric requires higher-order quantifications to detect the update traces, it is not directly definable in the $\phi^d_{\mathbf{\Delta}^U}$ formula. Nonetheless, second-order existential quantifications can be converted to first-order problems through a process known as *skolemization*. A second-order existential quantification is skolemized by creating a free relation variable that represents it: proving the existential quantification amounts to providing a witness to the fresh free variable. In this case, it amounts to, at each iteration, create relation variables to allocate the intermediate model states, to which the finder must assign valid values:

$$D^d_{\mathbf{\Delta}^U} = D_{\boldsymbol{M}^s} \cup D_{\boldsymbol{M}^1} \cup ... \cup D_{\boldsymbol{M}^{d-1}}$$
$$\phi^d_{\mathbf{\Delta}^U} = \phi_U\ [R_t \Mapsto R_1] \wedge ... \wedge \phi_U\ [R_s \Mapsto R_i, R_t \Mapsto R_{i+1}] \wedge ... \wedge \phi_U\ [R_s \Mapsto R_{d-1}]$$

Each $\boldsymbol{M}^i$ denotes a new copy of $\boldsymbol{M}$ as a typed relational constraint, whose variables $\mathcal{R}_{\boldsymbol{M}^i}$ must be solved. Following Proposition 6.6, variables R_s will be bounded by the problem's targets while variables R_t will replaced by free variables representing the fresh model. The intermediate models are not required to be well-formed (constraints $\phi_{\boldsymbol{M}^i}$ are not included in the formula) meaning that the edit operations may temporarily introduce inconsistencies, as long as last model of the sequence is consistent.

A typical instantiation of ϕ_U arises from the set of valid edit operations that can be applied to a model. For a transformation domain M, this *operation-based distance*

(OBD) can be automatically inferred from the operations defined within its meta-model, and is denoted by Δ^U_M. This allows the user to control the repair of the models by specifying in the meta-model which edit operations can be applied to update conforming models. In MDA, these may be specified using pre- and post-conditions defined in (a subset of) OCL following the technique from (Cunha et al., 2013) (detailed in Section 9.2).

Consider that for the purposes of our running example, the methods whose interfaces are depicted in Figure 6.1 are instantiated in the meta-model `CD` and let $[\![O]\!]$ denote the embedding of its OCL definition into relational logic. Formula $\phi^d_{\mathbf{\Delta}^U_{CD}}$ would contain, among the others, the following formula regarding the `setName` operation:

$$
\begin{aligned}
[\![\texttt{setName}]\!] \equiv \exists\ & n : \texttt{String}, \mathit{self} : \texttt{Class}_{CD^t} \mid \\
& \texttt{name}_{CD^t} \circ \mathit{self} = n \wedge \\
& \forall\ c : \texttt{Class}_{CD^t} \mid (\texttt{name}_{CD^t} \circ c = \texttt{name}_{CD^s} \circ c \vee \mathit{self} = c) \wedge \\
& \texttt{Package}_{CD^s} = \texttt{Package}_{CD^t} \wedge \texttt{Class}_{CD^s} = \texttt{Class}_{CD^t} \wedge \\
& \texttt{Attribute}_{CD^s} = \texttt{Attribute}_{CD^t} \wedge \texttt{classes}_{CD^s} = \texttt{classes}_{CD^t} \wedge \\
& \texttt{attributes}_{CD^s} = \texttt{attributes}_{CD^t} \wedge \texttt{general}_{CD^s} = \texttt{general}_{CD^t}
\end{aligned}
$$

Assuming that the set of valid edit operations is comprised by these methods, the overall $\phi^d_{\mathbf{\Delta}^U_{CD}}$ formula would test, for each step, if the pre- and post-states are related by an operation, resulting in the following formula:

$$
\begin{aligned}
\phi^d_{\Delta^U_{CD}} \equiv\ & op\ [R_t \Rightarrow R_1] \wedge \ldots \wedge op\ [R_s \Rightarrow R_i, R_t \Rightarrow R_{i+1}] \wedge \ldots \wedge op\ [R_s \Rightarrow R_{d-1}] \\
& \textbf{where } op = [\![\texttt{setName}]\!] \vee [\![\texttt{setPersistent}]\!] \vee [\![\texttt{addAttribute}]\!] \vee \\
& \qquad [\![\texttt{remAttribute}]\!] \vee [\![\texttt{moveAttribute}]\!] \vee \\
& \qquad [\![\texttt{addClass}]\!] \vee [\![\texttt{remClass}]\!]
\end{aligned}
$$

From this example it is easy to envision scenarios where the distance is not symmetric: it suffices to have operations that are not undoable by the other ones.

OBD allows the assignment of lower costs to complex updates (by simply creating an operation that composes smaller operations), providing the user a certain degree of control over the produced updates. However, assigning higher costs to simple operations is not as straight-forward as they may not be decomposable. This would require customizable operation costs which is left as future work.

Graph-edit distance While expressive, metrics like the one just presented require the duplication of free relations at each step, greatly encumbering the solving process. However, some "history-ignorant" metrics can measure the distance between two models without searching for intermediate states, which can be deployed as a lighter, albeit still iterative, procedure. The *graph-edit distance* (GED) (Voigt, 2011) is an example of such metric. GED can be implemented by applying the symmetric difference to tuple sets, as:

$$D^d_{\boldsymbol{\Delta}^\ominus} = D_{\boldsymbol{M}^s}$$
$$\phi^d_{\boldsymbol{\Delta}^\ominus} = \sum_{R \in \mathcal{R}_{\boldsymbol{M}}} |R_s \ominus R_t|$$

Since each sort A gives rise to a unary relation $A_{\boldsymbol{CD}}$ that denotes which elements of that sort are present in each instantiation of $\boldsymbol{CD}$, the symmetric difference between bindings counts changes both in the edges and in the nodes of the graph representation of a model. GED can be automatically derived for each provided meta-model. We denote the derivation of this distance from the meta-model of a transformation domain M by $\Delta^\ominus_M$. The tradeoff is that the constraint language must now support integer expressions. Since Kodkod relies on SAT solvers to solve problems, integers are converted to their binary representation while integer operations are converted into boolean circuits. While too elaborate to support complex operations, this embedding is feasible when used in a restrained manner.

GED is meta-model independent and may be automatically inferred for any meta-model provided by the user. For instance, for a transformation domain CD conforming to $\mathtt{CD}$, $\phi^d_{\Delta^\ominus_{CD}}$ could be defined as:

$$\begin{aligned}
&\phi^d_{\Delta^\ominus_{CD}} \equiv d = \\
&\quad |\mathtt{Package}_{\boldsymbol{CD}^s} \ominus \mathtt{Package}_{\boldsymbol{CD}^t}| + |\mathtt{Class}_{\boldsymbol{CD}^s} \ominus \mathtt{Class}_{\boldsymbol{CD}^t}| + \\
&\quad |\mathtt{Attribute}_{\boldsymbol{CD}^s} \ominus \mathtt{Attribute}_{\boldsymbol{CD}^t}| + \\
&\quad |\mathtt{name}_{\boldsymbol{CD}^s} \ominus \mathtt{name}_{\boldsymbol{CD}^t}| + |\mathtt{classes}_{\boldsymbol{CD}^s} \ominus \mathtt{classes}_{\boldsymbol{CD}^t}| + \\
&\quad |\mathtt{attributes}_{\boldsymbol{CD}^s} \ominus \mathtt{attributes}_{\boldsymbol{CD}^t}| + |\mathtt{general}_{\boldsymbol{CD}^s} \ominus \mathtt{general}_{\boldsymbol{CD}^t}|
\end{aligned}$$

Unlike OBD, the simple metric definition provided by GED assumes a fixed repertoire of edit operations which may not be desirable. In particular, there is no control over the "cost" of complex operations. For example, changing the name of a class will have a cost of 2, since it requires deleting the current $\mathtt{name}_{\boldsymbol{CD}^s}$ edge and inserting a new

one, while adding a new attribute to a class will cost 3, since it requires inserting a new attribute in $\texttt{Attribute}_{\boldsymbol{CD}^t}$, setting its $\texttt{name}_{\boldsymbol{CD}^t}$, and connect it to the class through $\texttt{attributes}_{\boldsymbol{CD}^t}$. If the user wishes both these operations to be atomic edits and have the same unitary cost, or to allow only particular edits in order to further control the acceptable solutions, she must rely on more expressive metrics, like OBD.

Combining metrics We have shown in Section 6.2 that the technique developed to deploy bidirectional constraint maintainers could be adapted to more general procedures, like multidirectional constraint maintainers and synchronizers, given a mechanism to combine metrics defined over each transformation domain.

Let $M_1, ..., M_n$ be n transformation domains and $\boldsymbol{\Delta}_i$, for $i \in [1..n]$ denote the respective distance function. Let also $+^{-n} : \mathbb{N} \leftrightarrow \mathbb{N} \times ... \times \mathbb{N}$ be a multi-valued function that, given a natural k, returns all n tuples whose sum is k (e.g., $+^{-2}\ 3 = \{(0,3),(1,2),(2,1),(3,0)\}$). Formula $\boldsymbol{\Delta}^d$ for a metric over product $M_1 \times ... \times M_n$ can then be defined as:

$$\phi^d_{\boldsymbol{\Delta}} = \langle \bigvee_{(d_1,...,d_n) \in +^{-n}\ d} (\bigwedge_{i \in [1..n]} \phi^{d_i}_i\ \boldsymbol{\Delta}_0), \bigcup_{i \in [1..n]} \mathcal{S}_{\boldsymbol{\Delta}_i}, \bigcup_{i \in [1..n]} D_{\boldsymbol{\Delta}_i} \rangle$$

Under particular metrics, this procedure can be greatly simplified. For instance, under GED one could easily define a distance over tuples by simply adding up the symmetric distances of the bindings over all relations $D_{\boldsymbol{M}^i}$. For instance, a distance on two transformation domains CD and DS over meta-models `CD` and `DBS` could be defined as:

$$\begin{aligned}
&\phi^d_{\Delta^{\ominus}_{CD,DS}} \equiv d = \\
&\quad |\texttt{Package}_{\boldsymbol{CD}^s} \ominus \texttt{Package}_{\boldsymbol{CD}^t}| + |\texttt{Class}_{\boldsymbol{CD}^s} \ominus \texttt{Class}_{\boldsymbol{CD}^t}| + \\
&\quad |\texttt{Attribute}_{\boldsymbol{CD}^s} \ominus \texttt{Attribute}_{\boldsymbol{CD}^t}| + \\
&\quad ... \\
&\quad |\texttt{Schema}_{\boldsymbol{DS}^s} \ominus \texttt{Schema}_{\boldsymbol{DS}^t}| + |\texttt{Table}_{\boldsymbol{DS}^s} \ominus \texttt{Table}_{\boldsymbol{DS}^t}| + \\
&\quad |\texttt{Column}_{\boldsymbol{DS}^s} \ominus \texttt{Column}_{\boldsymbol{DS}^t}| + \\
&\quad ...
\end{aligned}$$

Combined OBDs could also be defined, by defining the set of valid edit operations as the union of the valid operations in the meta-models of the transformation domains M_i.

6.3.2 Internal TO-MF

Due to its low-level, GED can be deployed in more effective ways, in particular by exploring advanced capabilities of SAT solvers. This allows, for particular metrics, the deployment target-oriented model finding as internal procedures.

A model finding problem $\langle \mathcal{A}, \phi, B \rangle$ is encoded in a boolean formula by Kodkod in the following way (Torlak and Jackson, 2007). For every mapping $R \mapsto (R_L, R_U)$ of arity n, a matrix R with n dimensions of size $|\mathcal{A}|$ is created, where the value at each position $j_1, ..., j_n$ denotes whether the tuple $(a_{j_1}, ..., a_{j_n})$ is present in relation R. Positions regarding tuples in the lower-bound R_L are automatically set to 1, while those regarding tuples outside the upper-bound are set to 0; for those in between, fresh boolean variables $R_{j_1,...,j_n}$ are created. Relational operations are then translated to matrix operations, resulting in a boolean formula whose free boolean variables are those occurring in the matrix representation of the relations. Let's call this translation $\lceil \mathcal{A}, \phi, B \rceil$.

When defined at the model finding level, model metrics are embedded in formula ϕ and encoded in the boolean formula in each iteration. When dealing with arithmetic operations, which are unfolded into corresponding boolean circuits, this may easily encumber the solving process. Modern SAT solvers provide additional functionalities that may allow minimization to be controlled internally, which has been explored elsewhere by us (Cunha et al., 2014).

One such capability is the support for *cardinality constraints*. In this kind of problems, the solver is given an additional set of clauses, where at most a given number of them must hold. Deploying GED remains an iterative procedure, but the distance computation is now handed internally.

Proposition 6.7 (minimize GED with cardinality constraints)**.** *A target-oriented model finding problem* $\llbracket \mathcal{A}, \phi, B, T \rrbracket_{\Delta^{\ominus}}$ *can be solved by iteratively calling SAT solvers with cardinality constraints* $\langle \lceil \mathcal{A}, \phi, B \rceil, c_d \rangle$ *with*

$$c_d = \sum_{R \in \delta B, (a_{j_1},...,a_{j_n}) \in R_T} \neg R_{j_1,...,j_n} + \sum_{R \in \delta B, (a_{j_1},...,a_{j_n}) \in R_U \setminus R_L \setminus R_T} R_{j_1,...,j_n} \leqslant d$$

from $d = 0$ *until SAT.*

Proof. The problem is correct because the problem with cardinality constraints will still guarantee that ϕ holds. A negated literal $\neg R_{j_1,...,j_n}$ is created for every element in

the target and a positive literal $R_{j_1,\ldots,j_n}$ for each element outside the target but within the bounds of the problem. Thus solving the problem for $d = n$ amounts to changing n elements from the target; since the process starts at $d = 0$, this will amount to a target-oriented model finding problem under GED. □

The cardinality constraint c_d essentially counts the GED between the returned valuation and the provided target, which is to be minimized. Since minimality is handled internally rather than by a boolean circuit representing the GED calculation, this technique has shown to be more efficient than that presented at Section 6.3.1 (Cunha et al., 2014). If there is a solution for $d = 0$, then it is exactly the target, guaranteeing stability.

Maximum satisfiability problems represent another class of SAT problems whose goal is to maximize the number of clauses that are satisfied. However, typically some of the problem's clauses are obligated to be satisfied. PMax-SAT problems $\langle \phi, \psi \rangle$ address this issue by dividing the problem into a set ϕ of *hard* clauses—i.e., mandatory—and a set ψ of *soft* clauses—i.e., optional—which are to be maximized.

Proposition 6.8 (minimize GED as PMax-SAT)**.** *A target-oriented model finding problem* $[\![\mathcal{A}, \phi, B, T]\!]_{\Delta^{\ominus}}$ *can be embedded in a PMax-SAT problem* $\langle \lceil \mathcal{A}, \phi, B \rceil, \psi \rangle$*, such that*

$$\psi = \bigcup_{R \in \delta B, (a_{j_1},\ldots,a_{j_n}) \in R_T} R_{j_1,\ldots,j_n} \cup \bigcup_{R \in \delta B, (a_{j_1},\ldots,a_{j_n}) \in R_U \setminus R_L \setminus R_T} \neg R_{j_1,\ldots,j_n}$$

Proof. Constraint ϕ is set as hard constraint, so the returned valuations are solutions of the model finding problem. For each potential tuple of a relation, if the respective tuple is in the target, a soft literal clause containing that variable is created; otherwise, a soft literal clause with the negated variable is created: maximizing the number of satisfied soft clauses will amount to a target-oriented model finding problem under GED. □

Essentially, maximizing the set of satisfying soft clauses minimizes, according to GED, the number of changes from the target. This technique has the advantage of no longer being an iterative procedure. However, it will depend on the solver's ability to extract suitable UNSAT cores, which may be unpredictable (Cunha et al., 2014). The optimal solution is returned if it is possible to assign true to all soft clauses, providing stability.

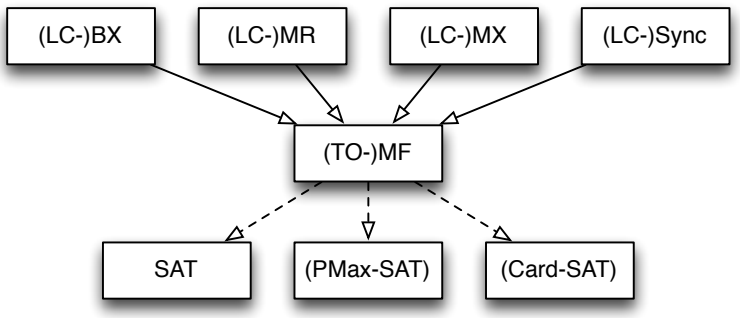


Figure 6.6: Model finding at the core of MDE tasks.

6.4 Discussion

This chapter explored how bidirectional transformations in the shape of constraint maintainers can be deployed as model finding procedures. In the process, we have shown that this process can be generalized to address other MDE tasks, as summarized in Figure 6.6. A deeper analysis of target-oriented model finding problems over different classes of SAT problems was performed by us elsewhere (Cunha et al., 2014).

The fact that different MDE tasks can be deployed over a unifying formalization hints that each of these tasks may draw inspiration from the solutions proposed for different applications. In particular, it raises the question of deploying constraint maintainers over existing model repair techniques. Model repair techniques can be classified over two broad categories. In one, techniques derive repair plans by syntactic analysis of the constraints and model instances. In the other, in which our embedding falls, techniques employ solvers to calculate a new model that satisfies the constraints. While the former is usually more efficient and scales better, it is less expressive and flexible than the latter. For example, it is not as well suited to deal with multiple inconsistencies, nor inconsistencies that affect a large portion of the model (likely to occur when using closures to express reachability properties). Moreover, they usually rely on a fixed set of abstract edit operations to specify the repairs, which the user must manually instantiate and is not able to parametrize.

Most existing intra-model repair tools (Reder and Egyed, 2012; Puissant et al., 2013; Xiong et al., 2009) fall into the first class of repair techniques. To achieve high efficiency, they typically limit the expressiveness of repair updates, do not require the generation of fully consistent models, or require constraints to be manually annotated

with inconsistency resolution hints. Nonetheless, it would be interesting to explore whether these techniques would scale to the bidirectional transformation setting. For a maintainer $T : M \bowtie N$, this could be achieved by creating a "dummy" meta-model $M \times N$, over which the model repairs are deployed by interpreting T as an intra-model constraint. To execute transformations $\overrightarrow{T}\ (m, n)$ (and dually for $\overleftarrow{T}$), an additional restrictions could be introduced in the constraint stating that m is not modified in the repair process (which may require the duplication of model m if there is no native mechanism to access the pre-state of the model being repaired).

The few existing tools based on model finding do not abide to the desirable principle of least-change. Kleiner et al. (2010) propose a general approach for constraint-based solving in the context of MDE (including application to model repair), using the Alloy Analyzer and OPL solvers as concrete examples. However, the original inconsistent model is specified as the lower-bound for the new model, meaning that the solver will only be able to add new atoms and edges while solving the constraints. Following a similar approach, Straeten et al. (2011) assess the feasibility of using Kodkod to repair inconsistencies. Given an inconsistent Kodkod problem, a consistent problem is found by relaxing the bounds of the original model in order to allow the addition of new relations or the removal of relations suspected of causing the selected inconsistencies. This assumes that the concrete inconsistencies were previously detected by an external tool. In both approaches there is also no control over how close the new model is to the original one and the authors do not reason on how to manage the creation of new atoms. Hegedüs et al. (2011) describe a technique for generating quick fixes for DSMLs embedding on CSP over models. The technique guarantees that the number of inconsistencies on the model decreases, even if side-effects occur. This is achieved by applying every candidate fix to the inconsistent model and detecting and counting the inconsistencies in the resulting model. In terms of expressivity, this last approach is the closest to ours, but, being also solver based, it suffers from the same scalability issues, as will be discussed in Section 9.5 when evaluating its deployment.

Under this formalization, concrete constraint maintainer frameworks must only process intra- and inter-model constraints into relational logic, at which point consistency-restoring transformations can be executed as model finding procedures. In the remainder of this part we will provide such instantiations for QVT-R (Chapter 7) and ATL (Chapter 8) model transformations. The deployment of the technique as an effective tool is lastly presented at Chapter 9.

Chapter 7

Deploying QVT-R Transformations

To support MDE, the Object Management Group (OMG) launched the Model-driven Architecture (MDA) initiative, which prescribed the usage of MOF (OMG, 2013) (usually presented as UML class diagrams (OMG, 2011b)) and OCL (OMG, 2012) for the specification of (object oriented) models and constraints over them. More recently, to specify transformations between models, OMG (2011a) proposed the *Query/View/Transformation* (QVT) standard. While QVT provides three different languages for the specification of transformations, the most relevant to MDE is the *QVT Relations* (QVT-R) language, that allows the specification of a bidirectional transformation by defining a single declarative consistency relation between two (or more) transformation domains. Given this specification, the transformation can be run in two modes: *checkonly*, to test whether two models are consistent according to the specified relation; or *enforce*, that given two models and an execution direction (picking one of the transformation domains as the target) updates the target model in order to recover consistency. The standard prescribes a "check-before-enforce" semantics, that is, enforce mode cannot modify the target if the models happen to be already consistent according to checking semantics. Clearly, this amounts to a constraint maintainer with a consistency checker and consistency-restoring transformations, stability arising from the "check-before-enforce" policy, an interpretation first pointed out by Stevens (2010).

However, effective tool support for QVT-R has been slow to emerge, which hinders the universal adoption of this standard. In part, this is due to the incomplete and ambiguous semantics defined in (OMG, 2011a). While the checking semantics has recently been clarified and formalized (Stevens, 2013; Bradfield and Stevens, 2012; Guerra and de Lara, 2012), the enforcement semantics still remains largely obscure and

even incompatible with other OMG standards, despite some recent efforts to provide a formal specification (Bradfield and Stevens, 2013). Namely, it completely ignores possible OCL constraints over the meta-models, thus allowing updates that can lead to ill-formed target models. Likewise, none of the existing QVT-R model transformation tools supports such constraints, which makes them unusable in many realistic scenarios. Unfortunately, there are other problems that affect them. Some do not even comply to the standard syntax and support only a "QVT-like" language (including not providing both running modes as required by the standard). Others do not support truly non-bijective bidirectional transformations (for example, ignoring the pre-state of the target model in the enforce mode). Some purposely disregard the intended QVT-R semantics (including checking semantics) and implement a new (still unclear and ambiguous) one. In most cases it is not clear if the supported checking semantics is equivalent to the one formalized in (Stevens, 2013; Bradfield and Stevens, 2012; Guerra and de Lara, 2012). And finally, none clarifies the problems and ambiguities in the standard concerning enforcement semantics, nor presents a simple enough alternative for this mode that makes its behavior predictable to the user.

In this chapter we explore how *least-change QVT-R bidirectional transformations can be deployed as constraint maintainers over model finding*. Doing so requires the derivation of a relational inter-model constraint that embodies the QVT-R transformation, so that bidirectional transformations can be deployed according to the formalization presented in Chapter 6. Such approach allows both the meta-models and transformation specifications to be annotated with OCL constraints, and will support a large subset of the standard QVT-R language, including execution of both modes independently as prescribed. The checking semantics will closely follow the one specified in the standard, being equivalent to the one formalized in (Stevens, 2013; Bradfield and Stevens, 2012; Guerra and de Lara, 2012). Finally, instead of the ambiguous (and OCL incompatible) enforcement semantics proposed in the standard, our transformations will follow the clear and predictable principle of least-change (Meertens, 1998), and just return updated consistent target models that are at a minimal distance from the original. In particular, the "check-before-enforce" policy required by QVT-R is trivially satisfied by this semantics. Our deployment supports the two different meta-model independent mechanisms to measure the distance between two models presented in Section 6.3: GED, that just counts insertions and deletions of nodes and edges in the graph that corresponds to a model; and OBD where the user is allowed to parameterize

which operations should count as valid edits, by attaching them to the meta-model and specifying their pre- and post-conditions in OCL.

While the bidirectional scenario is the focus of all work on QVT-R that we are aware of, the standard imposes no limitation on the number of transformation domains that may be related. The multidirectional semantics proposed by the standard is however too limitative to be used in practical situations. Backed up by the generalization of the bidirectional transformation problem from Section 6.2, we explore an extension to this semantics that renders it more expressive.

The contributions of the chapter are the following:

- we analyze the standard QVT-R *checking semantics* and embed it in relational logic (Section 7.2), solving ambiguities that occur;
- we show that the standard QVT-R *enforcement semantics* is undesirable, and propose instead an alternative based on least-change (Section 7.3);
- we explore the heretofore disregarded problem of *multidirectional* QVT-R transformations (Section 7.4).

These are prepended by an introduction to the QVT-R language (Section 7.1) and succeeded by a discussion of the overall results (Section 7.5).

7.1 QVT Relations

In this section the basic concepts regarding the QVT-R language are introduced. A more detailed presentation can be found in the OMG (2011a) standard.

7.1.1 Basic Concepts

The QVT standard by OMG (2011a) defines three model transformation languages: *QVT Operational*, an imperative language for the specification of unidirectional transformation; *QVT Relations*, whose specifications are declarative consistency relations that can be run in multiple directions; and *QVT Core*, a lower level declarative language over which QVT-R is supposed to be embedded.

A QVT-R specification consists of a *QVT-R transformation* T between a set of transformation domains, embodied by meta-model specifications, that states under which conditions their conforming models are considered consistent. For most of this

chapter we restrict ourselves to the bidirectional scenario, i.e., QVT-R transformations between two transformation domains. The generalization of these concepts to the multidirectional scenario is addressed independently in Section 7.4. From T, QVT-R requires the inference of three artifacts: a relation $T : M \leftrightarrow N$ that tests whether two models $m : M$ and $n : N$ are consistent and transformations $\overrightarrow{T} : M \times N \leftrightarrow N$ and $\overleftarrow{T} : M \times N \leftrightarrow M$ that propagate changes on a *source* model to a *target* model, restoring consistency between the two. Thus, QVT-R transformations can be interpreted in two modes: *checkonly* mode, where T simply checks the models for consistency; and *enforce* mode, where $\overrightarrow{T}$ or $\overleftarrow{T}$ is applied to inconsistent models in order to restore consistency, depending on which of the two models the user wishes to update. The transformations are aware of both models when executing: if models $m : M$ and $n : N$ are initially consistent, and m is updated to m', $\overrightarrow{T}$ takes as input both m' and n to produce the new consistent n'. This way the system is able to retrieve from n information not present in the opposing model. This formalization of QVT-R is inspired by the concept of constraint maintainer (Meertens, 1998), and was first proposed by Stevens (2010). Naturally, when the transformations propagate an update the result is expected to be consistent. The properties that transformations in QVT-R are expected to hold are precisely those of regular maintainers: CORRECT and HIPPOCRATIC, the latter embodied by the "check-before-enforce" policy enforced by the standard (OMG, 2011a, p. 15).

A QVT-R transformation is defined by a set of *QVT-R relations*. Each QVT-R relation consists of a *domain pattern* for each of the QVT-R transformation's domains, that defines which objects of the corresponding model it relates by pattern matching—we call these *candidate* elements. It may also include *when* and *where* constraints, that act as a kind of pre- and post-conditions for the QVT-R relation application, respectively. These constraints may contain arbitrary OCL expressions (although the standard imposes some strict restrictions to achieve effective enforcement semantics, as will be shown in Section 7.3). The abstract syntax of a (binary) QVT-R relation is the following:

```
[top] relation R {
    [variable declarations]
    domain M a : A { π_M^T } [{ π_M^C }];
    domain N b : B { π_N^T } [{ π_N^C }];
    [when { ψ }]
```

```
    [where { φ }]
}
```

In a QVT-R relation R, the domain pattern for transformation domain M consists of a *domain variable* a and a template π_M^T that binds the values of some of its properties (attributes or related associations), which candidate objects of type A must match. Likewise for the domain N for transformation domain N. To simplify the presentation, the above syntax restricts QVT-R relations to have exactly one domain variable per transformation domain. If the multiplicity of a navigated property R is different from one, pattern templates involving it denote inclusion tests, i.e., a pattern $R = a$ denotes the test $\langle a \rangle \in R$. Properties can also be navigated backwards through the **opposite** keyword. Templates can be complemented with arbitrary OCL constraints, denoted by π_M^C and π_N^C. The conjunction of π_M^T and π_M^C (π_N^T and π_N^C) will be denoted by π_M (π_N). QVT-R relations can optionally be marked as **top**, in which case they must hold for every candidate a and b elements. Otherwise, they are only tested for particular a and b elements when invoked in **when** or **where** clauses.

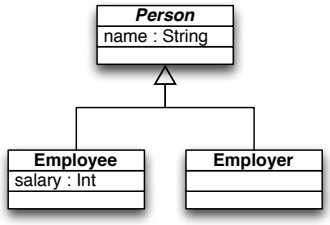
7.1.2 QVT-R Transformation Examples

The first example that will be used is a simplified version of the classic object-relational mapping transformation that illustrates the QVT-R standard (OMG, 2011a), already presented at Chapter 1. Figure 7.1 defines a QVT-R transformation `cd2dbs` between two transformation domains CD and DS conforming to meta-models `CD` (Figure 6.1a) and `DBS` (Figure 6.1b) respectively, that will give rise to a constraint maintainer `cd2dbs`: $CD \bowtie DS$. The goal of this bidirectional transformation is to map every persistent class of a package to a table of the matched schema. Each table should contain a column for each attribute (including inherited ones) of the corresponding class. An intra-model constraint of the `CD` meta-model that cannot be captured by class diagrams, nor by QVT-R key constraints (more in Section 7.3.1), is the requirement that the association `general`, which entails the inheritance tree of the class diagram, should be acyclic. One must resort to OCL constraints to express it, for example by adding the following invariant to the `CD` meta-model:

```
context Class inv:
    not self.closure(general)->includes(self)
```

```
transformation cd2dbs (CD:CD,DS:DBS) {

    // PackageToSchema
    top relation P2S {
        n:String;
        domain CD p:Package { name = n };
        domain DS s:Schema  { name = n };
    }

    // ClassToTable
    top relation C2T {
        n:String;
        domain CD c:Class {
            persistent                = true,
            opposite(Package.classes) = p:Package{},
            name                      = n };
        domain DS t:Table {
            opposite(Schema.tables)   = s:Schema{},
            name                      = n };
        when  { P2S(p,s); }
        where { A2C(c,t); }
    }

    // AttributeToColumn
    relation A2C {
        domain CD c:Class {};
        domain DS t:Table {};
        where { PA2C(c,t) and SA2C(c,t); }
    }

    // PrimitiveAttributeToColumn
    relation PA2C {
        n:String;
        domain CD c:Class {
            attribute  = a:Attribute { name = n } };
        domain DS t:Table {
            column     = l:Column    { name = n } };
    }

    // SuperAttributeToColumn
    relation SA2C {
        domain CD c:Class { general = g:Class {} };
        domain DS t:Table {};
        where { A2C(g,t); }
    }
}
```

Figure 7.1: Simplified version of the `cd2dbs` QVT-R transformation.



(a) Class diagram cd_0.

```
CREATE TABLE Employee (
  name     varchar (255),
  salary   int
);
CREATE TABLE Employer (
  name     varchar (255)
);
```

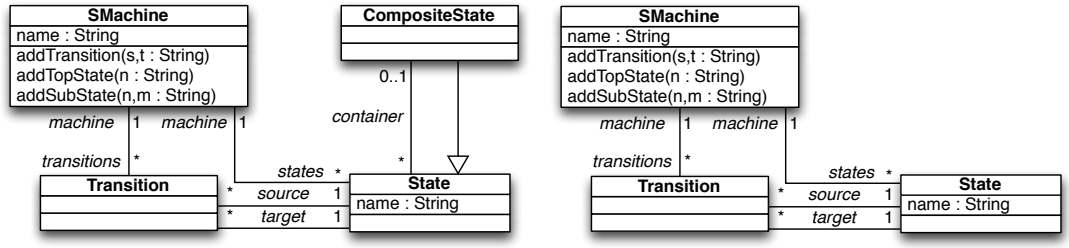
(b) Database schema ds_0.

Figure 7.2: Example models for `cd2dbs`.

The constraint relies on the transitive closure operator, which has recently been introduced to the OCL standard (OMG, 2012, p. 168). Although simplified (no associations in `CD` and no keys in `DBS`), this version of the transformation still exhibits some of the problems of the original version presented in the standard, which will be described in Section 7.2.2.

In `cd2dbs` there are two top QVT-R relations: `P2S` (*PackageToSchema*) that maps each package to a schema with the same name, and `C2T` (*ClassToTable*) that maps each class to a table with the same name. To ensure that a class is only mapped to a table if the respective package and schema are also matched, QVT-R relation `C2T` invokes `P2S` (with concrete domain variables) in the **`when`** clause. For a concrete class c and table t, `C2T` also calls QVT-R relation `A2C` (*AttributeToColumn*) in the **`where`** clause, that will be responsible for mapping each attribute in c to a column in t. `A2C` directly calls `PA2C` (*PrimitiveAttributeToColumn*, complex attributes are disregarded in this simplified version), that translates each attribute directly declared in c to a column in t, and `SA2C` (*SuperAttributeToColumn*), that recursively calls `A2C` on the super-class of c, so that each inherited attribute is also translated to a column in t. Figure 7.2 depicts a class diagram and a database schema that are supposed to be consistent according to this consistency relation (italic class diagram names denote non-persistent classes, like `Person`).

Another classical bidirectional model transformation example is that of the expansion/collapse of a hierarchical state machine (HSM). In a HSM, states may themselves contain sub-states (in which case they are called composite states), as defined by the `HSM` meta-model in Figure 7.3a. Likewise the `CD` meta-model, the `HSM` meta-model also requires an additional OCL constraint to avoid circular containment. Transitions may exist between sub-states and states outside their owning composite state. One



(a) Meta-model HSM. (b) Meta-model NHSM.

Figure 7.3: Class diagrams of the HSM and NHSM meta-models.

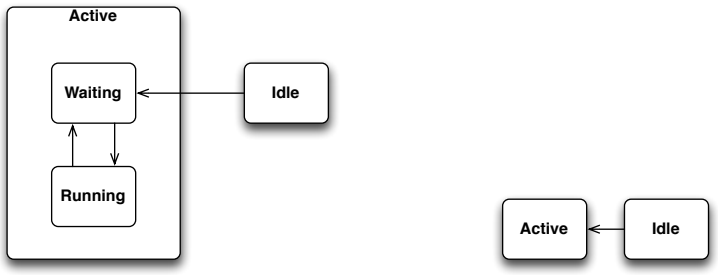
advantage of HSMs is abstraction, and a HSM can be collapsed into a non-hierarchical state machine (NHSM) that presents only top-level states, inheriting the incoming and outcoming transitions of their sub-states. The NHSM meta-model (Figure 7.3b) is similar to HSM without the `container` association and the `CompositeState` class.

The consistency relation between a HSM and its collapsed view is specified by the `hsm2nhsm` QVT-R transformation in Figure 7.4 between transformation domains HM and NM conforming to HSM and NHSM, respectively, giving rise to the constraint maintainer `hsm2nhsm` : $HM \bowtie NM$. Much like `cd2dbs`, top QVT-R relation M2M (*MachineToMachine*) relates state machines with the same name, being a pre-condition of the S2S (*StateToState*) QVT-R transformation. Top QVT-R relation S2S relates every state of a HSM with a NHSM state with the same name as the top-level state owning it. The **`where`** clause of QVT-R relation S2S tests whether the HSM state s is top-level or not: if so, TS2S (*TopStateToState*) is called, which matches s to a NHSM state with the same name; otherwise, SS2S (*SubStateToState*) is called, which recursively calls S2S with the container state of s. Each transition is mapped by the top QVT-R relation T2T (*TransitionToTransition*), which can be trivially specified by resorting to a **`where`** clause stating that two transitions are related if their source and target states are related by S2S. Since every sub-state in a HSM is related to the same NHSM top-state as its container, every transition between sub-states is automatically pushed up to their top-states.

Figure 7.5 depicts two very simple state machines that are consistent according to this consistency relation: in the HSM (Figure 7.5a) there is a state `Idle` and a composite state `Active` containing two states `Waiting` and `Running`; a transition connects the `Idle` state to the `Waiting` state. In its collapsed view (Figure 7.5b), the `Active` state is folded, while the transition from the sub-state `Waiting` is pushed up to it.

```
transformation hsm2nhsm (HM : HSM, NM : NHSM) {

    // SMachineToSMachine
    top relation M2M {
        n:String;
        domain HM s:SMachine { name = n };
        domain NM t:SMachine { name = n };
    }

    // StateToState
    top relation S2S {
        domain HM s:State {
            machine = sm:SMachine{} };
        domain NM t:State {
            machine = tm:SMachine{} };
        when  { M2M(sm,tm); }
        where {
            if s.container->isEmpty() then TS2S(s,t)
            else SS2S(s,t) endif;
      }
    }

    // TopStateToState
    relation TS2S {
        n: String;
        domain HM s:State { name = n };
        domain NM t:State { name = n };
    }

    // SubStateToState
    relation SS2S {
        domain HSM s:State {};
        domain NHSM t:State {};
        where { S2S(s.container,t); }
    }

    // TransitionToTransition
    top relation T2T {
        domain HM ht:Transition {
            target = htt:State{},
            source = hts:State{} };
        domain NM nt:Transition {
            target = ntt:State{},
            source = nts:State{} };
        where { S2S(hts,nts) and S2S(htt,ntt); }
    }
}
```

Figure 7.4: The hsm2nhsm QVT-R transformation.



(a) HSM hm_0. (b) NHSM nm_0.

Figure 7.5: Example for `hsm2nhsm`.

7.2 Checking Semantics

QVT-R's checking semantics assesses whether two models are consistent according to the specified QVT-R transformation. Although the consistency check is by itself fundamental to detect inconsistencies, in QVT-R it is also at the core of the enforcement semantics since the latter must "check-before-enforce". In this section we embed such semantics into a typed relational constraint that is amenable to be deployed under the formalization presented in Chapter 6, i.e., for a QVT-R transformation T, we show how to derive the typed relational constraint $\boldsymbol{T} = \langle \phi_{\boldsymbol{T}}, \mathcal{S}_{\boldsymbol{T}}, D_{\boldsymbol{T}} \rangle$.

7.2.1 Standard Checking Semantics

The semantics of a QVT-R relation differs whether it is invoked at the top-level or with concrete domain variables by **`when`** and **`where`** clauses. The specified top-level semantics is directional. As such, from each QVT-R relation R two formulae $R^{\blacktriangleright}$ and $R^{\blacktriangleleft}$ must be derived, to check whether $m : M$ is R-consistent with $n : N$ and if $n : N$ is R-consistent with $m : M$, respectively. For a QVT-R relation R, $R^{\blacktriangleright}$ and $R^{\blacktriangleleft}$ will denote the embedding of a top QVT-R relation R into a relational formula, the former being formalized as follows:

$$
\begin{aligned}
R^{\blacktriangleright} \equiv \forall\ & xs \mid \psi_{\triangleright} \wedge \pi_M \Rightarrow (\exists\ ys \mid \pi_N \wedge \phi_{\triangleright}) \\
\textbf{where}\ & xs = \mathsf{fv}(\psi \wedge \pi_M) \cup \{\, a : A_{\boldsymbol{M}} \,\}, \\
& ys = (\mathsf{fv}(\pi_N \wedge \phi) \cup \{\, b : B_{\boldsymbol{N}} \,\}) \setminus xs
\end{aligned}
$$

Here $\mathsf{fv}(e)$ retrieves the set of free variables from the expression e, so xs denotes the set of variables used in the **`when`** constraint and in the source domain pattern, while ys is the set of variables used exclusively in the **`where`** constraint and in the target domain

pattern. This semantics is rather straightforward: essentially, for every element $a : A$ that satisfies the **when** condition ψ and matches the π_M domain pattern, there must exist an element $b : B$ that matches the π_N domain pattern and satisfies the **where** condition ϕ. The semantics of $R^{\blacktriangleleft}$ in the opposite direction is dual. Domain pattern templates, which consist of sets of property bindings, for example

```
domain M a : A
 { X = b : B { Z = d : D { ... }, ... },
   y = c }
 { π_M^C };
```

where `X` and `Z` are many-to-many associations and `y` an attribute, are interpreted under their OCL equivalent (OMG, 2011a, p. 19–20), in this case

```
domain M a : A
 {}
 { π_M^C and a.X->includes(b) and b.Z->includes(d) and a.y = c and ... };
```

since domain patterns may be complemented with OCL constraints. For instance, the following pattern from the `cd2dbs` QVT-R transformation

```
domain CD c:Class
 { persistent = true,
   opposite(Package.classes) = p:Package{},
   name = n }
 {};
```

is equivalent to the pattern

```
domain CD c:Class {}
 {}
 { c.persistent and p.classes->includes{c} and c.name = n };
```

Likewise to the constraints that may annotate meta-models (Section 6.1.2), these OCL expressions, along with those occurring in **where**/**when** clauses ϕ / ψ, are processed into relational logic following the technique proposed by Cunha et al. (2013) (technical details are presented in Chapter 9). The domain pattern just mentioned results in

$$\langle c \rangle \in \mathsf{persistent}_{CD} \wedge \langle p \rangle \in \mathsf{classes}^{\circ}_{CD} \circ c \wedge \mathsf{name}_{CD} \circ c = n$$

The OCL navigation operator and the relational composition operator are defined in opposite directions. Thus, an OCL expression `a.R` results in the relational expression $R.a$.

The QVT-R standard defines rather precisely the top-level semantics, but is omissive about the semantics of QVT-R relations R between A and B invoked with concrete expressions $e_1 : A$ and $e_2 : B$ as $R\ (e_1, e_2)$. Recent works on the formalization of QVT-R checking semantics (Stevens, 2013; Bradfield and Stevens, 2012; Guerra and de Lara, 2012) clarify that it is essentially the same as the top-level—still directional, but defined over concrete elements by fixing the domain variables. However, each relation invocation cannot simply be expanded to this definition, since recursive calls would lead to infinite loops. For instance, in the `hsm2nhsm` QVT-R transformation, `S2S` and `SS2S` call each other recursively: if each `SS2S(s,t)` and `S2S(s.container,t)` calls were expanded, it would lead to infinite expansions. Instead, since predicates can be encoded by relations, for every relation R that is called by another we introduce two new relation variables $R_{\triangleright} : A \leftrightarrow B$ and $R_{\triangleleft} : A \leftrightarrow B$ in $D_{\boldsymbol{T}}$; then, each relation invocation $R\ (e_1, e_2)$ is translated as either the membership test $\langle e_1, e_2 \rangle \in R_{\triangleright}$ or $\langle e_1, e_2 \rangle \in R_{\triangleleft}$, depending on the direction the calling relation is being run. This was already depicted in the semantics defined above, where, given a formula ψ, $\psi_{\triangleright}$ denotes the same formula with all QVT-R relation invocations replaced by the respective directional version.

The valuation of the $R_{\triangleright}$ and $R_{\triangleleft}$ variables must then be restricted to the desirable QVT-R semantics. From each QVT-R relation R with domain variables of type A and B, two formulae $R_{\blacktriangleright}$ and $R_{\blacktriangleleft}$ are inferred, that restrict the valuation of the free relations $R_{\triangleright} : A \leftrightarrow B$ and $R_{\triangleleft} : A \leftrightarrow B$. The former is formalized as:

$$
\begin{aligned}
R_{\blacktriangleright} \equiv R_{\triangleright} = \{\, a : A_{\boldsymbol{M}}, b : B_{\boldsymbol{N}} \mid (\forall\ xs \mid \psi_{\triangleright} \wedge \pi_M \Rightarrow (\exists\ ys \mid \pi_N \wedge \phi_{\triangleright})) \,\} \\
\quad \textbf{where}\ xs = \mathsf{fv}(\psi \wedge \pi_M), \\
\qquad\qquad ys = \mathsf{fv}(\pi_N \wedge \phi) \setminus xs
\end{aligned}
$$

The test in the opposite direction $R_{\blacktriangleleft}$ is again dual. These formulae are derived a single time, although this step can be skipped for QVT-R relations to which there are no calls. A top relation R may also be called from other QVT-R relations, in which case both $R_{\blacktriangleright}/R_{\blacktriangleleft}$ and $R^{\blacktriangleright}/R^{\blacktriangleleft}$ formulae are derived.

Two models are consistent according to a QVT-R transformation T if they are consistent for all top QVT-R relations in both directions. However, since our embedding of relation calls also requires the axiomatization of the corresponding variables, non-top relations must also be processed into formula $\phi_{\boldsymbol{T}}$. For a QVT-R transformation T, let $\mathcal{T}_T$ denote the set of its top QVT-R relations and $\mathcal{S}_T$ the set of relations which are called

from other relations within it. Its embedding can then be specified as:

$$\phi_{\boldsymbol{T}} = \bigwedge_{R \in \mathcal{T}_T} (R^{\blacktriangleright} \wedge R^{\blacktriangleleft}) \wedge \bigwedge_{R \in \mathcal{S}_T} (R_{\blacktriangleright} \wedge R_{\blacktriangleleft})$$

Since formula $\phi_{\boldsymbol{T}}$ expectedly refers to relations from the transformation domains M and N, the overall typed relational constraint $\boldsymbol{T}$ is defined as:

$$\boldsymbol{T} = \langle \phi_{\boldsymbol{T}}, \mathcal{S}_{\boldsymbol{M}} \cup \mathcal{S}_{\boldsymbol{N}}, D_{\boldsymbol{M}} \cup D_{\boldsymbol{N}} \cup \bigcup_{R \in \mathcal{S}_T} \{R_{\triangleright}, R_{\triangleleft}\} \rangle$$

7.2.2 Relation Invocations

Although it may be tempting (and probably more intuitive) to define $R^{\blacktriangleright}$ in terms of $R_{\triangleright}$, that is $R^{\blacktriangleright} \equiv \forall\ a : A_{\boldsymbol{M}} \mid (\exists\ b : B_{\boldsymbol{M}} \mid \langle a, b \rangle \in R_{\triangleright})$, this definition is not semantically equivalent to the one presented above, as previously discussed by Bradfield and Stevens (2012). For instance, consider the semantics (in the direction of CD) of QVT-R relation `PA2C` from the `cd2dbs` QVT-R transformation:

$$\begin{aligned}
&\texttt{PA2C}^{\blacktriangleleft} \equiv \forall\ t : \texttt{Table}_{\boldsymbol{DS}}, l : \texttt{Column}_{\boldsymbol{DS}} \mid \\
&\quad \langle l \rangle \in \texttt{columns}_{\boldsymbol{DS}} \circ t \Rightarrow (\exists\ c : \texttt{Class}_{\boldsymbol{CD}}, a : \texttt{Attribute}_{\boldsymbol{CD}} \mid \\
&\qquad \langle a \rangle \in \texttt{attributes}_{\boldsymbol{CD}} \circ c \wedge \texttt{name}_{\boldsymbol{CD}} \circ a = \texttt{name}_{\boldsymbol{DS}} \circ l) \\
&\langle c, t \rangle \in \texttt{PA2C}_{\triangleleft} \equiv \forall\ l : \texttt{Column}_{\boldsymbol{DS}} \mid \\
&\quad \langle l \rangle \in \texttt{columns}_{\boldsymbol{DS}} \circ t \Rightarrow (\exists\ a : \texttt{Attribute}_{\boldsymbol{CD}} \mid \\
&\qquad \langle a \rangle \in \texttt{attributes}_{\boldsymbol{CD}} \circ c \wedge \texttt{name}_{\boldsymbol{CD}} \circ a = \texttt{name}_{\boldsymbol{DS}} \circ l)
\end{aligned}$$

Consider a simple `CD` model where a class a with an attribute x extends a class b with an attribute y. Consider also a `DBS` model with a single table a containing columns x and y. While $\texttt{PA2C}^{\blacktriangleleft}$ holds for this pair of models, $\texttt{PA2C}_{\triangleleft}$ returns false for every pair of classes and tables. Of course, there are cases where the two semantics are equivalent. For instance, `C2T` could be defined as a non-top QVT-R relation and be called from the **`where`** clause of `P2S`. The behavior is equivalent because the only free variable (n) is bound to a unitary attribute.

Due to this asymmetry and the directionality of the semantics, the behavior of QVT-R transformations may not meet the expectations of the user. In particular, `cd2dbs` as defined in the standard does not have a bidirectional semantics, because the check in the direction of CD imposes that the only pairs of `cd2dbs`-consistent and well-formed finite models are ones where all classes are non-persistent and there are no tables. To

see why this happens, consider the QVT-R relations A2C and SA2C when checked in the direction of CD. These QVT-R relations call each other recursively, and their non top-level semantics is:

$$\langle c, t\rangle \in \texttt{A2C}_{\triangleleft} \equiv \langle c, t\rangle \in \texttt{PA2C}_{\triangleleft}\ (c, t) \wedge \langle c, t\rangle \in \texttt{SA2C}_{\triangleleft}$$
$$\langle c, t\rangle \in \texttt{SA2C}_{\triangleleft} \equiv \exists\ g : \texttt{Class}_{\boldsymbol{CD}} \mid \langle g\rangle \in \texttt{general}_{\boldsymbol{CD}} \circ c \wedge \langle g, t\rangle \in \texttt{A2C}_{\triangleleft}$$

If the transformation takes into account the OCL constraint requiring inheritance to be acyclic, the predicate $\langle c, t\rangle \in \texttt{A2C}_{\triangleleft}$ never holds in a finite model, since c will be required to have an infinite ascending chain of general objects. This is due to the under-restrictive domain pattern for the domain DS in SA2C (empty in this case), that requires every table to have a matching class with a super-class, which, due to recursion, is also required to have a super-class, and so on. This is but one of the problems that occur in the original specification of this transformation, and is another example of the ambiguities that prevail in the QVT-R standard (OMG, 2011a): while it requires consistency to be checked in both directions, the case study used to illustrate it was clearly not developed with bidirectionality in mind. Note that checking consistency only in the direction of DS does not suffice, since, for example, it will not prevent spurious tables to appear in the target schema.

Relation calls in the **where** and **when** constraints require reasoning about recursion. Two situations can be distinguished: one is well-founded recursion, where the call graph of the transformation contains a loop, but in any evaluation it is traversed only finitely many times; another is cyclic (or infinite) recursion, where such a loop may actually be traversed infinitely many times (e.g., when a relation directly or indirectly calls itself with the same arguments). The semantics of well-founded recursion is not problematic, but the standard is omissive about what should happen when infinite recursion occurs. A possible interpretation is that it should not be allowed, although in general it is undecidable to detect if that is the case. Similarly to some QVT-R formalizations (Stevens, 2013; Guerra and de Lara, 2012), the embedding presented in this chapter is not well-defined when infinite recursion occurs.

Recently, a formal semantics of QVT-R was proposed by Bradfield and Stevens (2012) that is well-defined even in presence of infinite recursion, by resorting to modal mu calculus. To see why taking OCL constraints into account is fundamental, a transformation conforming to this semantics, but that ignores the requirement that general is acyclic, would consider an (ill-formed) CD model with a single persistent class a that generalizes itself consistent with a DBS model with a table a.

To prevent the problem in the cd2dbs QVT-R transformation described above, one could tag each column with the path to the particular general they originated from, and then refine the DS domain pattern to prevent problematic recursive calls. A simpler alternative is to resort to the transitive closure operation, and map at once every declared or inherited attribute of a given class to a column of the matching table. In this new version of cd2dbs—that will be considered in the remainder of the chapter—A2C, PA2C and SA2C are replaced by the following alternative definition of A2C:

```
// AttributeToColumn
relation A2C {
    n:String; a:Attribute; g:Class;
    domain CD c:Class {} {
        (c->closure(general)->includes(g) or g = c) and
        g.attributes->includes(a) and a.name = n };
    domain DS t:Table {
        column = l:Column { name = n } };
}
```

The additional OCL constraint in the CD domain pattern acts as a pre-condition when applying the transformation in the direction of DS, and as a post-condition in the other direction. As such, it could not be specified in the **when** clause, since it would act as (an undesired) pre-condition for both scenarios. Figure 7.6 presents the embedding of this transformation as the **cd2dbs** typed relational constraint, after some formula simplifications to improve readability.

Unlike cd2dbs, the recursive version of hsm2nhsm, whose forward typed relational constraint embedding is presented in Figure 7.7 (tests in the HM direction omitted from $\phi_{\textbf{hsm2nhsm}}$), does produce the intended behavior. The reason is that, while a single attribute in cd2dbs may give rise to columns in multiple tables, an HSM transition in hsm2nhsm gives rise to a single NHSM transition. As a consequence, unlike A2C that must be defined over class elements, T2T can be defined directly over transition elements. These particularities are difficult to grasp at design time, thus effective tool support for QVT-R is essential for the design of consistency relations that embody the intentions of the user.

The recursive typed relational constraint in Figure 7.7 helps understanding why this embedding is not well behaved in the presence of circular recursion. The axiomatization of $\mathtt{S2S}_{\triangleright}$ states that a sub-state a is related with a state b if a is related to b by $\mathtt{SS2S}_{\triangleright}$,

$$\mathcal{S}_{\texttt{cd2dbs}} = \mathcal{S}_{CD} \cup \mathcal{S}_{DS}$$

$$D_{\texttt{cd2dbs}} = D_{CD} \cup D_{DS} \cup \{\texttt{P2S}_{\triangleright} : \texttt{Package} \leftrightarrow \texttt{Schema}, \texttt{P2S}_{\triangleleft} : \texttt{Package} \leftrightarrow \texttt{Schema}, \texttt{A2C}_{\triangleright} : \texttt{Class} \leftrightarrow \texttt{Table}, \texttt{A2C}_{\triangleleft} : \texttt{Class} \leftrightarrow \texttt{Table}\}$$

$$\phi_{\texttt{cd2dbs}} =$$

$$\forall\, p : \texttt{Package}_{CD} \mid (\exists\, s : \texttt{Schema}_{DS} \mid \texttt{name}_{CD} \circ p = \texttt{name}_{DS} \circ s) \wedge$$

$$\forall\, s : \texttt{Schema}_{DS} \mid (\exists\, s : \texttt{Package}_{CD} \mid \texttt{name}_{DS} \circ s = \texttt{name}_{CD} \circ p) \wedge$$

$$\forall\, p : \texttt{Package}_{CD}, s : \texttt{Schema}_{DS} \mid \langle p, s\rangle \in \texttt{P2S}_{\triangleright} \Rightarrow$$
$$\forall\, c : \texttt{classes}_{CD} \circ p \cap \texttt{persistent}_{CD} \mid (\exists\, t : \texttt{tables}_{DS} \circ s \mid$$
$$\texttt{name}_{CD} \circ c = \texttt{name}_{DS} \circ t \wedge \langle c, t\rangle \in \texttt{A2C}_{\triangleright}) \wedge$$

$$\forall\, p : \texttt{Package}_{CD}, s : \texttt{Schema}_{DS} \mid \langle p, s\rangle \in \texttt{P2S}_{\triangleleft} \Rightarrow$$
$$\forall\, t : \texttt{tables}_{DS} \mid (\exists\, c : \texttt{classes}_{CD} \circ p \cap \texttt{persistent}_{CD} \mid$$
$$\texttt{name}_{CD} \circ c = \texttt{name}_{DS} \circ t \wedge \langle c, t\rangle \in \texttt{A2C}_{\triangleleft}) \wedge$$

$$\texttt{P2S}_{\triangleright} = \{p : \texttt{Package}_{CD}, s : \texttt{Schema}_{DS} \mid \texttt{name}_{CD} \circ p = \texttt{name}_{DS} \circ s\} \wedge$$

$$\texttt{P2S}_{\triangleleft} = \{p : \texttt{Package}_{CD}, s : \texttt{Schema}_{DS} \mid \texttt{name}_{CD} \circ p = \texttt{name}_{DS} \circ s\} \wedge$$

$$\texttt{A2C}_{\triangleright} = \{c : \texttt{Class}_{CD}, t : \texttt{Table}_{DS} \mid$$
$$\forall\, g : \texttt{general}^{*}_{CD} \circ c, a : \texttt{attributes}_{CD} \circ g \mid$$
$$(\exists\, l : \texttt{columns}_{DS} \circ t \mid \texttt{name}_{DS} \circ l = \texttt{name}_{CD} \circ a)\} \wedge$$

$$\texttt{A2C}_{\triangleleft} = \{c : \texttt{Class}_{CD}, t : \texttt{Table}_{DS} \mid \forall\, l : \texttt{columns}_{DS} \circ t \mid$$
$$(\exists\, g : \texttt{general}^{*}_{CD} \circ c, a : \texttt{attributes}_{CD} \circ g \mid \texttt{name}_{DS} \circ l = \texttt{name}_{CD} \circ a)\}$$

Figure 7.6: QVT-R transformation `cd2dbs` as a TRC.

which happens if $a \circ \texttt{container}$ is related to b by $\texttt{S2S}_{\triangleright}$, and so on until a top-level state is reached. If $a \circ \texttt{container} = a$, which introduces circularity, there will be two valid valuations for $\texttt{S2S}_{\triangleright}$ and $\texttt{SS2S}_{\triangleright}$: one where $\texttt{S2S}_{\triangleright} = \{(a, b)\}$ and $\texttt{SS2S}_{\triangleright} = \{(a, b)\}$ and another where $\texttt{S2S}_{\triangleright} = \{\}$ and $\texttt{SS2S}_{\triangleright} = \{\}$. The latter is clearly incorrect and will lead to undesirable behavior. Since the meta-model enforces acyclic containment associations, and our constraint maintainers are constraint-aware, this embedding will be well-behaved.

$$\mathcal{S}_{\texttt{hsm2nhsm}} = \mathcal{S}_{HM} \cup \mathcal{S}_{NM}$$

$$\begin{aligned} D_{\texttt{hsm2nhsm}} = {} & D_{HM} \cup D_{NM} \ \cup \\ & \{\texttt{M2M}_{\triangleright} : \texttt{SMachine} \leftrightarrow \texttt{SMachine}, \texttt{M2M}_{\triangleleft} : \texttt{SMachine} \leftrightarrow \texttt{SMachine}, \\ & \ \texttt{S2S}_{\triangleright} : \texttt{State} \leftrightarrow \texttt{State}, \texttt{S2S}_{\triangleleft} : \texttt{State} \leftrightarrow \texttt{State}, \\ & \ \texttt{TS2S}_{\triangleright} : \texttt{State} \leftrightarrow \texttt{State}, \texttt{TS2S}_{\triangleleft} : \texttt{State} \leftrightarrow \texttt{State}, \\ & \ \texttt{SS2S}_{\triangleright} : \texttt{State} \leftrightarrow \texttt{State}, \texttt{SS2S}_{\triangleleft} : \texttt{State} \leftrightarrow \texttt{State}\} \end{aligned}$$

$$\begin{aligned} \phi_{\texttt{hsm2nhsm}} = {} & \\ & \forall\, s : \texttt{SMachine}_{HM} \mid (\exists\, t : \texttt{SMachine}_{NM} \mid \texttt{name}_{HM} \circ s = \texttt{name}_{NM} \circ t) \wedge \\ & \forall\, sm : \texttt{SMachine}_{HM}, tm : \texttt{SMachine}_{NM} \mid \langle sm, tm \rangle \in \texttt{M2M}_{\triangleright} \Rightarrow \\ & \quad \forall\, s : (\texttt{machine}^{\circ}_{HM} \circ sm) \mid (\exists\, t : (\texttt{machine}^{\circ}_{NM} \circ tm) \mid \\ & \qquad \textbf{if } (\texttt{container}_{HM} \circ s = \emptyset) \textbf{ then } \langle s, t \rangle \in \texttt{TS2S}_{\triangleright} \\ & \qquad\qquad \textbf{else } \langle s, t \rangle \in \texttt{SS2S}_{\triangleright}) \wedge \\ & \forall\, ht : \texttt{Transition}_{HM} \mid (\exists\, nt : \texttt{Transition}_{NM} \mid \\ & \quad \langle \texttt{source}_{HM} \circ ht, \texttt{source}_{NM} \circ nt \rangle \in \texttt{S2S}_{\triangleright} \wedge \\ & \quad \langle \texttt{target}_{HM} \circ ht, \texttt{target}_{NM} \circ nt \rangle \in \texttt{S2S}_{\triangleright}) \wedge \\ & \texttt{M2M}_{\triangleright} = \{ s : \texttt{SMachine}_{HM}, t : \texttt{SMachine}_{NM} \mid \texttt{name}_{HM} \circ s = \texttt{name}_{NM} \circ t \} \wedge \\ & \texttt{S2S}_{\triangleright} = \{ s : \texttt{State}_{HM}, t : \texttt{State}_{NM} \mid \\ & \quad \forall\, sm : \texttt{SMachine}_{HM}, tm : \texttt{SMachine}_{NM} \mid \langle sm, tm \rangle \in \texttt{M2M}_{\triangleright} \Rightarrow \\ & \qquad \langle s \rangle \in \texttt{machine}^{\circ}_{HM} \circ sm \Rightarrow \langle t \rangle \in \texttt{machine}^{\circ}_{NM} \circ tm \wedge \\ & \qquad\quad (\textbf{if } (\texttt{container}_{HM} \circ s = \emptyset) \textbf{ then } \langle s, t \rangle \in \texttt{TS2S}_{\triangleright} \\ & \qquad\qquad\qquad \textbf{else } \langle s, t \rangle \in \texttt{SS2S}_{\triangleright})\} \wedge \\ & \texttt{TS2S}_{\triangleright} = \{ s : \texttt{State}_{HM}, t : \texttt{State}_{NM} \mid \texttt{name}_{HM} \circ s = \texttt{name}_{NM} \circ t \} \wedge \\ & \texttt{SS2S}_{\triangleright} = \{ s : \texttt{State}_{HM}, t : \texttt{State}_{NM} \mid \langle \texttt{container}_{HM} \circ s, t \rangle \in \texttt{S2S}_{\triangleright} \} \wedge \\ & \ldots \end{aligned}$$

Figure 7.7: QVT-R transformation `hsm2nhsm` as a TRC.

7.3 Enforcement Semantics

Besides showing many ambiguities and omissions, we believe that, due to the reasons presented next, the enforcement semantics intended in the standard is quite undesirable. Instead, we propose an alternative that is easy to formalize, more flexible, and more predictable to the end-user, built on the formalization presented in Chapter 6.

7.3.1 Standard Enforcement Semantics

In the QVT-R standard, update propagation is required to be deterministic (OMG, 2011a, p. 18). This is a desirable property, since it makes its behavior more predictable. However, to ensure determinism, every QVT-R transformation is required to follow very stringent syntactic rules that reduce update translation to a trivial imperative procedure. Namely, it should be possible to order all constraints in a QVT-R relation in such a way that the value of every free variable is fixed by a previous constraint. Although not clarified in the standard, this means that every QVT-R relation invoked in **`when`** and **`where`** constraints is either invoked with previously bound variables, or required to be deterministic as well, even if those constraints were only intended to control relation application in order to render update propagation deterministic. For example, in QVT-R transformation `cd2dbs`, update propagation in the DS direction will only be deterministic for QVT-R relation `C2T` if at most one s is consistent with p according to QVT-R relation `P2S` (note that s is still free in the **`when`** clause). In this particular example that happens to be the desired behavior, but in general such determinism is undesirable since it forces QVT-R relations to be one-to-one mappings, limiting the expressiveness of the language. Moreover, it defeats the purpose of a declarative transformation language, since one is forced to think in terms of imperative execution and write more verbose transformations. For example, our simpler version of `A2C` using transitive closure would not be allowed, since the value of g is not known *a priori* when enforcing consistency in the direction of CD.

Another problem is the predictability of update propagation. Being deterministic is just part of the story—it should be clear to the user why some particular element was chosen to be updated instead of another. The only mechanism proposed by QVT-R to control updatability are *keys*. For example, one could add the statement **`key`** `Table (name, schema);` to the running example to assert that each table is uniquely identified by the pair of properties `name` and `schema`. If an update is required on a table to restore consistency (for example, when an attribute is added to an existing class), such key is used to find a matching table. When found, an update is performed, otherwise a new table is created. This works well when all domains involved in QVT-R relations have natural keys—which again points to one-to-one mappings only—but fails if such keys do not exist. In those cases, the standard prescribes that update propagation should always be made by means of creation of new elements, even if sometimes a simple update to an existing element would suffice. Since creation requires defaults

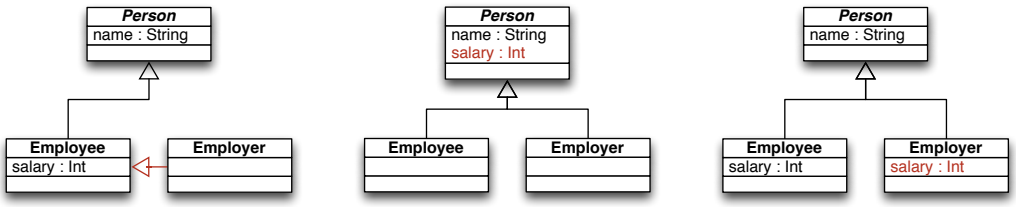
for mandatory (multiplicity one) properties, this would result in models with little resemblance with the original, which would probably end up being useless to the user.

7.3.2 Least-change Enforcement Semantics

Our alternative enforcement semantics is based on the principle of least-change, that promotes predictability by requiring updates to be as small as possible. QVT-R "check-before-enforce" policy is just a particular case of this more general principle. Following Section 6.1, given the embedding of the transformation domains (Section 6.1.2), of distances over conforming models (Section 6.3) and the embedding of the inter-model constraint (presented in this chapter for QVT-R transformations), we are able to deploy least-change constraint maintainers based on model finding. In this section we analyze the behavior of such constraint maintainer under the two concrete metrics presented in Section 6.3.1: GED and OBD. Note that least-change by itself does not ensure a single solution, although it substantially reduces the set of possible results. If among the returned models the user wishes to favor a particular subset, QVT-R keys or OCL constraints can be added to the meta-model to further guide the transformation engine.

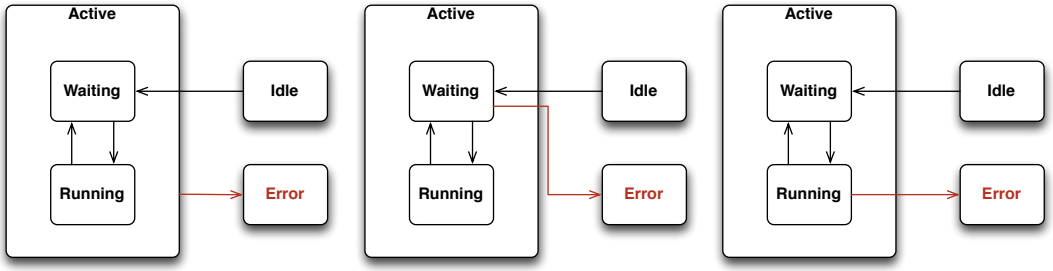
The choice of the metric directly affects the behavior of the bidirectional transformation, since the model finding procedure is guided by the implied preference order. Recall the class diagram and database schema presented at Figure 7.2 and imagine that the database manager decides that employers also have salaries, creating a `salary` column in the `Employer` table. Minimal repairs on the class diagram according to GED are either setting `Employee` as a super-class of `Employer` (Figure 7.8a) or to move the attribute `salary` from `Employee` up to `Person` (Figure 7.8b), both at distance 2. If none of these are desirable repairs, the user could eventually ask for the next closest solution at distance 3, which in this case is the introduction of a new attribute `salary` in `Employer` (Figure 7.8c).

Suppose the user wants to rule out all repairs that change the class hierarchy or assign the same cost to either create a new attribute or move an attribute from one class to another. To do so, she can specify (using OCL) which are the valid edit operations that can be performed to repair a model and propagate the update under OBD. For `CD` models, assume the existence of operations whose signature is presented in Figure 6.1a. Notice that there are no edit operations that modify the hierarchy, and both creation and moving of an attribute are now atomic edit operations. Through OBD our technique finds the minimal sequence of edit operations that repairs the model. In our company



(a) Class diagram cd_1. (b) Class diagram cd_2. (c) Class diagram cd_3.

Figure 7.8: Least-change propagation example for `cd2dbs`.



(a) HSM hs_1. (b) HSM hs_2. (c) HSM hs_3.

Figure 7.9: Least-change propagation example for `hsm2nhsm`.

running example, there will now be two minimal repairs, namely insert the new attribute `salary` in class `Employer` (Figure 7.8c), or moving the existing one from `Employee` to the common super-class `Person` (Figure 7.8b), which were considered at different distances under GED. As expected, the solution which set `Employer` as a sub-class of `Employee` has also been excluded.

For another example of the tradeoff between GED and OBD, consider the problem `hsm2nhsm` of expanding/collapsing hierarchical state machines. Imagine the user wants to allow the occurrence of errors and creates a new simple state `Error` on the collapsed diagram from Figure 7.5b and a transition to it from `Active`. Propagating this update back to the expanded state machine using GED would yield 3 possible solutions at minimal distance: the creation of the simple state `Error` with a transition to it from either the composite state `Active` (Figure 7.9a) or the sub-states `Waiting` (Figure 7.9b) or `Running` (Figure 7.9c), all at minimal cost. The user would be able to navigate through these solutions and select the most suitable one depending on the context.

Suppose however that the user prefers that transitions inserted in the collapsed state machine are reflected back only at the top-level states. If that is the case, she could

simply define an `addTransition` operation in such a way that it only allows the insertion of transitions between top-level states, without any other operation that introduces transitions (for instance, restricting the edit operations to those whose signatures are depicted in Figure 7.3a). In that case, if the previous update was propagated using the OBD metric, it would present only one solution at minimal distance, namely the insertion of a new transition from the composite state `Active` to `Error` (Figure 7.9a). We believe this combination of an automatically inferred metric and a user parameterizable one provides a high level of flexibility to our technique.

7.4 Multidirectional QVT-R Transformations

Heretofore, this chapter dealt with the scenario where QVT-R transformations relate two models. Nonetheless, the bidirectional scenario is not sufficient to tackle some applications. An arbitrary number of models may coexist in the same MDE environment, and their complex interrelationship may not be decomposable into a set of bidirectional relationships that can be maintained separately. As already shown in Section 6.2, the solver-based framework over which bidirectional transformations were formalized can be generalized to the *multidirectional transformation* scenario (Proposition 6.5).

To motivate the need for multidirectional transformation, consider the problem of keeping the consistency between a *feature model* and a set of k valid *configurations*, embodied by a multidirectional constraint maintainer $\texttt{fm2cf} : FM \bowtie CF_1 \bowtie ... \bowtie CF_k$. For the sake of simplicity, assume that feature models `FM` consist of named features, that may or not be mandatory, and configurations `CF` are simply a set of selected features; the respective meta-models are depicted in Figure 7.10. The inter-model constraint entailed by `fm2cf` can be decomposed into two parts $\texttt{fm2cf} = \texttt{mf} \cap \texttt{of}$: relation $\texttt{mf} : FM \leftrightarrow CF_1 \leftrightarrow ... \leftrightarrow CF_k$ (*MandatoryFeatures*) expresses that mandatory features match exactly the set of features appearing in every configuration; and relation $\texttt{of} : FM \leftrightarrow CF_1 \leftrightarrow ... \leftrightarrow CF_k$ (*OptionalFeatures*) expresses that the feature model contains at least the union of all selected features. Note that due to the intention of having features present in all configurations set as mandatory in the feature model, relation `mf` cannot be decomposed into k bidirectional relations between FM and each CF_i.

This kind of multidirectional scenario is already informally foreseen in the QVT-R standard, that admits an arbitrary number of transformation domains to be related in

(a) Meta-model CF. (b) Meta-model FM.

Figure 7.10: Class diagrams for the CF and FM meta-models.

a QVT-R transformation. However, the proposed checking semantics is too inflexible and only able to represent a restricted subset of consistency relations. In fact, none of the above relations can be specified using the standard checking semantics. Moreover, the standard hints at an enforcement semantics with very limited applicability. Namely, from a consistency relation such as `fm2cf`, the standard only prescribes the derivation of the following n transformations:

- $\overrightarrow{\texttt{fm2cf}}_{FM} : FM \times CF_1 \times ... \times CF_k \leftrightarrow FM$, for propagating updates to the configurations back to the feature model;
- $\overrightarrow{\texttt{fm2cf}}_{CF_i} : FM \times CF_1 \times ... \times CF_k \leftrightarrow CF_i$, for any $i \in [0..k]$, for propagating updates to the feature model and the remaining configurations into a specific target configuration.

That is, transformations may only propagate updates to a single target transformation domain. In the multidirectional scenario this is a very restrictive view of the enforcement semantics, as the user may wish to restore consistency in many ways depending on the context, leading to different update propagation transformations. Consider, for instance, the following interesting and alternative instantiations:

- $\overrightarrow{\texttt{fm2cf}}_{(CF_1,...,CF_k)} : FM \times CF_1 \times ... \times CF_k \leftrightarrow CF_1 \times ... \times CF_k$, which would allow updates to the feature model to be propagated to more than one configuration. For example, if a feature is changed to mandatory it must be selected in all configurations; this simple update could not be handled by the standard transformations, since full consistency could not be restored by an update over a single model.
- $\overrightarrow{\texttt{fm2cf}}_{(FM,CF_1,...,CF_{i-1},CF_{i+1},...,CF_k)} : FM \times CF_1 \times ... \times CF_k \leftrightarrow FM \times CF_1 \times ... \times CF_{i-1} \times CF_{i+1} \times ... \times CF_k$ for any $i \in [0..k]$, which would provide more flexibility in the propagation of updates to a configuration, as all the remaining artifacts are allowed to change. For example, if the name of a feature in a

configuration is changed, the natural way to recover consistency is to change the name of that feature in all the remaining configurations and in the feature model.

The main goal of this section is to start shedding some light on this currently unexplored multidirectional scenario. In particular, we explore the applicability of QVT-R for multidirectional model transformation and discuss some semantic issues that arise in this setting. To overcome its current limitations, we propose a simple extension that enables the specification of interesting multidirectional transformations, and discuss how to infer different kinds of consistency-restoring transformations from the *same* multidirectional specification.

7.4.1 QVT-R Multidirectional Checking Semantics

The abstract syntax of QVT-R relations between n transformation domains M_i essentially generalizes the one presented in Section 7.1.1 for the bidirectional scenario:

```
[top] relation R {
    [variable declarations]
    domain M_1 a_1 : A_1 { π_1^T } [{ π_1^C }];
    ...
    domain M_n a_n : A_n { π_n^T } [{ π_n^C }];
    [when { ψ }]
    [where { φ }]
}
```

Here, π_i (the conjunction of π_i^T and π_i^C) denotes a domain pattern over an element a_i from domain M_i (for $i \in [1..n]$), while ψ and ϕ are still arbitrary pre- and post-conditions. According to the standard, testing the consistency specified by top relations consists of running n directional tests (denoted by the subscript domain identifier as R_{M_i}, an extension of the bidirectional $R^{\blacktriangleleft}$ and $R^{\blacktriangleright}$ predicates), each validating one of the transformation domains. Thus, the embedding of a top multidirectional QVT-R relation R in a relational formula, denoted by $R^{\blacklozenge}$, consists of:

$$R^{\blacklozenge} \equiv \bigwedge_{i \in [1..n]} R_{M_i}$$

The semantics of these R_{M_i} predicates generalizes that of the bidirectional case with M_i as the target domain: if ψ holds, for all source domains elements a_j such that π_j

holds with $j \neq i$ (the source candidates), there must exist a target domain element a_i such that π_i and ϕ hold (the target candidate).

Back to our `fm2cf` running example, consider that we have a pair of configurations ($k = 2$ and $n = 3$). How can the `mf` consistency relation be specified in QVT-R? As a first attempt, let us consider the following specification:

```
// MandatoryFeatures
top relation mf {
    n : String;
    domain CF1 s1 : Feature { name = n };
    domain CF2 s2 : Feature { name = n };
    domain FM  f  : Feature { name = n,
                              mandatory = true };
}
```

The free variable n relates selected features in each configuration with mandatory features in the feature model. Following the standard, the consistency relation will consist of three predicates:

$$\mathtt{mf} \equiv \mathtt{mf}_{FM} \wedge \mathtt{mf}_{CF_1} \wedge \mathtt{mf}_{CF_2}$$

Given an embedding as typed relational constraints of the transformation domains, each of these directional tests is then concretized as:

$$\begin{aligned}
&\mathtt{mf}_{FM} \equiv \forall\, n : \mathtt{String}, s_1 : \mathtt{Feature}_{\boldsymbol{CF_1}}, s_2 : \mathtt{Feature}_{\boldsymbol{CF_2}} \mid \\
&\quad n = \mathtt{name}_{\boldsymbol{CF_1}} \circ s_1 \wedge n = \mathtt{name}_{\boldsymbol{CF_2}} \circ s_2 \Rightarrow \\
&\qquad (\exists\, f : \mathtt{Feature}_{\boldsymbol{FM}} \mid n = \mathtt{name}_{\boldsymbol{FM}} \circ f \wedge \langle f \rangle \in \mathtt{mandatory}_{\boldsymbol{FM}}) \\
&\mathtt{mf}_{CF_1} \equiv \forall\, n : \mathtt{String}, f : \mathtt{Feature}_{\boldsymbol{FM}}, s_2 : \mathtt{Feature}_{\boldsymbol{CF_2}} \mid \\
&\quad n = \mathtt{name}_{\boldsymbol{CF_2}} \circ s_2 \wedge n = \mathtt{name}_{\boldsymbol{FM}} \circ f \wedge \langle f \rangle \in \mathtt{mandatory}_{\boldsymbol{FM}} \Rightarrow \\
&\qquad (\exists\, s_1 : \mathtt{Feature}_{\boldsymbol{CF_1}} \mid n = \mathtt{name}_{\boldsymbol{CF_1}} \circ s_1) \\
&\mathtt{mf}_{CF_2} \equiv \ \ldots
\end{aligned}$$

But let us concretely analyze the meaning of these predicates. Relation $\mathtt{mf}_{FM}$ expresses part of the intended behavior—if the two configurations have the same selected feature then such feature is mandatory. However, the reverse implication requiring every mandatory feature to be present in both configurations is not entailed by $\mathtt{mf}_{CF_1}$ and $\mathtt{mf}_{CF_2}$. Looking at $\mathtt{mf}_{CF_1}$, the selection of the s_1 feature depends both on f and s_2, thus, if there are *no* selected features in cf_2, $\mathtt{mf}_{CF_1}$ will be trivially true due to the

empty range in the universal quantification. This problem persists whatever the domain patterns (and pre- or post-conditions), and as a consequence, the intended specification for `mf` cannot be realized by any QVT-R relation (with the standard semantics) between features in the three models.

This problem could be easily solved if we could control the extent of the universal quantifications in the semantics of $\texttt{mf}_{CF_1}$ and $\texttt{mf}_{CF_2}$, namely to range only over the feature model and ignore the remaining configurations

$$\begin{aligned}
&\texttt{mf}_{CF_1} \equiv \forall\, n : \texttt{String}, f : \texttt{Feature}_{\boldsymbol{FM}} \mid \\
&\quad n = \texttt{name}_{\boldsymbol{FM}} \circ f \wedge \langle f \rangle \in \texttt{mandatory}_{\boldsymbol{FM}} \Rightarrow \\
&\qquad (\exists\, s_1 : \texttt{Feature}_{\boldsymbol{CF_1}} \mid n = \texttt{name}_{\boldsymbol{CF_1}} \circ s_1) \\
&\texttt{mf}_{CF_2} \equiv \forall\, n : \texttt{String}, f : \texttt{Feature}_{\boldsymbol{FM}} \mid \\
&\quad n = \texttt{name}_{\boldsymbol{FM}} \circ f \wedge \langle f \rangle \in \texttt{mandatory}_{\boldsymbol{FM}} \Rightarrow \\
&\qquad (\exists\, s_2 : \texttt{Feature}_{\boldsymbol{CF_2}} \mid n = \texttt{name}_{\boldsymbol{CF_2}} \circ s_2)
\end{aligned}$$

The conjunction of these predicates entails the missing part of the desired `mf` specification, and hints at a possible extension to the standard checking semantics that largely improves its expressiveness, as described in the next section.

Although `mf` could alternatively be interpreted as a bidirectional constraint maintainer $FM \bowtie CF_1 \times ... \times CF_k$ between a feature model and a tuple of configurations, in general the n models may be of different nature. In fact, as will be shown below, `of` would still not be expressible. Moreover, the checking semantics prescribed by the QVT-R standard could not be reproduced under such interpretation.

7.4.2 Extending the Standard Semantics

As the previous section makes clear, the standard QVT-R language is not suitable for expressing many transformations of interest, namely those where the relationships between the domains are not symmetric. In fact, this is already a problem in the bidirectional setting (for example, how can plain subset relationships be expressed?), but is aggravated in the multidirectional setting due to the explosion of possible dependencies between transformation domains. In this section, we propose precisely to extend QVT-R with a language of dependencies between transformation domains in order to express the desired directionality of the checking semantics.

Checking dependencies Let σR denote the set of domains $\{ M_1, ..., M_n \}$ in a QVT-R relation R. A *checking dependency* $S \rightarrow t$ for R, where $S \subseteq \sigma R$ is a set of domains and $t \in \sigma R$ a single domain (with $(t \notin S)$), states that the transformation domain t depends on all the transformation domains in S. Formally, the semantics of a rule R according to a dependency $S \rightarrow t$, denoted by $R_{S \rightarrow t}$, prescribes that R should be checked by universally quantifying over elements of all the domains in S and, when the respective domain patterns and pre-condition hold, demanding an element satisfying the respective domain pattern and post-condition to exist in the t domain. The set of checking dependencies attached to a relation R will be denoted by $\underline{R}$. The semantics of a top relation R is now the conjunction of all dependency checks:

$$R^{\blacklozenge} \equiv \bigwedge_{d \in \underline{R}} R_d$$

For example, to obtain the desired specification of the `mf` relation, it suffices to attach to the above QVT-R specification the dependencies $\underline{\texttt{mf}} = \{ CF_1\ CF_2 \rightarrow FM, FM \rightarrow CF_1, FM \rightarrow CF_2 \}$. This extension is conservative, in the sense that the standard QVT-R semantics can still be specified by setting:

$$\underline{R} = \bigcup_{i \in [0..n]} \{ (\sigma R \setminus M_i) \rightarrow M_i \}$$

The `of` relation, stating that the selected features from both configurations should be included in the set of all available features (mandatory or optional), which is also not expressible under standard QVT-R semantics, can be represented by the following QVT-R relation

```
// OptionalFeatures
top relation of {
    n : String;
    domain CF1 s1 : Feature { name = n };
    domain CF2 s2 : Feature { name = n };
    domain FM  f  : Feature { name = n };
}
```

associated with the checking dependencies $\underline{\texttt{of}} = \{ CF_1 \rightarrow FM, CF_2 \rightarrow FM \}$. Note that, at this point, we are just disregarding the dependencies implied by the QVT-R standard. Expressing our dependency would require some sort of extended QVT-R

syntax.

Although at first sight this extension may seem too conservative, the fact is that from these simple dependencies more complex ones can be built. In particular, multiple model dependencies can be attained through the entailment $\{ M_1 \rightarrow M_2, M_1 \rightarrow M_3 \} \vdash \{ M_1 \rightarrow M_2\ M_3 \}$ (thus resulting in the expected $\texttt{mf}_{(CF^1, CF^2)}$) while dependencies over unions of models can be attained through $\{ M_1 \rightarrow M_3, M_2 \rightarrow M_3 \} \vdash \{ M_1 \mid M_2 \rightarrow M_3 \}$ (from which $\texttt{of}_{FM}$ arises), where $\{ M_1 \mid M_2 \rightarrow M_3 \}$ denotes the fact that the domain M_3 depends on information from M_1 and M_2 independently.

Asymmetric bidirectionality This extension also improves the expressiveness of QVT-R in the bidirectional setting. While the prevalent idea is that the QVT-R standard forces checking semantics to be bidirectional (i.e., run the test in both directions) (Bradfield and Stevens, 2012), this requirement may be too strong in some contexts. In fact, some ambiguities in the standard allow different interpretations and ModelMorf (Tata Research Development and Design Centre), the tool that allegedly follows the QVT-R standard the closest, allows unidirectional checks. Consider the `uml2rdbms` transformation. While in the bidirectional version, only extra classes not matched by any relation are disregarded (insertion of a non-persistent class does not introduce inconsistencies), in the unidirectional version any extra class not related to a table is disregarded (insertion of a class, even if persistent, never causes inconsistencies). Clearly, this is undesirable in `uml2rdbms` transformation, and would be as well in `hsm2nhsm`.

However, this is not always the case. Consider for instance the consistency relation between UML class diagrams and UML sequence charts. One of the basic consistency constraints between these models is that all classes mentioned in the sequence chart must exist in the class diagram; however, not all classes in the class diagram must be represented in the sequence chart. This kind of consistency relation would be impossible to specify with QVT-R's *forall-there-exists* bidirectional checks, unless the classes which are mentioned in the sequence chart were somehow (artificially) marked in the class diagram, so that a pattern to filter them out can be defined in the consistency relations. Using the domain dependencies introduced for the multidirectional scenario, this would be trivially solved: just introduce an asymmetric dependency from sequence charts to class diagrams.

Relation invocations As in the bidirectional scenario, relation calls in the multidirectional scenario must preserve the direction of the calling predicate. However, the QVT-R

syntax does not guarantee that every relation in a specification can be run in the same direction, e.g., nothing prevents a relation R with $\sigma R = \{ CF_1, ..., CF^k, FM \}$ running in the FM direction from calling another relation S with $\sigma S = \{ CF_1, ..., CF^k \}$, which does not relate the FM transformation domain. The standard is omissive about these situations. The newly introduced checking dependencies must also be taken into consideration, e.g., should a relation $\underline{R} = \{ M_1 \rightarrow M_2 \}$ be allowed to call another relation $\underline{S} = \{ M_2 \rightarrow M_1 \}$? We think the answer should be no, and this situation should be flagged as a typing error at static-time.

Notwithstanding, it is worth noting that the dependencies of R and S need not be perfect matches. In fact, a relation $\underline{R} = D$ may be called in the direction R_d by another relation with $d \in \underline{S}$ if $D \vdash d$, i.e., D entails d. In our restricted language this will allow, for instance, the call $R_{M_1 \rightarrow M_3}$ when $\underline{R} = \{ M_1 \rightarrow M_2, M_2 \rightarrow M_3 \}$, since $\{ M_1 \rightarrow M_2, M_2 \rightarrow M_3 \} \vdash M_1 \rightarrow M_3$. Since our dependencies are equivalent to *Horn clauses* (disjunctions with a single positive literal) this "type-checking" can be done in linear time.

7.4.3 QVT-R Enforcement Semantics

In Section 6.2 we showed that the same technique used to deploy least-change bidirectional constraint maintainers over model finders could also be adapted to the multidirectional scenario. Under this technique, for any selection of target domains σR_t, deploying a transformation $\overrightarrow{R}_{\sigma R_t}$ requires only the definition of a suitable distance function for the selected target transformation domains t.

Let us use the `fm2cf` example to explore the transformation space. Those with a single output model can be trivially applied. For instance, assuming a model distance Δ^{FM}, $\overrightarrow{\texttt{fm2cf}}_{FM} : CF_1 \times ... \times CF_k \leftrightarrow FM$ would produce a feature model fm' consistent with the input configurations (`fm2cf` $(cf_1, ..., cf_k, fm')$) and closest to the original feature model (minimizing distance Δ^{FM} between fm and fm'). Similarly for every $\overrightarrow{\texttt{fm2cf}}_{CF_i}$. As for the tuple returning transformations, a simple way to achieve the combined distance of the target models is to add up the distance between every model, as already exemplified at the end of Section 7.3.2. Of course, depending on the defined order, changes in each domain may be assigned different weights (e.g., in $\overrightarrow{\texttt{fm2cf}}_{(FM,CF_1,...,CF_k)} : FM \times CF_1 \times ... \times CF_k \leftrightarrow FM \times CF_1 \times ... \times CF_k$ changes to configurations could be prioritized over those to feature models).

When applying a transformation, the user must be aware that not all update directions

are able to restore the consistency of the system. Consider, for instance, that a new mandatory feature is introduced in the feature model. Then $\overrightarrow{\texttt{fm2cf}}_{CF_i}$, which updates a single model, will clearly not be able to restore consistency of the MDE environment. Instead, the user should apply $\overrightarrow{\texttt{fm2cf}}_{(CF_1,\ldots,CF_k)}$ and update all configurations.

7.5 Discussion

Combining the QVT-R embedding presented in this chapter with the model finding formalization from Chapter 6 results in a QVT-R bidirectional model transformation technique, that supports both the standard checking semantics and a clear and precise enforcement semantics based on the principle of least-change. It also supports meta-models annotated with OCL constraints and the specification of allowed edit operations, which allows its applicability to non-trivial domains and provides a fine-grained control over the selection of the updated model. The main restriction is that recursion must be non-circular (or well-founded), which is satisfied by most of the interesting MDE scenarios.

Tool support for QVT-R transformations is scarce, ModelMorf and Medini being the main existing functional tools. ModelMorf (Tata Research Development and Design Centre) allegedly follows the QVT-R standard closely, since its development team was involved in the specification of the standard (although its concrete semantics is unknown). However, the development of the tool seems to have stopped. Medini (ikv++ technologies ag) is an Eclipse plugin for a subset of the QVT-R language. Although popular, its (unknown) semantics admittedly disregards the semantics from the QVT-R standard (it does not have a checkonly mode, for instance). To support incremental executions, it stores explicit traces between elements of the two models. None of these tools has support for OCL constraints on the meta-models. Other prototype tools have been proposed but once again the implemented semantics are not completely clear. Moment-QVT (Boronat et al., 2006) is an Eclipse plugin for the execution of QVT-R transformations by resorting to the Maude rewriting system; de Lara and Guerra (2009) have proposed the embedding of QVT-R in colored Petri nets. These tools support only unidirectional transformations, in the sense that they ignore the pre-state of the target model. As such, they are not able to retrieve information not present in the source, leading to the generation of fresh new models every time the transformation is applied. Once again, none supports OCL constraints on the meta-

model. Greenyer and Kindler (2010) have discussed the possible implementation of QVT-R transformations in TGGs. While some TGGs tools prevent loss of information by supporting incremental executions (Giese and Wagner, 2009) or partial matches between domains (Greenyer et al., 2011; Hermann et al., 2013), this embedding focuses only on the embedding of QVT-R specifications in the TGG architecture, disregarding the consequences on the enforcement semantics. There is also work in progress on providing execution functionalities to the Eclipse *QVT Declarative* project, which is currently able to parse and edit QVT-R specifications, by the successive transformation of such specifications into lower-level languages (Willink et al., 2013). However, the impact of these transformations on the QVT-R semantics is still not clear.

A technique that follows an approach similar to ours is the JTL tool by Cicchetti et al. (2010), although it does not support QVT-R, but rather a restricted "QVT-like" language. Like ours, JTL generates models by resorting to a solver (the DLV solver), which is able to retrieve some (unquantified) information from the original target. However, it is not clear how the solver chooses which information to retrieve or how the new model is generated. It also forces the totality of the transformation, returning inconsistent models in case there is no consistent solution.

Regarding standard QVT-R enforcement semantics, there has recently been an attempt to formalize it by Bradfield and Stevens (2013), following previous work on the checking semantics (Stevens, 2013; Bradfield and Stevens, 2012). As prescribed in the standard, to enforce the *forall-there-exists* semantics, the procedure consists of a creation phase of new target elements whenever a source element does not have a matching target (or modification of existing ones, if keys are used), followed by a deletion phase, to remove target elements that are no longer matching a source element. These phases occur only at top-level relations, as **`when`** and **`where`** are assumed to be predicates that top-level quantified elements must comply. This procedure does not take into consideration additional constraints on the meta-model, in particular no specific technique is proposed to fill-in mandatory attributes and associations of newly created (or modified) elements, taking into account such meta-model constraints, and **`when`** and **`where`** clauses (likewise to our technique, the usage of solvers is hinted as a possible solution). Since it closely follows the standard, this semantics also suffers from the problems already described in Section 7.3.

Two approaches have been proposed for the validation of QVT-R transformations that also rely on solvers. Garcia (2008) proposes the use of Alloy Analyzer to verify the

correctness of QVT-R specifications, in order to guarantee that the output is well-formed and avoid run-time errors. Cabot et al. (2012) propose inferring OCL invariants of the *forall-there-exists* shape from QVT-R transformations (much like the checking semantics), that allow the validation of QVT-R specifications under a set of properties. It supports OCL constraints in the meta-model and recursive calls are translated to recursive OCL specifications. However, these approaches are not focused on enforce mode and its semantics, and do not analyze the behavior of the transformation for concrete input models, which is the focus of our embedding. Although not the current focus of our embedding, since it is based on model finding it would be straight-forward to support functionalities dedicated to automatically check specific properties of model transformations—like the fact that they are total, deterministic, or that they always produce well-formed models.

To the best of our knowledge, there exists no previous work dedicated to the study of multidirectional QVT-R transformations. Therefore, validation of our approach would require the collection of reasonable and realistic case studies, in order to explore the expressive power of our dependencies extension. It is important to emphasize that such extension to QVT-R specifications is also useful to express asymmetric problems heretofore undefinable in the bidirectional transformation scenario. This issue has also be tackled by Stevens (2014) when studying bidirectional transformation frameworks with tolerance for inconsistencies: in this case, having the check predicates hold in both directions could be defined to yield *perfect* consistency, while holding in a single direction would yield *partial* consistency. We have purposely left out subjective considerations about the most adequate syntactic extensions to the QVT-R language for expressing our proposed checking dependencies, and have focused primarily on the multidirectional semantics. We are currently considering several syntactic extensions to allow the specification of the checking dependencies in QVT-R.

The strength of our approach to QVT-R is that, as we believe is intended, the consistency relation entailed by a QVT-R transformation is at the core of the bidirectionalization procedure, giving rise to transformations that are correct by construction and, thanks to the formalization from Chapter 6, to predictable least-change semantics.

Chapter 8

Bidirectionalizing ATL Transformations

In the previous chapter we provided least-change bidirectional transformation semantics to a bidirectional model transformation language, by deploying it over model finding procedures as formalized in Chapter 6. In particular, we focused on the QVT-R language that was designed with the intent of allowing bidirectional transformation. Thus, since QVT-R transformations embody the concept of constraint maintainer by having a consistency relation at the core of bidirectional transformation, deriving a suitable inter-model constraint from that specification allowed us to enforce transformations that were correct and least-change by construction. However, not all model transformation languages were designed with bidirectional concerns. Nonetheless, even if the focus of such languages is to "simply" generate an output model from an input model, it is easy to envision scenarios where bidirectional behavior would be useful. The original and the generated models may be expected to coexist in the MDE environment, and in that context, the unidirectional transformation entails an implicit traceability between the two models: an update on the freshly generated output model could render it inconsistent with the original input model. By being able to infer a backward transformation from the unidirectional transformation, one could propagate the update of the output back to the original input, restoring the consistency of the environment. Since such task is outside the scope of unidirectional transformation languages, providing such languages with bidirectional semantics is an active research topic.

The *ATLAS Transformation Language* (ATL) (Jouault and Kurtev, 2005) is a widely used model transformation language created to answer the original QVT RFP, and

thus shares some characteristics with the standardized QVT languages. Unlike QVT-R, ATL has *de facto* standard operational semantics implemented as a plugin for the Eclipse IDE[1]. However, it is unidirectional, in the sense that a transformation between two domains M and N only specifies how to create an N model from an M model. The prescribed method to obtain bidirectional transformations with ATL is to write two unidirectional transformations. Unfortunately, this leads to obvious correctness and maintenance problems that motivate the use of bidirectional transformation, since the language provides no means to check that they are inverses of each other, nor to automatically derive one from the other. Moreover, unlike QVT-R, ATL is not able to natively retrieve information from the pre-state of the output model, and thus obtaining real bidirectional behavior for non-bijective transformations would require the design of complex unidirectional transformations that take as input both M and N models. With that goal in mind, studies have been made on providing ATL transformations with bidirectional semantics (Xiong et al., 2007; Sasano et al., 2011). The main drawback of these approaches is that the bidirectional semantics that results from the bidirectionalization process is not clear, resulting in backward transformations whose relationship with the original specification is unclear.

Following the perspective that has been driving this part of the dissertation, we advocate a different approach to the bidirectionalization of unidirectional transformation languages: if we are able to derive the notion of consistency between the input and output models from the unidirectional transformation, we are able to maintain this consistency through the mechanisms from Chapter 6. Since ATL specifications are mainly declarative (although there are some imperative constructs), the ATL language proves to be a suitable case study. This approach is fundamentally different from those followed in (Xiong et al., 2007; Sasano et al., 2011): instead of interpreting the bidirectional transformations as lenses, providing backward semantics for unidirectional semantics, we try to infer the consistency relation underlying the unidirectional transformation, achieving least-change bidirectional semantics by maintaining such constraint.

This gives rise to the question: *can inter-model consistency constraints be inferred from unidirectional transformations and subsequently be deployed as constraint maintainers?* In this chapter we try to explore this issue using the declarative subset of the ATL transformation language as proof-of-concept. All the techniques explored in Chapter 7 for the embedding of intra-model constraints, distance functions and

[1] http://www.eclipse.org/atl/.

multidirectional transformations still hold in this context; in this chapter we restrict ourselves to the core problem of bidirectional inter-model constraints.

The contributions of this chapter are summarized as follows:

- we explore the best suited schemes for the embedding of *unidirectional transformations* into our formalization of bidirectional constraint maintainers (Section 8.2);
- we propose a translation from *ATL transformations* into relational inter-model constraints (Section 8.3), whose constraint maintainers are able to keep models generated by ATL transformations consistent.

These presentations are prepended by an introduction to the ATL language (Section 8.1) and succeeded by a discussion of the overall results (Section 8.4).

8.1 ATL Language

ATL is a hybrid language with both declarative and imperative constructs. The authors advocate that transformations should be declarative whenever possible, and imperative specifications should only be used if specifying the transformation declaratively proves to be difficult (Jouault and Kurtev, 2005). In this work we restrict ourselves to declarative constructs, disregarding imperative ones (known in ATL as *called* rules and *do* blocks)[2].

ATL transformations are defined from a set of input transformation domains to a set of output transformation domains. For the scope of this presentation we restrict ourselves to a single output domain, although the presented concepts could be extended to the multidirectional scenario much like was done for QVT-R in Section 7.4. Thus, we address ATL transformations of the shape $\overrightarrow{t} : M \rightarrow N$ for models $m : M$ and $n : N$; we assume $\overrightarrow{t}$ to denote the standard ATL transformation. The main constituents of ATL transformations are *rules*, the ATL equivalent to QVT-R relations, whose abstract syntax is:

```
[[unique] lazy] rule R {
```

[2]Giving semantics to imperative constructs in relational logic is doable (see, for example, (Near and Jackson, 2010) in the context of Alloy), but the need to explicitly represent all intermediate updates to the models in execution traces would deem our solver based approach completely unfeasible, in particular in presence of loops.

```
    from a   : A (π_M)
    to b     : B (φ)
}
```

In our context, rules are defined from a single input element a from the input domain to a single output element b from the output domain. Candidate input elements are selected by an OCL pattern π_M over their properties, while output patterns consist of bindings ϕ over the output element, which may refer to the input element. Roughly, the execution semantics creates an output element for every input element that matches π_M. Input models are read-only and output models write-only, and thus transformations are not able to take into consideration existing elements in the output model, not even being able to "check-before-enforce" (although more recent work on incremental ATL executions by Jouault and Tisi (2010) could eventually be used to address this issue).

Default rules are called *matched rules* and must be executed for all elements of the input model (similarly to QVT-R top relations). *Lazy rules*, unlike matched rules, are only executed if explicitly called from other rules (similarly to QVT-R non-top relations). Lazy rules can either be *unique* or not: in unique lazy rules an input element is always matched to the same output element, no matter how many times the rule is called over that input element; in non-unique lazy rules a new output element is created every time it is called. Our bidirectionalization technique focuses on matched rules and unique lazy rules.

One particular characteristic of ATL is that output bindings may rely on implicit traceability links between elements created by matched rules. Output elements may be directly "assigned" input elements, in which case traces are used to retrieve the corresponding output element. This is possible because execution is divided in two phases: the first phase binds input elements to the input patterns and creates the output elements, implicitly creating traces between them; the second phase applies the bindings to the output elements, resorting to the traces created in the previous phase if necessary. Since an input element cannot be matched by more than one rule (ATLAS group), there is no ambiguity in the selection of the output element for each input element. Lazy rules must be explicitly called and thus are not taken into consideration in implicit resolutions.

Figure 8.1 presents an ATL version of the `hsm2nhsm` transformation using transitive closure. Rule `S2S` relates every top-level state (filtered by an OCL pattern) in the HM transformation domain to a state with the same name in the opposing domain NM,

```
module hsm2nhsm;
create NM : NHSM from HM : HSM;

// SMachineToSMachine
rule M2M {
    from hm : HSM!SMachine
    to nm   : NHSM!SMachine ( name <- hm.name )
}

// StateToState
rule S2S {
    from hs : HSM!State ( hs.container->isEmpty() )
    to ns   : NHSM!State (
        name    <- hs.name,
        machine <- hs.machine
    )
}

// TransitionToTransition
rule T2T {
    from ht : HSM!Transition
    to nt   : NHSM!Transition (
        source  <- ht.source->closure(container)->any(container->isEmpty()),
        target  <- ht.target->closure(container)->any(container->isEmpty()),
        machine <- ht.machine
    )
}
```

Figure 8.1: The `hsm2nhsm` ATL transformation.

while rule `T2T` maps every transition in HM to a transition in NM between the top-level containers of its source and target states. These top-level containers are retrieved by filtering the result of the closure operation over `container` by a select operation. Rule `M2M` simply maps state machines in HM to state machines with the same name in NM. Note how in `T2T`, `HSM` states from the HM input domain are being attributed to the source and target of `NHSM` transitions in the NM output domain: the rule is taking advantage of the implicit traces created by `S2S` between the states of HM and NM. A similar situation occurs in `S2S` (and `T2T`) when binding the owner state machine: the state machine of *ns* is directly assigned the state machine of *hs*, which is expected to have been previously bound by `M2M`.

8.2 Bidirectionalization Technique

To bidirectionalize unidirectional transformations following the technique from Chapter 6, there is the need to derive a consistency relation $T : M \leftrightarrow N$ from a forward transformation $\overrightarrow{t} : M \rightarrow N$, and then use it to determine suitable (inverse) transformations according to the least-change semantics proposed in Section 6.1.

Since we are given the forward transformation $\overrightarrow{t} : M \rightarrow N$, one could imagine that it would suffice to derive a suitable backward transformation $\overleftarrow{t} : M \times N \leftrightarrow N$, thus lifting unidirectional transformations to the framework of lenses with $t : M \blacktriangleright N$, in which case each bidirectional transformation would be comprised of a pair of transformations $\mathsf{get}_t = \overrightarrow{t} : M \rightarrow N$ and $\mathsf{Put}_t : M \times N \leftrightarrow M$. This was attempted before by Sasano et al. (2011) for ATL transformations. The lens framework is designed to deal with transformations that are abstractions (i.e., surjective transformations), as implied by the asymmetric nature of the two transformations: the view n can always be derived solely from a source m as it contains less information. In particular, if a model n is updated to an n' that falls outside the range of get_t, the behavior of Put_t is undetermined. Such well-behaved lens could be obtained in our least-change constraint maintainer framework, by setting the forward transformation as an implicit consistency relation as $T\ (m, n) \equiv n = \mathsf{get}_t\ m$.

Unfortunately, this imposes some undesirable limitations in the allowed usage scenarios. Consider a very simple example where `World` models consists of a set of named persons, and `Company` models consists of a set of named employees having an (optional) salary (Figure 8.2). Consider also a trivial ATL transformation $\overrightarrow{\texttt{employ}} : WD \rightarrow CP$ from `World` models to `Company` models that maps every person to an employee with the same name and without a salary assigned (Figure 8.3). This transformation is clearly not surjective since it only targets the subset of `Company` models where employees have no assigned salary. Now, consider a model $wd : WD$ with a single person p and the corresponding model $cp : CP$ created by $\mathsf{get}_{\texttt{employ}}$. If the user updates cp to cp' by assigning a salary to p, there will be no valid wd' such that $cp' = \mathsf{get}_{\texttt{employ}}\ wd'$, and thus cp' would be an invalid model. This limitation would greatly reduce the updatability of the framework.

A possible solution to this problem would be to weaken the lens laws, as suggested in (Sasano et al., 2011), by allowing $\mathsf{Put}_t\ (m, n)$ to produce a source m' whose view $n' = \mathsf{get}_t\ (m')$ is not n (breaking acceptability) as long as propagating n' backward produces m' again, i.e., forcing only weak-acceptability PUTGETPUT to hold. However,

(a) Meta-model `World`. (b) Meta-model `Company`.

Figure 8.2: Class diagrams of the `World` and `Company` meta-models.

```
module employ;
create CP : Company from WD : World;

// PersonToEmployee
rule P2E {
    from p : World!Person
    to e   : Company!Employee ( name <- p.name )
}
```

Figure 8.3: The `employ` ATL transformation.

even if the above update is now allowed, if the user updates the wd model (for example, by inserting a new person) and wishes to propagate such change to the cp model, the forward transformation $\text{get}_{\texttt{employ}}$ would erase the previously assigned salary of p, since it is not incremental. Embedding a unidirectional transformation in a lens framework with such weakened laws presumes that once $\overrightarrow{t}$ is run to generate a new target model from a source, subsequent updates can only be safely propagated backwards. (Chapter 4 provides a more thorough discussion on the consequences of non-surjectivity in lens frameworks.)

To overcome this limitation we opt instead to embed unidirectional transformations in the framework of constraint maintainers, likewise to QVT-R. The main idea is to infer from $\overrightarrow{t} : M \to N$ a consistency relation $T : M \leftrightarrow N$ that does not consider such unbound properties, in the sense that every model m is considered consistent not only with $\overrightarrow{t}\ m$, but also with any model that extends $\overrightarrow{t}\ m$ by assigning values to properties not bound by $\overrightarrow{t}$. This of course implies that $\overrightarrow{t} \subseteq T$. From T a new forward transformation $\overrightarrow{T} : M \times N \leftrightarrow N$ and a backward transformation $\overleftarrow{T} : M \times N \leftrightarrow M$ can then be derived to propagate updates in both directions, using the least-change semantics described in Section 6.1 (obviously satisfying both the correctness and hippocraticness laws), resulting in a well-behaved exhaustive constraint maintainer $T : M \bowtie N$. Back to the example from Figure 8.3, since $\overrightarrow{\texttt{employ}}$ does not bind the `salary` attribute of employees in CP models, the WD with a single person p would be consistent with any

CP with a single employee p, whatever the assigned salary (if any). Section 8.3 will explore how to infer one such possible T from $\overrightarrow{t}$.

The bidirectional framework obtained with this technique satisfies the following properties. First, since $\overrightarrow{t} \subseteq T$, the consistency relation trivially holds for pairs of models (m, n) such that $n = \overrightarrow{t}\ m$. Second, if $\overrightarrow{t}$ is surjective, applying either $\overrightarrow{t}$ or $\overrightarrow{T}$ to an updated source will yield the same updated target: in this case, $\overrightarrow{t}$ completely defines the target elements, thus T only relates a model m to $\overrightarrow{t}\ m$. In this case, the pair of transformations $\overrightarrow{t}$ and $\overleftarrow{T}$ will form a well-behaved lens. In contrast, for non-surjective transformations this is no longer the case. It is easy to see why by considering the toy example from Figure 8.3: by updating a view $cp = \overrightarrow{t}\ wd$ to cp' with the insertion of a salary, wd and cp' will still be consistent by `Employ`, and thus neither $\overrightarrow{\texttt{Employ}}$ nor $\overleftarrow{\texttt{Employ}}$ will update the models; applying $\overrightarrow{\texttt{employ}}$ however would revert cp' back to cp. In this case $\overrightarrow{\texttt{employ}}$ and $\overleftarrow{\texttt{Employ}}$ do not form a well-behaved lens, satisfying only weak-acceptability. Thus, while the derivation of a constraint maintainer T does not invalidate the use of $\overrightarrow{t}$ paired with $\overleftarrow{T}$—the pair comprises a stable and weakly-acceptable lens—due to its non-incrementality, $\overrightarrow{t}$ is better suited for batch transformations when fresh models are created, at which point $\overrightarrow{T}$ and $\overleftarrow{T}$ can be used to propagate updates and maintain consistency.

8.3 Inferring a Consistency Relation

As in the QVT-R case, the technique from Chapter 6 requires the derivation of a typed relational constraint $\boldsymbol{T}$ from a unidirectional transformation T, in particular a formula $\phi_{\boldsymbol{T}}$ and any additional relation variables $D_{\boldsymbol{T}}$ (sorts $\mathcal{S}_{\boldsymbol{T}}$ are assumed to be retrieved from the transformation domains' embedding). This section will propose a technique to derive such inter-model constraint from ATL transformations. At first glance, the semantics of an ATL transformation shares some similarities with the checking semantics of QVT-R described in Section 7.2: pattern matching is used to filter candidate input elements, and it resembles the *forall-there-exists* quantification pattern to relate input and output elements. There are also some apparent differences: it is a directional semantics, in the sense that the above *forall-there-exists* quantification in principle should only be checked in the direction of the output model (the direction of the transformation), and the existential quantifier should be unique, that is, for all candidate input elements, there must exist *exactly one* output element built with the output bindings (matched

rules create an output element for every input candidate). However, there are some subtle differences: as the following example will show, the *forall-there-exists* semantics cannot be realized using quantifiers, and explicit traceability links must be used instead; furthermore, checks must be also performed in the opposite direction to avoid the occurrence of spurious output elements.

Consider again the simple transformation from Figure 8.3, and the following semantics for the rule P2E with universal and uniqueness quantifications:

$$\forall\ p : \mathtt{Person}_{WD} \mid (\exists^1\ e : \mathtt{Employee}_{CP} \mid \mathtt{name}_{WD} \circ p = \mathtt{name}_{CP} \circ e)$$

Given an input model wd with two persons with the same name a, this semantics would force a consistent output model cp with exactly one employee with name a. This is obviously not the intended ATL semantics, as two employees with the same name would be created by the ATL transformation, one for each input person. Obviously, relaxing the uniqueness constraint of the existential quantifier will not solve the problem, as an arbitrary number of employees would be allowed. The one-to-one mapping between candidate input and generated output elements cannot be realized by quantifiers, but requires an explicit traceability relation between them. Moreover, this directional check does not guarantee that the only employees in the output model are the ones created by the transformation, and some check in the opposite direction must be performed to ensure that every employee originates from a person. As in the case of QVT-R relation invocations (Section 7.2), such traceabilities must be represented through fresh relation variables in the model finding problem: for a matching rule R from input elements A to output elements B, a relation $R^{\diamond} : A \leftrightarrow B$ is created and axiomatized by an embedding $R^{\blacklozenge}$. For instance, for P2E, this would result in the following embedding:

$$\begin{aligned}\mathtt{P2E}^{\blacklozenge} \equiv\ & \forall\ p : \mathtt{Person}_{WD} \mid (\exists^1\ e : \mathtt{Employee}_{CP} \mid \\ & \quad \langle p, e\rangle \in \mathtt{P2E}^{\diamond} \wedge \mathtt{name}_{WD} \circ p = \mathtt{name}_{CP} \circ e) \wedge \\ & \forall\ e : \mathtt{Employee}_{CP} \mid (\exists^1\ p : \mathtt{Person}_{WD} \mid \\ & \quad \langle p, e\rangle \in \mathtt{P2E}^{\diamond} \wedge \mathtt{name}_{WD} \circ p = \mathtt{name}_{CP} \circ e)\end{aligned}$$

This constraint ensures that there exists a traceability relation $\mathtt{P2E}^{\diamond} : \mathtt{Person} \leftrightarrow \mathtt{Employee}$ between every person (since π_{WD} is empty) to a unique employee with the same name, and vice-versa. Note that this formalization also relates output elements that fall outside the range of $\overrightarrow{\mathtt{employ}}$, namely those that have the attribute salary defined.

In general, the semantics of a matched rule R can be specified as follows:

$$R^{\blacklozenge} \equiv \forall\ a : A_{\boldsymbol{M}} \mid \pi_M \Rightarrow (\exists^1\ b : B_{\boldsymbol{N}} \mid \langle a, b\rangle \in R^{\triangleleft\triangleright} \wedge \phi) \wedge \\ \forall\ b : B_{\boldsymbol{N}} \mid (\exists^1\ a : A_{\boldsymbol{M}} \mid \langle a, b\rangle \in R^{\triangleleft\triangleright} \wedge \pi_M \wedge \phi)$$

This forces $R^{\triangleleft\triangleright}$ to be an one-to-one relation between every candidate input element and a single corresponding valid (according to ϕ) output element. The first expression states that every $a : A_{\boldsymbol{M}}$ that matches the pattern π_M must be related to a single $b : B_{\boldsymbol{N}}$ with the bindings ϕ; the second expression states that every $b : B_{\boldsymbol{N}}$ must be related to a valid $a : A_{\boldsymbol{M}}$. The binding ϕ in a rule assigns to the output element values derived from the input element: unlike in QVT-R target patterns, all variables in ϕ must be previously assigned in the input pattern π_M. As such, they are interpreted likewise to **`where`** conditions in QVT-R. Although both invoked relations in QVT-R and matched rules in ATL rely on derivation of trace relations, their axiomatization is considerably different; this arises from the fact that, unlike in QVT-R, ATL traceabilities are one-to-one.

Implicit rule calls in the bindings, that are allowed in ATL, are translated as follows in our embedding. Let a property of type B be bound to an expression e. If e has a primitive type or is an output element, then $e : B$ must hold, and e is directly translated to the relational expression. If e denotes an input element of type A, we retrieve the matching output element of type B from the respective traceability relation $R^{\triangleleft\triangleright} : A \leftrightarrow B$ (notice that it is always possible to uniquely determine $R^{\triangleleft\triangleright} : A \leftrightarrow B$, since input elements of a given type are restricted to be matched by a single rule (ATLAS group)). Traceabilities can also be implicitly called over collections, as in the `T2T` transformation. Currently these implicit calls are supported for collections that are sets, which are directly representable in relational logic, in which case the above procedure is applied to every element e in the set.

Regarding unique lazy rules, although their semantics also relies on explicit traceability links, there is a subtlety that prevents their encoding in a similar way to (regular) matched rules: only a subset of the input elements (those over which a unique lazy rule has been called) will belong to the one-to-one traceability. Since this set is impossible to calculate at static-time, these traceabilities can not be axiomatized by "forall-there-exists" formulas like $R^{\blacklozenge}$. Thus, the semantics of these rules will be divided in two parts. Given a unique lazy rule R from A to B, its embedding $R_{\blacklozenge}$ that axiomatizes the respective traceability $R_{\triangleleft\triangleright} : A \leftrightarrow B$, will only enforce the correctness of the respective traceability relation, with the existence and uniqueness checks being deferred to the rule call:

$$R_{\blacklozenge} \equiv \forall\ a : A_{\boldsymbol{M}}, b : B_{\boldsymbol{N}} \mid \langle a, b \rangle \in R_{\lhd\rhd} \wedge \pi_M \Rightarrow \phi$$

Then, when a call $R\ (e)$ of the unique lazy rule R over an expression $e : A$ occurs, the following additional constraints is inserted, to ensure that the trace $R_{\lhd\rhd}$ between the value of e and the returned matched element b exists and is unique:

$$\exists^1\ b : B_{\boldsymbol{N}} \mid \langle e, b \rangle \in R_{\lhd\rhd} \wedge (\forall\ a : A_{\boldsymbol{M}} \mid \mathcal{S}_{T\lhd\rhd}\ (a, b) \Rightarrow a = e)$$

Where $\mathcal{S}_{T\lhd\rhd}$ denotes the union of all unique lazy traces in T. The first part of the conjunction states that a is uniquely matched to a b by $R_{\lhd\rhd}$, and the second that b is not being matched to any other element by any other rule.

Finally, for the embedding $\boldsymbol{T}$ of an ATL transformation T with matched rules $\mathcal{T}_T$ and unique lazy rules $\mathcal{S}_T$, the formula denoting when two models are consistent is defined as

$$\phi_{\boldsymbol{T}} = \bigwedge_{R \in \mathcal{T}_T} R^{\blacklozenge} \wedge \bigwedge_{R \in \mathcal{S}_T} R_{\blacklozenge}$$

while the newly introduced variables amount to

$$D_{\boldsymbol{T}} = \bigcup_{R \in \mathcal{T}_T} R^{\lhd\rhd} \wedge \bigcup_{R \in \mathcal{S}_T} R_{\lhd\rhd}$$

Once the inter-model constraint is derived, the corresponding transformations can be deployed following the formalization over model finding presented in Chapter 6.

8.4 Discussion

In this chapter we applied our technique to the bidirectionalization of unidirectional model transformations, using ATL as a case study. Rather than providing a fully-fledged system, the intent of this presentation was to emphasize the flexibility of the model finder-based approach.

Some previous work has been done toward the bidirectionalization of ATL. Xiong et al. (2007) infer a synchronization procedure from a subset of the byte-code produced by the ATL compiler, in the sense that both the input and output model can be updated in order to restore consistency. However, it is not clear how the restrictions imposed on the byte-code are reflected on the higher-level ATL language. Sasano et al. (2011) interpret models as graphs and ATL declarative rules as UnCAL operations over said graphs, which are bidirectionalized in the GRoundTram bidirectional graph transformation

system (Hidaka et al., 2011). However, the supported subset of the ATL language is much more limited than ours, namely matching rules cannot have input patterns and output bindings must comply to a very limited OCL subset, excluding rule calls (implicit or explicit). As discussed in Section 8.2, if we see our ATL bidirectional transformations as lenses, the bidirectional properties from (Sasano et al., 2011) also hold in our framework. None of these approaches are concerned with enforcing least-change updates.

The relationship between ATL and QVT has previously been explored by Jouault and Kurtev (2006). However, the focus of that work was on aligning the architecture of the two languages, without any semantic considerations. They suggest that interoperability could be attained by mapping both QVT-R (through the translation to QVT Core) and ATL to QVT Operational, one of the languages proposed by OMG (2011a) that is imperative and based on operational mappings. In fact, it has already been shown by Stevens (2013) that the translation from QVT-R to QVT Core does not preserve the semantics of the transformation.

Similar to the work by Cabot et al. (2012) where OCL invariants were inferred from QVT-R transformations to validate their properties, Büttner et al. (2012) have developed a technique for the verification of ATL transformations. As in the QVT-R scenario, our underlying model finding framework could be naturally extended to support such functionalities.

This chapter has shown that our model finding-based technique allows not only the bidirectionalization of unidirectional model transformation languages, but does so while providing clear and predictable semantics. Moreover, this deployment is expected to be associated with the embedding of meta-models and their intra-model constraints following the procedure from Section 6.1.2, guaranteeing that the generated solutions are well-formed, and may also be extended in a straightforward manner to support multiple input models, following techniques similar to the ones presented for multidirectional QVT-R in Section 7.4.

Chapter 9

The Echo Framework

In the previous chapters we have been developing a technique for the deployment of typical MDE tasks on top of model finding. Such technique relied on a relational representation of the various artifacts—intra- and inter-model constraints, as well as the distance functions—so that they could be processed by relational model finders like Kodkod. However, to be useful in practice, this technique should be offered to the user as backend, requiring from her only the management of artifacts in standard MDE formats.

To that goal we have implemented Echo, a tool for model repair and transformation based on model finding that aims at providing a seamless integration with the MDE environment. Echo is deployed as an Eclipse plugin, developed on top of the popular *Eclipse Modeling Framework* (EMF), with meta-models being specified in Ecore with embedded OCL constraints. Its engine works by translating both meta-models (annotated with OCL) and QVT-R and ATL transformations to Alloy (Jackson, 2012), a lightweight formal specification language implemented over Kodkod, with support for automatic model finding via SAT solving.

While less scalable that some of the existing tools, Echo is: more *expressive*, allowing the annotation of meta-models with rich OCL constraints and the specification of bidirectional model transformations; more *flexible*, being able to check and repair both intra- and inter-model consistency, and providing control over repairs by letting the user specify the allowed edit operations; *correct*, in the sense that resulting models are always fully consistent; and *minimal*, in the sense that it follows the clear and predictable principle of least-change according to the two metrics presented in Chapter 6, GED and OBD. In fact, Echo has already proved effective in debugging existing

transformations, namely helping us unveiling several errors in the well-known object-relational mapping that illustrates the QVT-R specification (OMG, 2011a), that were explored in Section 7.2.

The goal of this chapter is to present and evaluate Echo, with a focus on the translation of the MDE artifacts to relational logic. Alloy specifications can be subjected to automated analysis through an embedding into Kodkod provided by the Alloy Analyzer, with support for bounded model checking and model finding. Alloy has a type system that supports overloading and sub-typing, which allows the detection of many erroneous expressions that can render the specification trivially unsatisfiable (Edwards et al., 2004) prior to dispatching the solver. This makes it more suitable for the interactive development of specifications, while Kodkod is more suitable as an engine for automated analysis. The object-oriented type system, associated with a graph visualizer for the presentation of solutions provided by the Alloy Analyzer, brings Alloy even closer to the MDE environment justifying our choice of Alloy over Kodkod. In fact, this connection has already been previously explored, through the embedding of UML class diagrams and OCL constraints into Alloy (Anastasakis et al., 2010; Kuhlmann and Gogolla, 2012; Cunha et al., 2013). Alloy has also been used to reason about static properties of QVT-R specifications (Garcia, 2008), while the possibility to use Kodkod for intra-model consistency repair has also been explored (Kleiner et al., 2010; Straeten et al., 2011).

This chapter unfolds as follows, presenting different components of Echo:

- we present an *overview* of the main capabilities of the Echo framework (Section 9.1), that aims at supporting typical MDE tasks;
- we present Echo's *architecture* (Section 9.2), with EMF artifacts at the interface layer and model finding procedures at the core;
- we show how the typed relational constraints representing the artifacts are translated to Alloy *specifications* (Section 9.3), so that they can be effectively managed by model finding procedures;
- we present a technique for the derivation of Alloy *themes* from the MDE environment (Section 9.4), improving the readability of the proposed repaired models;
- we *evaluate* the performance of Echo (Section 9.5).

Finally, Section 9.6 draws conclusions about Echo's effectiveness.

9.1 Echo Overview

The focus of the Echo framework is to help users develop and keep their models consistent in the context of MDE. It is able to manage both intra-model (i.e., the conformity of a model with its meta-model) and inter-model consistency (i.e., the relationship between several models entailed by bidirectional transformations, the focus of this dissertation). In both cases, Echo is able to detect and repair inconsistencies. Below is a more detailed list of Echo's current capabilities.

Consistency check Concerning intra-model consistency, for a transformation domain M defined by a meta-model possibly attached with OCL constraints, Echo can automatically check whether a model m conforms to M, that is, $m : M$ holds. This can be done for newly created models or every-time the user updates an existing model.

Model repair If the intra-model consistency of model m is broken, for example because some of the OCL constraints in M is violated, then Echo can automatically suggest minimally repaired models $m' : M$, that is, it can find consistent models m' at minimum GED or OBD from m. Various alternatives for repaired models are presented to the user in increasing distance to the original model, from among which the user is able to choose the preferred one.

Model visualization To help the user decide which model she prefers, they can be depicted as graphs by resorting to the Alloy Analyzer's visualizer, as seen in Figure 9.1. For better readability, an Alloy theme is automatically inferred from the MDE environment (as described below in Section 9.4). A user-defined theme can also be provided if desired.

Model generation To help kickstart model development, Echo can also be used to generate a fresh minimal model $m : M$ (notice that often models cannot be empty due to meta-model constraints), or to generate scenarios for meta-model validation, that is, models parameterized by particular scopes and/or additional OCL constraints targeting specific configurations.

Inter-model consistency check Concerning inter-model consistency, given a QVT-R or ATL transformation T between M and N and two models $m : M$ and $n : N$, Echo can automatically check whether m and n are T-consistent, that is,

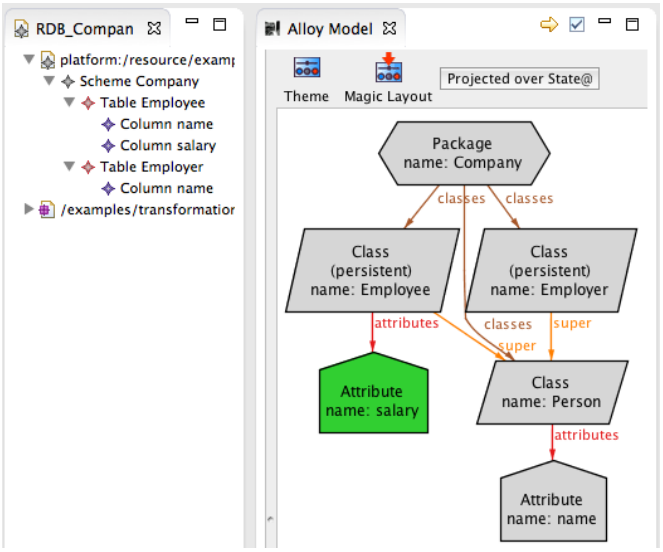


Figure 9.1: A snapshot of Echo, with DBS and CD models depicted in EMF and in the Alloy visualizer.

$T\ (m, n)$ holds, following the standard-compliant checking semantics presented in Section 7.2 for QVT-R and the one proposed for ATL in Section 8.3.

Inter-model consistency repair Given a transformation T between M and N and two models $m{:}M$ and $n{:}N$ such that $\neg T\ (m, n)$, Echo can perform a minimal update to one of the models selected by the user in order to recover consistency, for example produce $n' : N$ such that $T\ (m, n')$. This repair follows the enforcement semantics satisfying the principle of least-change proposed in Chapter 6, under GED or OBD. Likewise to intra-model consistency recover, the user is able to choose the desired repaired model from among all minimal consistent models.

Batch transformation Finally, given a transformation T between M and N and a model $m : M$, Echo can produce a fresh minimal model $n : N$ such that $T\ (m, n)$ (likewise for the opposite direction). These batch transformations are useful at early phases of model-driven software development, when the user has developed a source model, from which she wishes to derive a first version of the target model. Afterward, consistency-restoring updates can be performed to any of the models, by resorting to the same transformation.

While also available as a command-line application, Echo's main distribution platform is as a plugin for the Eclipse IDE, which automates the features just presented. Echo's environment consists of a set models, conforming to OCL-annotated

meta-models, and a set of inter-model constraints specified by QVT-R and ATL transformations. Each model is thus restricted by the intra-model constraint entailed by the meta-model and any number of inter-model constraints simultaneously. The Echo plugin was designed to be used in an *online setting* (Hu et al., 2008), in the sense that the consistency tests are automatically applied as the user is editing the models and, thus, updates are expected to be incremental, leaving the original models as unmodified as possible. Every time the user updates a model, the system automatically checks its consistency in relation to the other artifacts. If a model is flagged as inconsistent, the plugin displays an inconsistency error and proposes possible fixes, all within the Eclipse's standard environment. As there may be more than one consistent model at minimal distance, Echo presents all possible models in succession using the Alloy Analyzer's visualizer, allowing the user to choose the desired one, at which time the update is effectively applied to the model instance. If none of the minimal solutions is chosen, Echo presents models at increasingly higher distances from the original.

9.2 Architecture

As should be expected, the consistency checking and repair tasks are implemented using the technique presented in Chapter 6, given the embedding of QVT-R and ATL transformations presented in Chapter 7 and Chapter 8, respectively. Model generation and batch transformations are obtained using the same technique, given empty models as the initial state. Nonetheless, this solving layer of the tool is hidden from the user, that is expected to handle only artifacts and tools that are familiar to MDE practitioners.

To that purpose, Echo is deployed as plugin for the Eclipse IDE. It is built on top of the *Eclipse Modeling Framework* (EMF)[1], and resorts to the *Model Development Tools* (MDT) component to parse OCL formulas and to the *Model-to-Model Transformation* (MMT) component to parse QVT-R (through the QVT Declarative project) and ATL specifications. EMF prescribes Ecore—Eclipse's version of MOF—for the specification of meta-models, while model instances are presented as XMI resources. The left-hand side of Figure 9.1 depicts a DBS model instance in the Eclipse editor. To enhance the meta-models with additional constraints, we follow the technique proposed by MDT, of embedding the OCL constraints in meta-model annotations. These are translated to relational logic following the technique previously developed in (Cunha

[1] http://www.eclipse.org/modeling/.

family	operations
base	=, <>, **oclIsKindOf**, **oclAsType**, **@pre**, **allInstances**, **if-then-else**, **oclIsNew**
integer	+, -, >, <, <=, >=
boolean	**and**, **or**, **not**, **implies**, **true**, **false**
set	**size**, **includes**, **includesAll**, **excludes**, **excludesAll**, **isEmpty**, **notEmpty**, **union**, -, **intersection**, **including**, **excluding**, **asSet**
iterators	**exists**, **forAll**, **one**, **any**, **collect**, **select**, **reject**, **closure**
QVT	**opposite**

Table 9.1: Supported OCL operations.

et al., 2013). Table 9.1 summarizes the currently supported operations from the OCL standard library (OMG, 2012) (operation `oclIsNew` may only be used in controlled contexts, as will be explained in Section 9.3.5). The QVT standard extends the OCL language with the insertion of the **`opposite`** keyword that allows the navigation of associations in the opposite direction.

Currently, least-change is obtained though the external iterative mechanism proposed in Section 6.3.1. The two model metrics presented there are supported: GED, derived for every provided meta-model, and OBD, derived for meta-models with user-defined edit operations specified by OCL pre- and post-conditions. A source of ambiguity in operations defined in OCL concerns frame conditions, the problem of detecting which portions of the model are modified by an operation. Assuming that everything that is not mentioned in the post-condition is not changed is generally a reasonable assumption, but this is not trivial to infer from declarative specifications. Given the lack of OCL statements focusing on frame conditions, we introduce "*modifies*" clauses, through which the user must explicitly specify which elements of the model may be modified by the operation—the remainder are assumed to remain unchanged. This mechanism is similar to those introduced by behavioral interface specification languages, like the *Java Modeling Language* (JML) (Leavens et al., 2006). For instance, the operation `setName` from `Class` could be defined by the following OCL specification:

```
context Class::setName(n : String)
  post name
    self.name = n;
  post frame_class_name
    Class.allInstances()->forAll(c | c.name@pre = c.name or c = self)
  modifies Class::name
```

Here, the **modifies** keyword states that only the attribute `name` in `Class` is modified by operation `setName`. This specification gives rise to the $\|\texttt{setName}\|$ formula presented at Section 6.3.1.

To promote inter-operability, EMF processes models into an abstract syntax, which are persisted as XMI resources. Thus, as a model-to-model transformation tool over EMF, Echo is only able to directly process models represented in XMI, much like the other MMT components (like those with support for QVT-R and ATL). Echo's core engine can also be used directly as a library, in which case models are expected to be already parsed into the EMF's abstract syntax. Nonetheless, EMF has a wide support for domain-specific languages presented in a concrete syntax, which can be directly harnessed by Echo. The currently prescribed mechanism to convert models from concrete to abstract syntax is through Xtext[2], a language processing framework that provides parser generators as well as full integration with the Eclipse IDE through custom code editors. As an example, to parse QVT-R transformations, Echo translates QVT-R specifications following the QVT standard's concrete syntax to EMF's abstract syntax by relying on the MMT functionalities built over Xtext.

The plugin processes these artifacts as typed relational constraints, which are then translated into Alloy so that they can be deployed as model finding procedures. The following translations, summarized at Figure 9.2, are at the core of Echo:

Ecore → TRC embeds Ecore meta-models into typed relational constraints, following the technique proposed by Cunha et al. (2013) adapted as exposed in Section 6.1.2;

OCL → TRC translates the OCL constraints over the meta-model (and OCL expressions that may occur in QVT-R/ATL transformations) to typed relational constraints, following the technique proposed by Cunha et al. (2013);

QVT-R → TRC embeds QVT-R transformations into typed relational constraints as presented in Chapter 7;

[2] http://www.eclipse.org/xtext/.

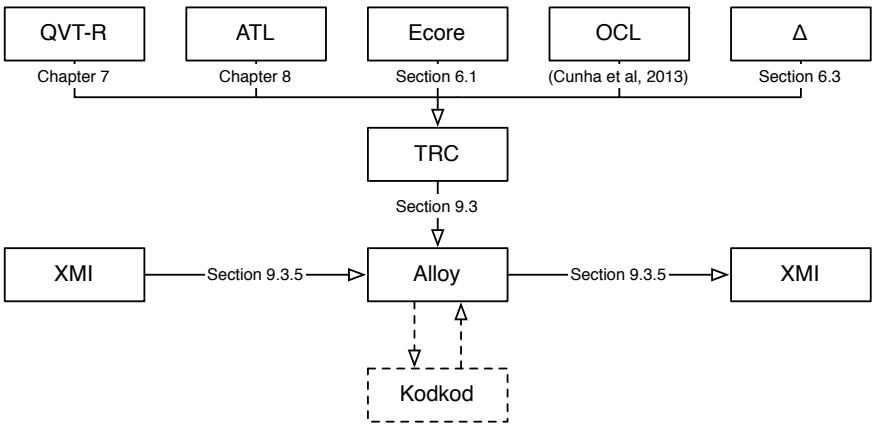


Figure 9.2: Echo's architecture.

ATL → TRC embeds ATL transformations into typed relational constraints as presented in Chapter 8;

TRC ↔ Alloy embeds typed relational constraints into Alloy specifications, as will soon be presented in Section 9.3;

XMI ↔ Alloy converts XMI model instances to and from Alloy, as will soon be presented in Section 9.3.5.

9.3 Embedding TRCs in Alloy

In the previous chapters it has been shown how meta-models, inter-model constraints and metric distances can be embedded in typed relational constraints and subsequently deployed as model finding problems. This section explores the last deployment step, that of translating typed relational constraint into Alloy specifications, which in turn are translated to relational model finding problems in Kodkod by the Alloy Analyzer.

9.3.1 A Brief Introduction to Alloy

Alloy is a rich and flexible language; in this section we focus only on concepts deemed essential for the scope of this dissertation.

Alloy specifications are developed in *modules*, that consist of *paragraphs*: signature declarations, constraints and commands. A *signature* declaration introduces a set of elements sharing a similar structure and properties. In Alloy such elements are

uninterpreted, immutable and indivisible, and are thus referred to as *atoms*. A signature declaration may also introduce *fields*, i.e., relations that connect its atoms to those of other (or the same) signatures. These are represented as sets of tuples of atoms in instances. Alloy is not restricted to binary relations, and it is not uncommon to have fields that relate three or more signatures. A signature that **`extends`** other signatures inherits their fields. It can also be contained **`in`** another signature, in which case it is simply a subset of the parent signature.

Signatures may be annotated with *multiplicity* keywords to restrict their cardinality, namely **`some`** (at least some elements), **`lone`** (at most one element), and **`one`** (exactly one element). The range signature in a field declaration can also be annotated with such multiplicities, to restrict the number of atoms that can be connected to each atom of the source signature. If that number is arbitrary, the special multiplicity keyword **`set`** should be used. Alloy's multiplicity tags draw similarities with the binary relation taxonomy presented in Section 2.4.

Facts specify properties that must hold in every instance. These may call *functions* and *predicates*, that are essentially containers for reusable expressions. *Commands* are used to perform particular analyses, by invoking the underlying solver. *Run* commands try to find instances for which the specified properties hold, while *check* commands try to find counter-examples that refute them. Commands can be parametrized by scopes for the declared signatures, thus bounding the search-space for the solver. If no scope is specified a default of 3 is assumed.

Figure 9.3 depicts a possible (incomplete) specification of the `CD` class diagram meta-model in Alloy. Signatures `Package`, `Class` and `Attribute` declare the corresponding classes and introduce (binary) fields to represent the classes' attributes and associations. Alloy does not have a primitive boolean type, so boolean attributes are usually represented by subset signatures containing the elements that have the attribute set to true. This is the case of the `persistent` attribute of `Class`, here represented by the `Persistent` subset signature. The run command instructs the analyzer to search for instances conforming to the `acyclic` predicate, setting a specific scope for each of the signatures. This is similar to the scope $s : \mathcal{S} \rightharpoonup \mathcal{T}$ that must be provided to deploy typed relational constraints as model finding problems (Section 6.1).

Formulas in Alloy are defined in relational logic. Everything in Alloy is a *relation*, i.e., a set of tuples of atoms (with uniform arity). Signatures are unary relations (sets) containing the respective atoms and scalar values (including quantified variables) are

```
module CD

abstract sig NamedEntity {
  name    : one String
}

sig Package extends NamedEntity {
  classes : set Class
}
sig Attribute extends NamedEntity {}
sig Class extends NamedEntity {
  attribute : set Attribute,
  general   : lone Class
}
sig Persistent in Class {}

pred acyclic {
  all self:Class | self not in self.^general
}

run { acyclic }
  for 3 Class, 3 Attribute, 1 Package, 3 String
```

Figure 9.3: A (static) specification of CD in Alloy.

just singleton sets. This uniformity of concepts leads to a very simple semantics. The relational logic operators also favor a navigational style of specification that is appealing to software engineers, as it resembles object-oriented languages.

When the relational logic formalism was defined in Chapter 2, we were careful to define a language close to both Kodkod and Alloy. Thus, the embedding of relational formulae into Alloy is rather straightforward: the similarities can be inspected by comparing the `acyclic` predicate in Figure 9.3 back with ACYCLICGEN in Section 2.1. The key operator in Alloy is the *dot join* composition that allows the navigation through fields and amounts to relational composition. For example, if `c` is a `Class`, `c.name` denotes its name (a scalar) and `c.general` accesses its super-class (a set containing at most one `Class`). The direction of the navigation operator in Alloy is the same as the one in OCL, and opposed to the composition from relational logic. Besides composition, relational expressions can also be built using the union (`+`), intersection (`&`), difference (`-`), and cartesian product (`->`) operators. In particular, singleton tuples can be defined by taking the cartesian product of two (or more) scalars. Relations can also have their domain restricted to a given set (`<:`) and likewise for the range (`:>`). Binary relational expressions can also be reversed ($\sim$), extended with the transitive closure (`^`), or with

the reflexive transitive closure ($*$). Relational expressions may also be created by set comprehension. Finally, there are some primitive relations pre-defined in Alloy: **`univ`** denotes the *universe*, i.e., the set of all atoms, **`none`** denotes the empty set, and **`iden`** the binary identity relation over the universe.

Alloy has limited support for integers: the pre-defined **`Int`** signature contains all available integers. In commands, the scope of **`Int`** determines the available number of bits to represent them (in two's complement notation). This reflects the bitwise representation of integers down in the SAT solver already mentioned in Section 6.3.1. Integers can be added and subtracted with the functions **`plus`** and **`minus`**, respectively. The default semantics for integer operations is wrap around: for example, if the scope for **`Int`** is 3, **`plus`**`[3,1]` is `-4`. However, a technique that suppresses solutions where overflow occurs has been proposed by Milicevic and Jackson (2012), which has been implemented in the Alloy Analyzer as an optional *Forbid Overflow* feature. Every relation expression can have its cardinality determined with the `#` operator.

Atomic formulas are built from relational expressions using inclusion (**`in`**), equality (`=`), or cardinality checks (besides **`lone`**, **`some`**, and **`one`**, keyword **`no`** can also be used to check whether a relational expression is empty). Formulas can be combined with conjunction (`&&`), disjunction (`||`), implication (`=>`), possibly associated with an **`else`** formula, equivalence (`<=>`), and negation (**`not`**). Besides the universal (**`all`**) and existential (**`some`**) quantifiers, Alloy also supports **`lone`** (property holds for at most one atom), **`one`** (property holds for exactly one atom), and **`no`** (property holds for no atom) quantifiers.

9.3.2 Embedding Intra-model Constraint TRCs

In our framework, a transformation domain M is assumed to be embedded in a typed relational constraint $\langle \phi_M, \mathcal{S}_M, D_M \rangle$ following the technique presented at Section 6.1.2 for translating meta-models and a translation inspired by the one proposed in (Cunha et al., 2013) for the embedding of OCL constraints into relational logic. This section explores how such intra-model constraint can be deployed in Alloy.

Since Alloy instances are built from immutable atoms, well-known idioms have been developed to capture the evolution of models. The *local state idiom* (Jackson, 2012) is one such idiom, where a special signature is introduced to represent each meta-model, whose atoms will denote different models (or different states of a model). To each field (representing an association or an attribute) an extra column of this type

is added, to allow its value to change in different models. The translation proposed in (Cunha et al., 2013) follows this idiom, but was extended to allow the set of atoms present in each model state to vary. For a signature `A`, we will refer to this field as the signature's *state field* and denote it by `A_` to avoid overloading. Boolean attributes are encoded similarly: a binary field captures which objects have the attribute set to true in each model.

To produce Alloy specifications that follow typical design patterns, typed relational constraints are also interpreted under the local state idiom. The main difference is that, while in two typed relational constraints $\boldsymbol{M}^1$ and $\boldsymbol{M}^2$ representing a transformation domain M, every relation $R : A_1 \leftrightarrow ... \leftrightarrow A_n$ is duplicated as $R_{\boldsymbol{M}^1}$ and $R_{\boldsymbol{M}^2}$, in Alloy's local state idiom it only gives rise to a single field `R : A1 -> ... An -> M` whose atoms `M` denote the state of the transformation domain: given two atoms `M1,M2 : M` representing $\boldsymbol{M}^1$ and $\boldsymbol{M}^2$ respectively, the valuations of $R_{\boldsymbol{M}^1}$ and $R_{\boldsymbol{M}^2}$ are retrieved by `R.M1` and `R.M2` respectively. This allows the derivation of a single Alloy module from a meta-model `M`, no matter how many typed relational constraints are derived from it. Sorts $\mathcal{S}_M$ (along with the hierarchy entailed by $\sqsubseteq_{\mathcal{S}}$) can be directly translated to Alloy signatures. Enumerations are translated to abstract "static" signatures (in the sense that they are present in every model, having no state field), while enumeration literals are singleton signatures that extend that signature.

Figure 9.4 presents the embedding of the CD typed relational constraint (Figure 6.2) in Alloy. The $\mathcal{S}_{CD}$ sorts `NamedEntity`, `Package`, `Class` and `Attribute` are directly translated to signatures, as well as the class hierarchy denoting `NamedEntity` as the abstract super-class. An extra signature `CD` is also created to denote the different CD typed relational constraints: each declared relation R_{CD} from D_{CD} is assigned the extra `CD` domain during translation. For each sort A, the set A_{CD} denoting the elements present in each CD state is captured by the state field `A_` of signature `A`.

Formula ϕ_{CD} is translated to a predicate `CD_Constraint` that takes as an argument a `CD` state: for an atom `CD1 : CD`, `CD_Constraint[CD1]` tests whether model `CD1` conforms to ϕ_{CD}. Translation of this formula is rather straightforward since our relational formalism is already quite close to that of Alloy. The notable difference is that the formula is also translated under the local state idiom, having the model state passed as the predicate parameter applied to each field call. The embedding of the relations multiplicity takes advantage of the connection between the terminology for binary relations presented at Section 2.4 and Alloy's multiplicity keywords (e.g. `A` **`one`** `-> B`

```
sig CD {}
abstract sig NamedEntity {
  NamedEntity_ : set CD
  name         : String -> CD
}
sig Package extends NamedEntity {
  Package_     : set CD,
  classes      : Class -> CD
}
sig Class extends NamedEntity {
  Class_       : set CD,
  attributes   : Attribute -> CD,
  general      : Class -> CD,
  persistent   : set CD
}
sig Attribute extends NamedEntity {
  Attribute_   : set CD
}

pred CD_Constraint [cd:CD] {
  (all c : Class_.cd | not (c in c.^(general.cd))) &&

  name.cd         in NamedEntity_.cd -> one String &&
  classes.cd      in Package_.cd one -> Class_.cd &&
  attributes.cd   in Class_.cd one -> Attribute_.cd &&
  general.cd      in Class_.cd -> lone Class_.cd &&
  persistent.cd   in Class_.cd

  NamedEntity_.cd = Package_.cd + Class_.cd + Attribute_.cd
}
```

Figure 9.4: Meta-model CD embedded in Alloy.

amounts to surjective and injective relations).

9.3.3 Embedding Inter-model Constraint TRCs

A typed relational constraints $\boldsymbol{T}$ representing an inter-model constraint can be embedded in a similar manner as the just presented meta-models: $\mathcal{S}_{\boldsymbol{T}}$ creates signatures, relations $D_{\boldsymbol{T}}$ are lifted to the local state idiom and formula $\phi_{\boldsymbol{T}}$ results in a predicate that receives as input the model states to be compared. In this section we follow the embedding of a QVT-R typed relational constraint into an Alloy module to provide an idea of the final specification, in particular that of the `cd2dbs` embedding presented at Figure 7.6.

Formula $\phi_{\boldsymbol{T}}$ consists of formulae $R^{\blacktriangleright}$ and $R^{\blacktriangleleft}$ for top QVT-R relations and $R_{\blacktriangleright}$ and

$R_{\blacktriangleleft}$ for QVT-R relations that are invoked from other QVT-R relations, while the variables $R_{\triangleright}$ and $R_{\triangleleft}$ introduced by $D_{\boldsymbol{T}}$ for every relation R that is called by another, give rise to the definition of new fields. Figure 9.5 presents part of the embedding of `cd2dbs` in Alloy[3]. The top-level embedding of $\texttt{C2T}^{\blacktriangleright}$ is represented by predicate `Top_C2T_DS`. Fields `P2S_DS` and `A2C_DS` called in `Top_C2T_DS` represent the relation variables $\texttt{P2S}_{\triangleright}$ and $\texttt{A2C}_{\triangleright}$, and are defined as fields in the `CD` signature. Their valuation is defined by the embedding of $\texttt{P2S}_{\blacktriangleright}$ and $\texttt{A2C}_{\blacktriangleright}$, the former embodied by predicate `Sub_P2S_DS` in Figure 9.5. The checking semantics of the whole `cd2dbs` transformation is represented by the predicate `cd2dbs` that forces all related predicates to hold.

In the presence of recursion, as in the `hsm2nhsm` example (Figure 7.7), predicates take the following shape:

```
pred Sub_S2S_NM [hm:HM,nm:NM] {
  S2S_NM[hm,nm]  = { s:State_.hm,t:State_.nm |
    all sm : SMachine_.hm, tm : SMachine_.nm |
      sm->tm in M2M_NM[hm,nm] => s in sm.states.hm =>
        t in sm.states.tm &&
        (no s.container.hm => s->t in TS2S_NM[hm,nm]
                         else s->t in SS2S_NM[hm,nm]) }
}

pred Sub_SS2S_NM [hm:HM,nm:NM] {
  SS2S_NM[hm,nm] = { s:State_.hm,t:State_.nm |
    s.container.hm->t in S2S_NM[hm,nm] }
}
```

As discussed in Section 7.2, the constraint over HSM models forcing the `constainer` association to be acyclic ensures that this recursive definition is well-behaved.

9.3.4 Embedding Metrics

Regarding enforcement semantics, Echo employs the principle of least-change by implementing the external iterative mechanism defined in Section 6.3.1. As a consequence, the distance function is also embedded in a typed relational constraint $\boldsymbol{\Delta}$ that must also be translated into Alloy.

[3]In practice, the inter-model constraint is translated as a single predicate representing $\phi_{\texttt{cd2dbs}}$; Figure 9.5 is structured to promote readability.

```
sig CD {
  ...
  P2S_DS : DS -> Package -> Schema,
  A2C_DS : DS -> Class -> Table,
  P2S_CD : DS -> Package -> Schema,
  A2C_CD : DS -> Class -> Table,
  ...
}

pred Top_C2T_DS [cd:CD,ds:DS] {
  all p:Package_.cd, s:Schema_.ds | P2S_DS[cd,ds,p,s] =>
    (all c:Class_.cd, n:String |
      n in c.name.cd && c in persistent.cd && p in c.namespace.cd =>
        (some t:Table_.ds |
          s in t.schema.ds && n in t.name.ds && c->t in A2C_DS[cd,ds]))
}

...

pred Sub_P2S_DS [cd:CD,ds:DS] {
  P2S_DS[cd,ds] = { p : Package_.cd, s: Schema_.ds | p.name.cd = s.name.ds }
}

...

pred cd2dbs [cd:CD,ds:DS]{
  Top_P2S_DS[cd,ds] && Top_P2S_CD[cd,ds] &&
  Top_C2T_DS[cd,ds] && Top_C2T_CD[cd,ds] &&
  Sub_P2S_DS[cd,ds] && Sub_P2S_CD[cd,ds] &&
  Sub_A2C_DS[cd,ds] && Sub_A2C_CD[cd,ds]
}
```

Figure 9.5: Part of the Alloy specification for `cd2dbs`.

The first proposed metric is GED. Note that an Alloy instance is isomorphic to a labelled graph whose nodes are the atoms, and edges tuples in fields (technically to hypergraphs, since fields are n-ary). With this mechanism, $\phi^{d}_{\mathbf{\Delta}^{\ominus}_{CD}}$ can be computed as `d = Delta_CD[cd,cd']`, given the definition:

```
fun Delta_CD [cd,cd':CD] : Int {
  #((Class_.cd - Class_.cd') + (Class_.cd' - Class_.cd)).plus[
  #((name.cd - name.cd') + (name.cd' - name.cd)).plus[
  ... // symmetric difference of remainder fields
  ]]
}
```

Assuming `cd'` represents an updated version of `cd`, this function sums up, for every signature and field, the size of their symmetric difference between both models. Here the state of `cd` and `cd'` embody the variables D_{CD^s} and D_{CD^t} that are introduced by $D_{\Delta^{\ominus}_{CD}}$ to represent the original model and the target model, respectively. Under the local state idiom, bounding the state of `cd` to the a concrete state amounts to assigning concrete values to the fields under `cd`; those under `cd'` shall remain free so that the model finder finds a valid assignment for them. To avoid Alloy's standard wrap around semantics for integers, model finding is executed with the option *Forbid Overflow* set (Milicevic and Jackson, 2012).

Regarding OBD, the edit operations, specified by the user in OCL using pre- and post-conditions, are automatically converted to Alloy using the translation procedure defined in (Cunha et al., 2013). Essentially, each operation will originate an Alloy predicate that specifies when it can hold between two models. The resulting Alloy predicate takes as arguments the pre- and post-states of the affected model, the receiver element of the edit operation (denoted by argument `self` of the appropriate context class), as well as the stated operation parameters. For example, the result of translating ⟦`setName`⟧ to Alloy is the following:

```
pred setName[self:Class,n:String,cd,cd':CD] {
  self.(name.cd') = n;
  all c:Class_.cd' | c.(name.cd) = c.(name.cd') or c = self
  // frame conditions inferred from modifies
  Class_.cd' = Class_.cd
  Attribute_.cd' = Attribute_.cd
  general.cd' = general.cd
  classes.cd' = classes.cd
  ...
  // remaining frame conditions
}
```

The body of the predicate consists of the translation of the pre- and post-conditions from the OCL specification. Pre-conditions (if any) are evaluated over the pre-state model `cd`, while post-conditions refer to the respective post-model `cd'`, except in the case of operations and properties marked by the tag **`@pre`** which are still evaluated in the pre-state. Frame conditions for all classes and associations not included in the **`modifies`** clause introduced in Section 9.2 are also automatically inferred by the tool.

Given the operation specifications, we constrain models to form a sequence, where

each step corresponds to the application of an edit operation:

```
open util/ordering[CD]
fact {
  all cd:CD, cd':cd.next | {
    some c:Class_.cd,n:String | setName[c,n,cd,cd'] or
    some c:Class_.cd,n:String | addAttribute[c,n,cd,cd'] or
    ...
    // remaining operation predicates
  }
}
```

The Alloy module `ordering` imposes a total order on all atoms of the given signature (in this case, `CD`), and declares a binary relation `next` that captures such order. The presented fact restricts the possible values of `next`, by requiring each state `cd` and subsequent state `cd.next` to be related by one of the specified operations. Since every `CD` atom is required to be ordered under edit operations, a solution with $d + 1$ `CD` atoms entails $\phi^{d}_{\mathbf{\Delta}^{U}_{CD}}$, while the intermediary `CD` states embody the extra variables that are declared by $D_{\mathbf{\Delta}^{U}_{CD}}$.

9.3.5 Executing the Semantics

Executing the transformation requires representing concrete models (i.e., binding the valuation of the fields regarding the original model) and setting adequate scopes. Since Alloy has no specific constructs to denote model instances, singleton signatures are used to denote specific objects and facts to fix the valuation of fields. For example, a `CD` model `CP1` denoting the company example that has been used through the dissertation can be specified as depicted in Figure 9.6.

To check whether the `CD1` model `M` is consistent with a `DBS` model `DS1` the command `check { cd2dbs[CD1,DS1] }` is issued, with the scope of each signature being set to the number of elements of the respective class in each of the two models. Regarding enforce mode with GED minimization, if `CD1` and `DS1` happened not to be consistent, in order to determine a new `CD` model `CD2` the command

```
run { cd2dbs[CD2,DS1] && Delta_CD[CD1,CD2] = d }
```

is issued with increasing d values (starting at 0). In this case, the scope of each signature is set to the number of elements of the respective class plus d, to allow complete freedom

```
one sig CD1 extends CD {}
one sig P extends Package {}
one sig C1,C2,C3 extends Class {}
one sig A1 extends Attribute {}
fact {
    Package_.CD1    = P &&
    Class_.CD1      = C1 + C2 + C3 &&
    Attribute_.CD1  = A1 &&
    name.CD1        = P->"Company" + C1->"Person" + C2->"Employee" +
                      C3->"Employer" + A1->"name" &&
    classes.CD1     = P->C1 + P->C2 + P->C3 &&
    general.CD1     = C2->C1 + C3->C1 &&
    persistent.CD1  = C2 + C3 &&
    attributes.CD1  = C1->A1
}
```

Figure 9.6: A model instance in Alloy.

in the choice of edit operations. Alloy requires the definition of the bitwidth available to represent integer atoms in a two's complement encoding. For a concrete d value, this is calculated by rounding up $1 + \log_2(d + 1)$. Since the distance function for GED only adds up the symmetric difference between relations (which is positive), if the bitwidth is sufficient to represent d, it will also be sufficient to represent the clauses of the addition. Since we are dealing with exact scopes, the class hierarchy must also be taken into consideration. For instance, for a HSM solution with one `CompositeState` and one `State`, the scope of `State` must be set to 2. This iterative process, with the calculation and increment of both d and the scope, is performed automatically by Echo, being is opaque to the user.

Regarding enforce mode with OBD minimization, the command

```
run { cd2dbs[CD2,DS] && CD1 = first && CD2 = last }
```

is issued with increasing scopes d (plus one) for signature `CD`. Singleton fields `first` and `last` denote the first and last atoms of the `next` total order: they are constrained to be the original and updated model, respectively, meaning that the latter should be obtained from the former using d edit operations. The scope of the remaining signatures is inferred from the operations specified in the meta-model, allowing a finer control over the scopes of the model finder, since the system will be aware of the behavior of all possible update steps. This requires the creation of elements by the operations to be detected, which is by itself an ambiguous issue in OCL-specified operations. For our technique, we assume that every new element created by an operation is identified

with the `oclIsNew()` operation in the post-condition and inside a **one** quantification (a predicate which holds for exactly one element (OMG, 2012, p. 170)). With `oclIsNew()` tags inside other quantifiers we would not be able to precisely measure the scope increment. For instance, consider the operation `addAttribute(n:String)` from the `Class` class. Its post-condition would contain, among others, the following constraint:

```
self.attributes->one(a | a.oclIsNew() and a.name = n)
```

This in turn would be translated to Alloy as:

```
a not in self.(attributes.cd) &&
self.(attributes.cd') = self.(attributes.cd) + a &&
a.(name.cd') = n
```

The user is required to specify an upper-bound for d that limits the search for consistent targets. If several consistent models are found at the minimum distance our tool warns the user and allows her to see the different alternatives. If the user then desires to reduce the range of produced solutions, she can, for example, add extra OCL constraints to the meta-model or narrow the set of allowed edit operations to target a specific class of solutions. Section 7.3.2 already presented a concrete example of how such narrowing can be done.

9.3.6 Optimizing Alloy Models

The major caveat of model finding approaches is scalability. While we are aware that our technique will never be as efficient as syntactic approaches (even if more expressive), in this section we present some optimizations that enable its application to many realistic examples. Although a novel contribution, the reader uninterested in Alloy technical details may skip this section as it does not affect the semantics of the proposed technique.

As presented in Section 7.2, QVT-R semantics relies heavily on nested *forall-there-exists* quantifications. These introduce inefficiency, since the complexity of the generated formulas may prevent skolemization and other optimizations performed by Kodkod (the underlying relational model finder that supports Alloy) when translating to SAT. As such, the main goal of our optimization procedure is to eliminate (or reduce the scope of) as many quantifiers as possible, sometimes taking advantage of meta-model knowledge not readily available to Kodkod.

$$(\texttt{all x:none | R}) \equiv \texttt{true} \qquad \forall\text{-EMPTY}$$
$$(\texttt{some x:none | R}) \equiv \texttt{false} \qquad \exists\text{-EMPTY}$$
$$(\texttt{all x:A | true}) \equiv \texttt{true} \qquad \forall\text{-TOP}$$
$$(\texttt{all x : A | false}) \equiv \texttt{no A} \qquad \forall\text{-BOTTOM}$$
$$(\texttt{some x : A | true}) \equiv \texttt{some A} \qquad \exists\text{-TOP}$$
$$(\texttt{some x:A | false}) \equiv \texttt{false} \qquad \exists\text{-BOTTOM}$$
$$(\texttt{all x:A | R}) \equiv \texttt{R}[\texttt{x} := \texttt{A}], \quad \text{if } \texttt{one A} \qquad \forall\text{-ONE-POINT}$$
$$(\texttt{some x:A | R}) \equiv \texttt{R}[\texttt{x} := \texttt{A}], \quad \text{if } \texttt{one A} \qquad \exists\text{-ONE-POINT}$$
$$(\texttt{all x:A | x in B => R}) \equiv (\texttt{all x:A\&B | R}) \qquad \forall\text{-TRADING}$$
$$(\texttt{some x:A | x in B \&\& R}) \equiv (\texttt{some x:A\&B | R}) \qquad \exists\text{-TRADING}$$
$$(\texttt{all x:A | x in B}) \equiv \texttt{A in B} \qquad \subseteq\text{-DEF-SET}$$
$$(\texttt{all x:A | x.R in x.S}) \equiv \texttt{A<:R in S} \qquad \subseteq\text{-DEF-REL}$$

Figure 9.7: Quantifier elimination and restriction.

Figure 9.7 presents the equivalence laws used by our system (as rewriting rules oriented from left to right) to eliminate or reduce the scope of quantifiers. Among the most effective, we have the one point rules, that require as side-condition that the set over which the quantified variable ranges is a singleton. Using knowledge about the meta-model, this condition is many times trivial to check, namely when such set is the result of a navigation expression over a mandatory attribute. Figure 9.8 presents some additional laws that are used to eliminate redundant expressions, again using knowledge about the meta-model.

To simplify the application of such rules, navigation expressions are kept normalized in the shape `x.R`, where `x` is typically a quantified variable and `R` an arbitrary composition of binary relations or their inverse. Such normalization can be done by application of associativity and inverse laws concerning the relational composition operator, such as $\texttt{R.x} \equiv \texttt{x.}{\sim}\texttt{R}$ or $(\sim(\texttt{R.S})) \equiv (\sim\texttt{S}).(\sim\texttt{R})$. Moreover, in this normalization we attempt to isolate the nearest quantified variable in a membership check using the rule $\texttt{y in x.R} \equiv \texttt{x in y.}(\sim\texttt{R})$ to potentiate the application of trading rules. Finally, whenever possible, we also replace multiplicity checks by their navigational equivalent, for example using the law $\texttt{some x.R} \equiv \texttt{x in x.R.}(\sim\texttt{R})$.

As an optimization example, consider the most simple QVT-R relation from `cd2dbs`, `P2S`, in the direction of DS. By applying QVT-R semantics the following formula

$$\texttt{R\&S} \equiv \texttt{R}, \quad \text{if } \texttt{R in S} \qquad \cap\text{-SUBSET}$$
$$\texttt{R\&S} \equiv \texttt{S}, \quad \text{if } \texttt{S in R} \qquad \cap\text{-SUBSET}$$
$$\texttt{R:>A} \equiv \texttt{R}, \quad \text{if } \texttt{univ.R in A} \qquad \rho\text{-SUBSET}$$
$$\texttt{A<:R} \equiv \texttt{R}, \quad \text{if } \texttt{R.univ in A} \qquad \delta\text{-SUBSET}$$

Figure 9.8: Redundancy elimination.

would result from our embedding in Alloy:

```
all p:Package_.cd, n:String | n in p.(name.cd) =>
  some s:Schema_.ds | n in s.(name.ds)
```

Although simple, this formula already contains 3 quantifications whose range is loosely restricted (for instance, `n` is freely quantified over all strings). Figure 9.9 shows how this formula can be simplified using the above rules. To understand how the side-conditions can be easily checked using meta-model knowledge, consider the `name` attribute in the `Package` class of the `CD` meta-model. As we have seen in Section 9.3.2, when embedding this meta-model in Alloy this attribute is encoded as a relation of type `Package -> String -> CD` (to be used always as a binary relation in the context of particular `CD` model—in our optimization example the model `cd`), constrained by the following multiplicity and inclusion dependency fact:

```
all cd:CD | name.cd in package.cd -> one String
```

From this we can deduce that `p.(name.cd) in String`, the side-condition required for the first application of rule ∩-Subset, that `one p.(name.cd)`, the side-condition to the application of ∀-One-Point, and that the domain of `name.cd` is a subset of `package.cd`, formally `(name.cd).univ in package.cd`, in the final application of δ-Subset.

This optimization procedure essentially attempts to apply the point-free transform to translate relational logic formulae to the point-free notation, briefly presented in Section 2.4. Such notation is well-know for its amenability to proof and optimization through simple equational reasoning (Oliveira, 2009), and transformation of Alloy formulas to such style has been explored before (Frias et al., 2004; Macedo, 2010), as means to perform unbounded verification proofs. In this case, its application to optimization is particularly effective, since it takes advantage of the fact that formulas originating from our embedding follow a very specific pattern, and information about the meta-model is readily available to speed-up side-condition checks.

```
all p:Package_.cd, n:String | n in p.(name.cd) =>
    some s:Schema_.ds | n in s.(name.ds)                    (∀-TRADING)
all p:Package_.cd, n:String&(p.(name.cd)) |
    some s:Schema_.ds | n in s.(name.ds)                    (∩-SUBSET)
all p:Package_.cd, n:p.(name.cd) |
    some s:Schema_.ds | n in s.(name.ds)                    (NORMALIZATION)
all p:Package_.cd, n:p.(name.cd) |
    some s:Schema_.ds | s in n.~(name.ds)                   (∃-TRADING)
all p:Package_.cd, n:p.(name.cd) |
    some s:(Schema_.ds)&(n.~(name.ds)) | true               (∃-TOP)
all p:Package_.cd, n:p.(name.cd) |
    some (Schema_.ds)&(n.~(name.ds))                        (∩-SUBSET)
all p:Package_.cd, n:p.(name.cd) |
    some n.~(name.ds)                                       (NORMALIZATION)
all p:Package_.cd, n:p.(name.cd) |
    n in n.~(name.ds).(name.ds)                             (∀-ONE-POINT)
all p:Package_.cd |
    p.(name.cd) in p.(name.cd).~(name.ds).(name.ds)         (⊆-DEF-REL)
(Package_.cd)<:(name.cd) in (name.cd).~(name.ds).(name.ds)  (δ-SUBSET)
(name.cd) in (name.cd).~(name.ds).(name.ds)
```

Figure 9.9: Optimization example.

Finally, some other optimizations, not related to quantifier elimination, are also performed during the translation of the MDE artifacts into Alloy. For example, when embedding meta-models in Alloy, fields are not created for associations marked as opposite of another existing association. Instead, when a call to an opposite association occurs in a formula (e.g. `namespace.cd`), it is just replaced to a call on its opposite using the inverse operator (e.g. `~(classes.cd)`). This further reduces the overall amount of variables and constraints during SAT solving.

Note that our tool performs these optimizations only once, when the meta-models and transformations are loaded and embedded into Alloy, and not every time the transformation is run after an update. As such, the time spent on the optimizations (which is almost negligible anyway) does not affect the performance of the least-change update propagation procedure.

9.4 Visualizing Model Instances

As just described, the user is able to choose the desired repaired model from the range of all minimal consistent solutions. Performing such choice over the concrete XMI files would not be user-friendly (even with the standard Eclipse's XMI editor), so instead we resort to the graph visualizer of the Alloy Analyzer, where perceiving models is as easy as grasping graphs. However, in order to be better understandable by the user, these graphs must be presented in a shape that resembles its model structure. The Alloy visualizer allows the definition of custom themes, and our tool automatically determines one such theme using the information available from the Ecore meta-models. Alloy's magic theme functionality (Rayside et al., 2007) also tries to infer a suitable theme from an Alloy specification through a set heuristics. However, while some of the visualization properties determined by our technique are similar to those inferred by the magic theme, the extra information available in the meta-model, and knowledge about the underlying encoding of the transformations, proves to be an advantage and eliminates the need of said heuristics.

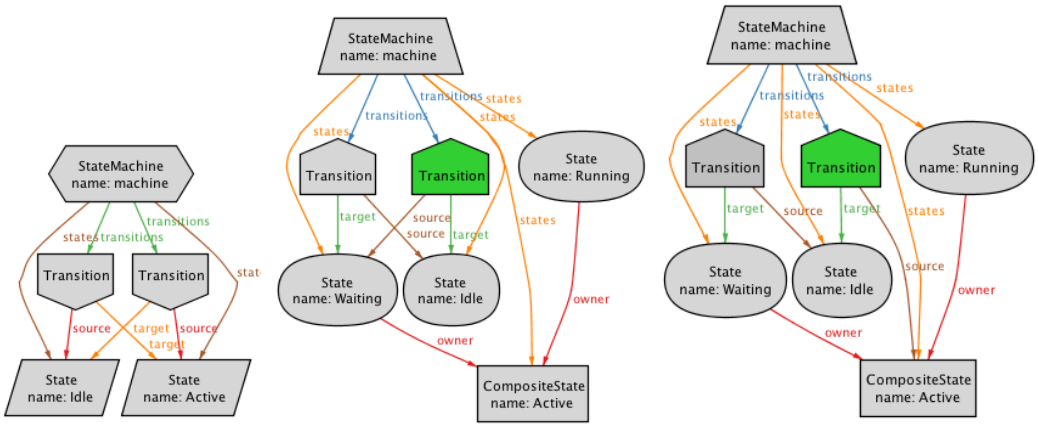
The most evident feature of the inferred theme is hiding the extra Alloy fields required by the underlying enforcing mechanism but irrelevant for the user, in particular the auxiliary fields used to represent relation calls (the $R_{\triangleright}$, $R_{\triangleleft}$, $R^{\triangleleft\triangleright}$, or $R_{\triangleleft\triangleright}$ relation variables that emerge from the skolemization). Our enforcing mechanism also requires that both the original source and target models, as well as the updated target model, coexist in a single Alloy instance (essentially, the transformation domains $\boldsymbol{M}$, $\boldsymbol{N}$ and $\boldsymbol{M}'$, for a repair on $\boldsymbol{M}$). Presenting them together to the user would be very confusing, so we opted to project the instance over the transformation domains, focusing first on the updated model, but allowing the user to visualize the others if she so desires (Alloy's magic theme would try to infer such projections (Rayside et al., 2007), but our experiments showed that it would fail to pick the desired one in this particular case). This projection allows us to omit the signature's state fields $A_{\boldsymbol{M}}$ denoting the elements belonging to each model state. To better highlight the differences between the original and the repaired models, the elements inserted or removed by the repair are painted in a different color (green), while the preserved elements are painted gray. Calculating the GED between Alloy instances already requires calculating the symmetric difference between their elements (and links). Since the Alloy visualizer allows subset signatures to be drawn differently, that component of the $\Delta^{\ominus}$ metric is reused to that end. Elements belonging to different classes are distinguished by shape.

Like with the default magic theme (Rayside et al., 2007), enumeration literals are hidden and fields whose target type is an enumeration are presented as node labels rather than as edges to the enumeration literals. However, we need not use heuristics to detect enumerations, as their existence can be detected directly in the Ecore meta-model. Following the same reasoning, Alloy fields that originated from attributes in the meta-model are also presented as node labels rather than as edges to the attribute's value node, to minimize the number of visual elements. As a consequence of the projected model states, sets end up also being represented by node labels.

Finally, we are also able to determine a suitable spanning tree for the graph, that defines its dominant hierarchical structure. In our context, these are represented by the containment associations of the meta-model, which define the overall structure of the model instances, the remaining associations depicting only references between existing elements. In the Alloy visualizer spines are defined by tagging such fields as *influencing the layout*.

Figure 9.1 shows two models: the left-one conforms to the `DBS` meta-model and is depicted with the standard Eclipse XMI editor; the right-one conforms to the `CD` meta-model and is depicted with the embedded Alloy visualizer, using the theme automatically inferred by our technique. Note how the graph is adapted to the `CD` meta-model: different classes are shaped differently, while the attribute `name` and the set `persistent` are presented as labels rather than edges. All information not relevant to the presentation of the model is hidden. The class diagram is a variation of the very simple company model that has been used throughout the dissertation. The left-hand side represents a relational schema that is `cd2dbs`-consistent with the class diagram. The `Employee` table has a `salary` column, whose matching attribute in the `CD` model is painted green (in contrast to the other elements painted gray). This means that this attribute has just been inserted by Echo in order to restore consistency between the two models.

Figure 9.10 presents models kept consistent by the `hsm2nhsm` transformation. The original `HSM` model was a simple state machine depicted at Figure 7.5a, with the `hsm2nhsm`-consistent collapsed `NHSM` model from Figure 7.5b. It is worth noting that in this example `HSM` and `NHSM` are two different meta-models, and thus the different shape assigned to elements of similarly named classes. At some point, the `NHSM` model was updated with the insertion of a transition from `Active` to `Idle`, breaking the consistency between the models. When propagating the update, Echo proposes three



(a) NHSM model. (b) Possible HSM model. (c) Possible HSM model.

Figure 9.10: hsm2nhsm-consistent models as presented in Echo.

minimal solutions. Figure 9.10 presents two of them: set the composite state `Active` as the source of the new transition or choose instead one of its sub-states, in this case, `Waiting`. In the third minimal repair (not shown) the sub-state `Running` is set as the source of the new transition.

9.5 Evaluation

At the time of writing, no benchmark for the assessment of bidirectional transformation tools has been proposed, although some efforts in that direction are beginning to be made (Anjorin et al., 2014). Thus, to assess the scalability of our technique we devised a class of synthetic examples of the familiar `cd2dbs` transformation, with the intention of achieving linear increases both in model size (number of nodes and edges when seen as a graph) and required update distance.

In this context, the shape of the CD class diagram of dimension n consists of a spine of n non-persistent classes (identified as class i at level i), each with a persistent sub-class (identified as class i' at level i), which have themselves a single attribute (with the same name i' as the owning class). Thus, a CD model of dimension n has $5n + 2$ nodes and $8n$ edges when interpreted as a graph. As an example, Figure 9.11 depicts the CD model for $n = 3$, the number of nodes being the number of model elements (10) and string literals (7), while the number of edges is the number of association links (14) and attribute assignments (13), the latter shown as node labels in Figure 9.11. The corresponding DS models, to be `cd2dbs`-consistent, must contain a table with a single

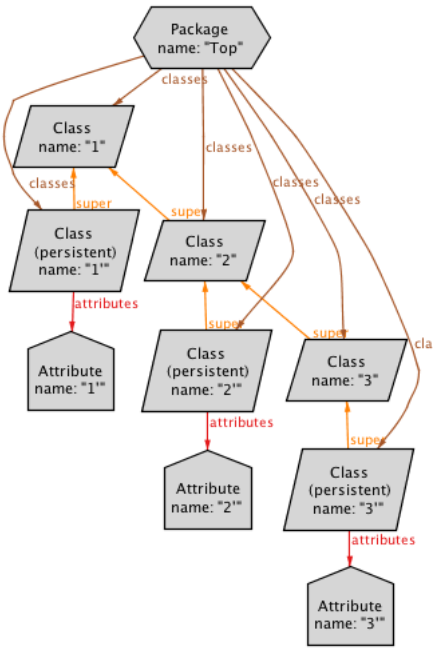


Figure 9.11: Synthetic CD model with $n = 3$.

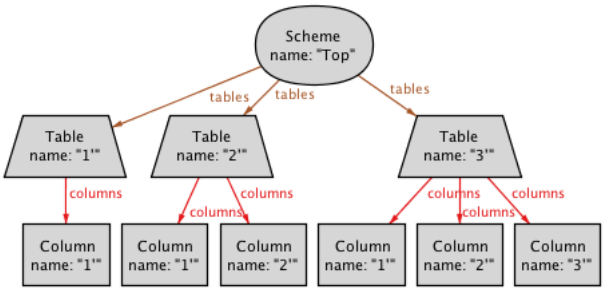


Figure 9.12: Synthetic DBS model with $n = 3$ and $d = 2$.

column for each persistent class i', that is, $3n + 2$ nodes and $4n + 1$ edges. Since the CD and DS models coexist when solving the problem, the total size of the environment is the sum of the respective sizes, with the exception of string literals which are shared. These models were generated using Echo's model generation feature, using extra OCL constraints to parametrize the shape of the generated solutions.

To introduce inconsistencies, new columns are inserted in the DS model that impose repairs on the CD model. The smallest inconsistency consists of inserting in table n' a column $(n-1)'$. To solve this inconsistency, the minimal update is to move attribute $(n-1)'$ in the CD model to the $(n-1)$ non-persistent class, so that it is shared by both class n' and class $(n-1)'$. This has a cost $\Delta = 2$ for GED and $\Delta = 1$ for OBD.

n	nodes	edges	variables
2	18	27	449
3	25	39	763
4	32	52	1063
5	39	63	1469
6	46	75	1941
7	53	87	2479
8	60	99	3083
9	67	111	3753
10	74	123	4489

Table 9.2: Scalability tests size for enforce mode with GED and $d = 1$.

Increasingly distant updates $d < n$ can be attained by inserting in every table i' such that $i > n - d$ every column j' such that $n - d \leqslant j < i$, resulting in updates $\Delta = 2d$ for GED and $\Delta = d$ for OBD in the CD model. Thus, an inconsistency at distance d introduces $\frac{d(d+1)}{2}$ nodes and $d(d + 1)$ edges. As an example, Figure 9.12 presents the DS model for $n = 3$ with inconsistencies for $d = 2$.

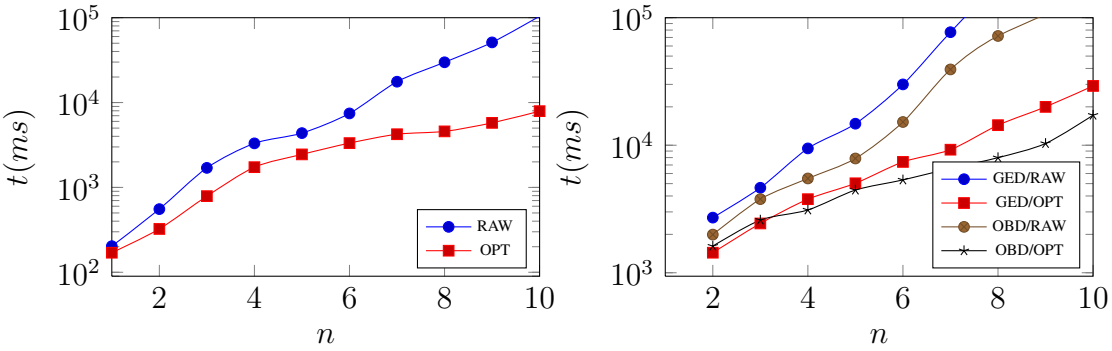
All tests were run using Echo over Alloy 4.2 with the MiniSat solver, on a 1,8 GHz Intel Core i5 with 4 GB memory running OS X 10.8. We performed experiments for models up to $n = 10$ and update distance up to $d = 3$, when applicable. Table 9.2 summarizes the total size of both models for n up to 10 given an update $d = 1$. The last column represents the number of variables present in the SAT problem generated by Kodkod when repairing the consistency between both models. All tests were run multiple times as to get the average performance values.

Figure 9.13 compares execution times (shown in log scale) of runs with and without the optimizations presented in Section 9.3.6. Figure 9.13a compares checkonly runs, and the gains are very significant. For $n = 10$ the optimized version takes only 7% of the time spent by the non-optimized version, with average gains of 45%. Checkonly runs do not require the measurement of model distances, so the choice of the metric does not affect the performance. Figure 9.13b compares enforcement runs using GED and OBD again with and without formula simplifications, for a fixed $d = 1$. The optimized versions are in average 29% and 56% more efficient than the non-optimized versions, for GED and OBD respectively, but again the difference grows fast, and for $n = 10$ the optimized versions take only 7% of the execution time of the non-optimized ones for

both GED and OBD. As already mentioned in Section 9.3.6, optimizing Alloy formulas may take some time, but since this optimization is performed only once at static-time (when the transformations are translated to Alloy), it does not affect the time effectively spent in the repair. The gain from enforcement executions using GED to those using OBD is also significant (in average the first takes 75% of the time of the second, and around 40% for $n = 10$), but these results should be analyzed with caution, as they occur in a controlled scenario where GED repairs require only twice as much solving iterations than the ones with OBD. In practice, OBD can be much faster than GED if each atomic operation is more complex, combining multiple insertions/deletions of nodes and edges, allowing inconsistencies to be repaired with smaller distances.

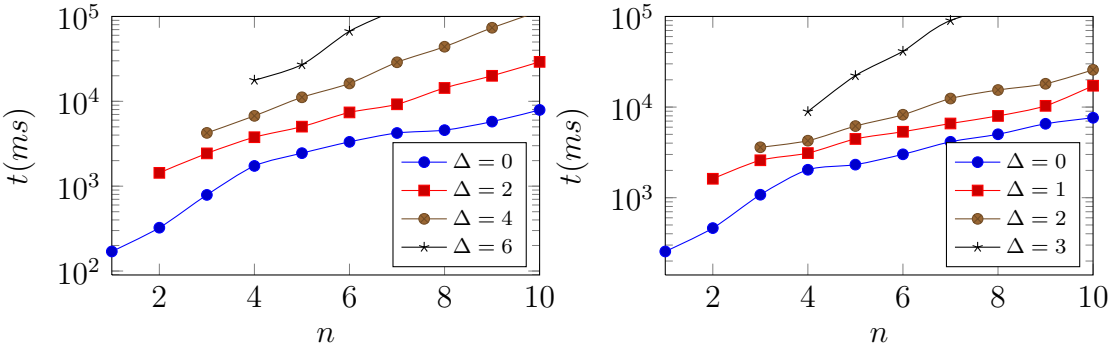
Figure 9.14 depicts the execution times of checkonly and enforcement modes (with optimizations), using both GED and OBD, respectively, as the dimension n of the model increases, and for different fixed update distances d. Execution times for $d = 0$, i.e., consistency checks, take up to $8s$ for $n = 10$. While these values are not competitive against other existing techniques for consistency checking, they are due to the lack of support for instances of Alloy: partial solutions must be encoded by additional singleton signatures and constraints in the model. The performance in this case could be significantly improved by embedding the technique directly in Kodkod or by using Alloy extensions with support for partial instances, like the one proposed by Montaghami and Rayside (2012), as our current studies show (Cunha et al., 2014). The impact on running times of the increasing d is intrinsic to our iterative technique, since every Δ step requires a new model finding run. This is better depicted in Figure 9.15 that presents the same data but in relation to increasing distance Δ, for fixed model dimensions n.

Although not ready to handle industrial-size models, Echo's greatest strength lies in its ability to allow the user to quickly and simply analyze and debug transformation specifications. In fact, a great challenge in model transformation is to guarantee that the behavior of the specified artifacts reflects the intention of the user: with a predictable least-change semantics and quick provision of feedback to the user, Echo excels in these tasks. In this context, the size of the models is not as crucial—as put by the small scope hypothesis advocated by the Alloy creators (Jackson, 2012), most problems on specifications may be flagged by small instances. This is attested by the fact that we were able to detect heretofore undetected problems in the standard's `cd2dbs` transformation, as presented in Section 7.2. Nonetheless, despite the size of the models,



(a) Checkonly. (b) Enforcement ($d = 1$) for GED and OBD.

Figure 9.13: Performance for optimized (OPT) and non-optimized (RAW) implementations.



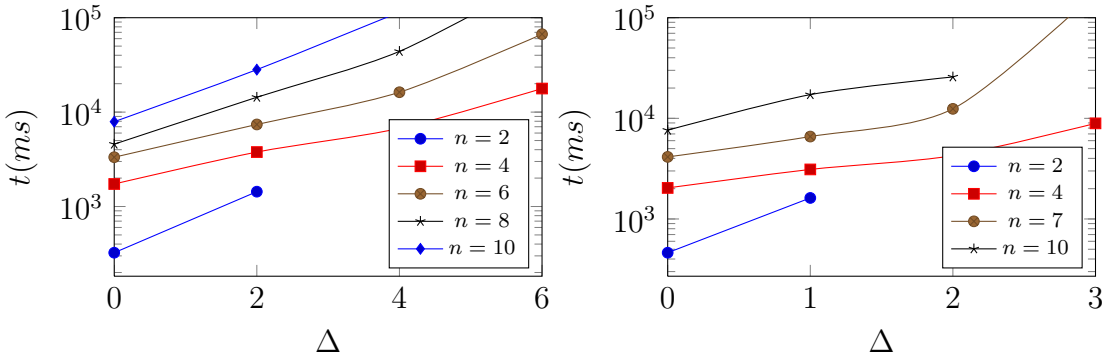
(a) GED. (b) OBD.

Figure 9.14: Performance over model size n, for fixed Δ values.

the complexity introduced by the meta-models and inter-model constraints could alone render solving unfeasible—in fact, without the optimizations presented in Section 9.3.6 that was precisely the case. In the future we intend to develop functionalities dedicated to automatically check specific properties of model transformations—like the fact that they are total, deterministic, or that they always produce well-formed models—to fully exploit this facet of Echo, as discussed in Section 7.5. As our technique is already based on model finding, these extensions are rather straight-forward to implement.

9.6 Discussion

The chapter exposed the development of an Eclipse plugin that enables the deployment of MDE tasks in a standard environment over the techniques based on model finding



(a) GED. (b) OBD.

Figure 9.15: Performance over model distance Δ, for fixed n values.

presented in Chapter 6.

Being solver-based, the main drawback of the proposed tool is performance. Improving it is the main goal of our future work: we intend to explore incremental solving techniques to speed-up the execution of successive commands with increasing scope, and to define mechanisms to infer which parts of target model can be fixed *a priori* in order to speed-up solving. In particular, we are currently analyzing the impact of embedding our technique directly in Kodkod, which has support for partial instances, and adapting it to rely on Max-SAT solvers instead, through the use of target-oriented solving techniques (Cunha et al., 2014). Consistency checking in particular can be extremely improved, since all free variables are already bounded (except those that may be introduced by the inter-model constraint). In fact, as the embedding of meta-models in Alloy hints (Figure 9.4), the need to essentially duplicate the hierarchy facts in the local state idiom undermines one of the main advantages of an Alloy embedding (its type system).

Nonetheless, even in its present status the tool is already fully functional[4], much due to the development of optimization techniques. In particular, it already proved effective in debugging existing transformations, namely helping us unveiling several errors in the well-known object-relational mapping that illustrates QVT-R specification. In the future we plan to further explore the debugging aspect of the tool by providing means to automatically verify and validate correctness properties of model transformations.

[4]Download and more information about Echo is available at `http://haslab.github.io/echo` and in the tool demo (Macedo et al., 2013a).

Chapter 10

Conclusion

This chapter summarizes the main contributions of this thesis and presents some remarks on its development. It finalizes by pointing some directions for future work.

10.1 Main Contributions

Throughout this dissertation we aimed at addressing the two limitations of existing bidirectional model transformation frameworks presented at Chapter 1: the lack of support for datatype invariants and for update minimization criteria for non-tree-like data structures. These were studied in both the asymmetric scheme of lenses and the symmetric scheme of constraint maintainers.

- We explored the viability of an invariant-constrained lens framework whose bidirectional transformations are total and well-behaved within provided data constraints (Chapter 4). The general scheme was combinatorial and relied on multi-valued backward transformation that passed on all valid solutions to the succeeding lenses, with expected performance limitations. To tame this issue, two effective constraint-aware instantiations of this scheme were developed that relied on syntactic bidirectionalization procedures: one where invariants and backward transformations are defined as general relational expressions and another in the specific domain of spreadsheet formulas under normalized and manageable invariants;
- We explored the viability of a least-change lens framework whose bidirectional transformations produce minimal updates according to parametrizable metrics

over the data domains (Chapter 5). The preservation of least-change under composition was extensively studied, with several criteria under which it is well-behaved being proposed. Two dual approaches were explored: one where the backward transformations select single minimal values and another where all minimal values are returned, each better suited under different metrics;

- We proposed the formalization of invariant-constrained least-change constraint maintainers on top of relational model finding procedures (Chapter 6). Rather than syntactically deriving consistency-restoring transformations, at the core of this approach are embeddings of the meta-models and model transformations into intra- and inter-model relational constraints that are to be solved by off-the-shelf model finders. Mechanisms were explored to implement least-change behavior on top of such relational model finders. The nature of this approach renders it prone to be generalized into other MDE tasks, and we have done so for model repair, model synchronization and multidirectional model transformation;

- We provided the embedding of two concrete model transformation languages into relational constraints, so that they could be managed by the above mentioned constraint maintainer formalization: QVT-R, a multidirectional declarative language (Chapter 7) and ATL, a unidirectional language (Chapter 8). The former allowed us to address some problems and ambiguities known to affect the bidirectional semantics of QVT-R, while the latter allowed us to reason about the bidirectionalization of inherently unidirectional model transformation languages. Since the technique is generalizable to multidirectional transformations, we explored the potential of multidirectional QVT-R transformations, which have heretofore been widely disregarded. We conclude that the language would need to be extended if it is to support realistic multidirectional scenarios;

- We have deployed the constraint maintainer formalization as an Eclipse plugin, using Alloy and the underlying relational model finder (Chapter 9). The implementation aimed at providing a seamless integration with a standard MDE environment. Being an Eclipse plugin, Echo is developed on top of the popular EMF, with meta-models being specified in Ecore with embedded OCL constraints and inter-model constraints as QVT-R or ATL transformations. Thus, Echo fully supports the execution of correct and least-change QVT-R bidirectional transformations. Being solver-based scalability is its main drawback, and thus

techniques to improve performance were explored.

10.2 Final Remarks

Relational logic As the unifying formalism underlying the frameworks proposed in this dissertation, relational logic has proven to be sufficiently expressive and flexible to address the issues we initially set out to study. In the proposed lens frameworks, this allowed us to support partial and multi-valued backward transformations, which proved essential to achieve well-behaved combinatorial techniques; in the constraint maintainer framework this allowed us to naturally support non-bijective consistency relations. The timely development of the toolchain of relational model finders Alloy/Kodkod also proved to be fundamental, providing means to quickly validate the properties of the lens frameworks and ending up being used as the base over which the constraint maintainer framework was effectively deployed. Furthermore, since relations are first-class citizens in this formalism, they provide a natural way to represent graph-like data structures (like models).

Multi-valued transformations Backed up by the expressive power of relational logic, multi-valued transformations were at the core of the proposed lens and constraint maintainer frameworks. While in the constraint maintainer setting this was a simple consequence of the adoption of model finders under the assumption that may be multiple valid solutions, in the lens frameworks multi-valued transformations were required to preserve the well-behavedness in purely combinatorial approaches. As should be expected, this imposes a great toll on the efficiency of the framework. The constraint-aware instantiations of invariant-constrained lenses somehow tamed this issue by reducing multi-valuedness to a minimum while forfeiting the combinatorial approach; due to the complexity of the task and the embryonic stage of our studies on least-change lenses, we fell short of defining similar techniques for least-change lenses.

QVT-R language While the QVT-R language is the most well-known model transformation language with clear bidirectional concerns, reasoning about the bidirectional behavior of QVT-R transformations has proven to be an arduous task. This is probably reflected in the slow adoption of this standard by the MDE community and in the lack of tool support, contrasting with the other standards proposed by the MDA

initiative. Although we do believe that some of the characteristics of QVT-R are attractive—the specification of a bidirectional transformation through declarative consistency relations—there is still room for improvement. The great limitations that arose when the multidirectional scenario was addressed, and even the lack of support for simple subset consistency relations in the bidirectional scenario, make this claim even more evident.

10.3 Future Work

Model metrics The distance functions have proved to be essential on controlling the behavior of least-change bidirectional transformations. However, at certain points, they have also proven not to be sufficiently flexible to address interesting examples. This was more evident in the least-change lens framework, where the compositionality of simple examples failed for typical distance functions. To assess the feasibility of said framework, a more in-depth study of model metrics is in order, e.g. using distance functions that entail lexicographic orders. Likewise in the constraint maintainer setting. While we believe that providing both a meta-model independent and automatically inferred metric (GED) and another one parametrizable by the user (OBD) results in an interesting combination, there have been scenarios where we could benefit from more expressive metrics, namely by assigning varied weights to the insertion/removal of particular nodes/edges (GED) or edit operations (OBD). We are currently exploring other ways through which target-oriented model finders can be used to control the generation of solutions (Macedo et al., 2014a).

Performance While the solver-based approach to constraint maintainers imposes a clear threshold on the efficiency of the technique, there are a number of techniques that can be explored to improve the current performance of Echo. Concretely, we intend to explore incremental solving techniques to speed-up the execution of successive commands with increasing scope, and to define mechanisms to infer which parts of target model can be fixed *a priori* in order to speed-up solving. In particular, we are currently analyzing the impact of embedding our technique directly in Kodkod, which has support for partial instances, and adapting it to rely on Max-SAT solvers instead, through the use of target-oriented solving techniques (Cunha et al., 2014). The connection between bidirectional transformation and other MDE tasks outlined at

Chapter 6 could also allow the combination of some successful model repair techniques with our solver-based approach, like the incremental model repair technique proposed by Reder and Egyed (2012) and that has shown to be extremely efficient.

Transformation validation Embodying the intentions of the user in the specification of a bidirectional model transformations is still a challenge, has been made evident by the problems detected on the bidirectional semantics of QVT-R. One of the main advantages of Echo is its ability to quickly provide useful feedback to the user and act as a debugger for the specified transformations. In the future we plan to further explore this debugging aspect of the tool by providing means to automatically verify and validate correctness properties of model transformations. While some techniques for the validation of QVT-R (Garcia, 2008; Cabot et al., 2012) and ATL (Büttner et al., 2012) transformations have been proposed, our embedding of the transformations into relational constraints and subsequent deployment over model finders renders such tasks rather straight-forward to implement and extend.

Appendix A

Relation Algebra Laws

Inclusion $\subseteq$

$$R = S \equiv R \subseteq S \wedge S \subseteq R \qquad =\text{-DEF}$$

$$f \subseteq g \equiv f = g \equiv g \subseteq f \qquad \subseteq\text{-FUNCTIONAL}$$

$$R \subseteq S \wedge S \subseteq T \Rightarrow R \subseteq T \qquad \subseteq\text{-TRANSITIVITY}$$

Composition $\circ$

$$R \subseteq S \wedge T \subseteq U \Rightarrow R \circ T \subseteq S \circ U \qquad \circ\text{-MONOTONICITY}$$

$$R \subseteq S \Rightarrow R \circ T \subseteq S \circ T$$

$$R \subseteq S \Rightarrow T \circ R \subseteq T \circ S$$

$$f \circ R \subseteq S \equiv R \subseteq f^\circ \circ S \qquad \text{SHUNTING}$$

$$R \circ f^\circ \subseteq S \equiv R \subseteq S \circ f$$

$$S \circ R \subseteq T \equiv \delta S \circ R \subseteq S^\circ \circ T \Leftarrow \mathsf{img}\ S \subseteq \mathsf{id} \qquad \text{SHUNTING (SIMPLE)}$$

$$R \circ S^\circ \subseteq T \equiv R \circ \delta S \subseteq T \circ S \Leftarrow \mathsf{img}\ S \subseteq \mathsf{id}$$

Meet $\cap$

$$X \subseteq (R \cap S) \equiv (X \subseteq R) \wedge (X \subseteq S) \qquad \cap\text{-UNIVERSAL}$$

$$R \subseteq S \Rightarrow R \cap T \subseteq S \cap T \qquad \cap\text{-MONOTONICITY}$$

$$R \subseteq S \Rightarrow R \cap T \subseteq S \qquad \cap\text{-SMALLER}$$

$$R \cap \overline{R} = \bot \qquad \cap\text{-COMPLEMENT}$$

$$(R \cap S) \circ T = (R \circ T) \cap (S \circ T) \Leftarrow R \circ \mathsf{img}\ T \subseteq R \vee S \circ \mathsf{img}\ T \subseteq S$$
$$(R \cap S) \circ f = (R \circ f) \cap (S \circ f) \qquad \cap\text{-DISTRIBUTIVITY}$$
$$T \circ (R \cap S) = (T \circ R) \cap (T \circ S) \Leftarrow \mathsf{ker}\ T \circ R \subseteq R \vee \mathsf{ker}\ T \circ S \subseteq S$$

Join $\cup$

$$R \cup S \subseteq T \equiv R \subseteq T \wedge S \subseteq T \qquad \cup\text{-UNIVERSAL}$$

$$(R \cup S) \circ T = (R \circ T) \cup (S \circ T) \qquad \cup\text{-DISTRIBUTIVITY}$$
$$T \circ (R \cup S) = (T \circ R) \cup (T \circ S)$$

$$R \subseteq S \Rightarrow R \subseteq S \cup T \qquad \cup\text{-LARGER}$$

$$R \cup \overline{R} = \top \qquad \cup\text{-COMPLEMENT}$$

$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T) \qquad \cap/\cup\text{-DISTRIBUTIVITY}$$
$$R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$$

Inverse $^\circ$

$$R \subseteq S \Rightarrow R^\circ \subseteq S^\circ \qquad {}^\circ\text{-MONOTONICITY}$$

$$(R^\circ)^\circ = R \qquad {}^\circ\text{-INVOLUTION}$$
$$(R \circ S)^\circ = S^\circ \circ R^\circ \qquad {}^\circ\text{-COMPOSITION}$$

$$(R \cap S)^\circ = R^\circ \cap S^\circ \qquad \circ\text{-MEET}$$

$$(R \cup S)^\circ = R^\circ \cup S^\circ \qquad \circ\text{-JOIN}$$

Domain δ | Range ρ

$$\mathsf{ker}\ R = R^\circ \circ R \qquad \mathsf{ker}\text{-DEF}$$

$$\mathsf{img}\ R = R \circ R^\circ \qquad \mathsf{img}\text{-DEF}$$

$$\delta R = (R^\circ \circ R) \cap \mathsf{id} \qquad \delta\text{-DEF}$$

$$\rho R = (R \circ R^\circ) \cap \mathsf{id} \qquad \rho\text{-DEF}$$

$$\delta(R \circ S) = \delta(\delta R \circ S) \qquad \delta\text{-COMPOSITION}$$

$$\rho(R \circ S) = \rho(R \circ \rho S) \qquad \rho\text{-COMPOSITION}$$

$$\delta(R^\circ) = \rho R \qquad \delta/\rho\text{-INVERSE}$$

$$\rho(R^\circ) = \delta R$$

$$\delta(R \cup S) = \delta R \cup \delta S \qquad \delta/\rho\text{-JOIN}$$

$$\rho(R \cup S) = \rho R \cup \rho S$$

$$\delta(R \cap S) = (R^\circ \circ S) \cap \mathsf{id} \qquad \delta/\rho\text{-MEET}$$

$$\rho(R \cap S) = (R \circ S^\circ) \cap \mathsf{id}$$

$$R \subseteq S \Rightarrow \delta R \subseteq \delta S \qquad \delta\text{-MONOTONICITY}$$

$$R \subseteq S \Rightarrow \rho R \subseteq \rho S \qquad \rho\text{-MONOTONICITY}$$

$$\rho R \circ R = R \qquad \rho\text{-NEUTRAL}$$

$$R \circ \delta R = R \qquad \delta\text{-NEUTRAL}$$

Coreflexives Φ

$$\Phi \subseteq \mathsf{id} \qquad \Phi\text{-DEF}$$

$$R \subseteq S \circ \Phi \circ T \Rightarrow R \subseteq S \circ T \qquad \Phi\text{-LARGER}$$

$$R \circ S \subseteq T \Rightarrow R \circ \Phi \circ S \subseteq T \qquad \Phi\text{-SMALLER}$$

$$\Phi \circ \Psi = \Phi \cap \Psi \qquad \Phi\text{-COMPOSITION}$$

$$\Phi \circ \Phi = \Phi \qquad \Phi\text{-REFLEXIVITY}$$

$$\Phi^\circ = \Phi \qquad \Phi\text{-INVERSE}$$

$$\rho\Phi = \Phi = \delta\Phi \qquad \Phi\text{-DOMAIN/RANGE}$$

$$R \circ \Phi \subseteq S \equiv R \circ \Phi \subseteq S \circ \Phi \qquad \Phi\text{-SHUNTING}$$

$$\Phi \circ R \subseteq S \equiv \Phi \circ R \subseteq \Phi \circ S$$

Empty $\bot$ | Universal $\top$

$$\bot \subseteq R \qquad \bot\text{-DEF}$$

$$R \circ \bot = \bot = \bot \circ R \qquad \bot\text{-COMPOSITION}$$

$$R \cup \bot = R \qquad \bot\text{-NEUTRAL}$$

$$R \cap \bot = \bot \qquad \bot\text{-ABSORPTION}$$

$$R \subseteq \top \qquad \top\text{-DEF}$$

$$R \cup \top = \top \qquad \top\text{-ABSORPTION}$$

$$R \cap \top = R \qquad \top\text{-NEUTRAL}$$

$$\top \circ \delta R = \top \circ R \qquad \delta\text{-ELIMINATION}$$

$$\rho R \circ \top = R \circ \top \qquad \rho\text{-ELIMINATION}$$

Product $\triangle$

$$X \subseteq R \triangle S \equiv \pi_1 \circ X \subseteq R \wedge \pi_2 \circ X \subseteq S \qquad \triangle\text{-UNIVERSAL}$$

$$R \triangle S = (\pi_1^\circ \circ R) \cap (\pi_2^\circ \circ S) \qquad \triangle\text{-DEF}$$

$$\pi_1 \circ (R \triangle S) = R \circ \delta S \wedge \pi_2 \circ (R \triangle S) = S \circ \delta R \qquad \triangle\text{-CANCELLATION}$$

$$\pi_1 \circ (f \triangle g) = f \wedge \pi_2 \circ (f \triangle g) = g$$

$$\pi_1 \triangle \pi_2 = \mathsf{id} \qquad \triangle\text{-REFLECTION}$$

$(R \vartriangle S) \circ T = (R \circ T) \vartriangle (S \circ T) \Rightarrow R \circ \mathsf{img}\ T \subseteq R \vee S \circ \mathsf{img}\ T \subseteq S$ $\vartriangle$-FUSION

$(R \vartriangle S) \circ f = (R \circ f) \vartriangle (S \circ f)$

$(R \vartriangle S) \circ \Phi = (R \circ \Phi) \vartriangle (S \circ \Phi)$

$(R \vartriangle S)^\circ\ (T \vartriangle U) = (R^\circ \circ T) \cap (S^\circ \circ U)$ $\vartriangle$-INVERSE

$\delta\pi_1 = \mathsf{id} = \delta\pi_2$ π-DOMAIN

$\delta(R \vartriangle S) = \delta R \cap \delta S$ $\vartriangle$-DOMAIN

$\rho\pi_1 = \mathsf{id} = \rho\pi_2$ π-RANGE

$\rho(R \vartriangle S) = (\pi_1^\circ \circ R \circ S^\circ \circ \pi_2) \cap \mathsf{id}$ $\vartriangle$-RANGE

$R \vartriangle S = \bot \Leftarrow R = \bot \vee S = \bot$ $\vartriangle$-EMPTY

Sum $\triangledown$

$X = R \triangledown S \equiv X \circ i_1 = R \wedge X \circ i_2 = S$ $\triangledown$-UNIVERSAL

$R \triangledown S = (R \circ {i_1}^\circ) \cup (S \circ {i_2}^\circ)$ $\triangledown$-DEF

$(R \triangledown S) \circ i_1 = R \wedge (R \triangledown S) \circ i_1 = S$ $\triangledown$-CANCELLATION

$i_1 \triangledown i_2 = \mathsf{id}$ $\triangledown$-REFLECTION

$T \circ (R \triangledown S) = (T \circ R) \triangledown (T \circ S)$ $\triangledown$-FUSION

$(R \triangledown S) \circ (T \triangledown U)^\circ = (R \circ T^\circ) \cup (S \circ U^\circ)$ $\triangledown$-INVERSE

$\delta i_1 = \mathsf{id} = \delta i_2$ i-DOMAIN

$\rho i_1 = i_1 \triangledown \bot$ i-RANGE

$$\rho i_2 = \bot \triangledown i_2$$

$$\delta(R \vartriangle S) = (i_1 \circ \delta R) \triangledown (i_2 \circ \delta S) \qquad \triangledown\text{-DOMAIN}$$

$$\rho(R \vartriangle S) = \rho R \cup \rho S \qquad \triangledown\text{-RANGE}$$

$$R \triangledown S = \bot \equiv R = \bot \wedge S = \bot \qquad \triangledown\text{-EMPTY}$$

Division $/$

$$R \circ S \subseteq T \equiv R \subseteq T \,/\, S \qquad /\text{-DEF}$$

$$(R \cap S) \,/\, T = (R \,/\, T) \cap (S \,/\, T) \qquad /\text{-DISTRIBUTIVITY}$$

$$R \,/\, (S \cup T) = (R \,/\, S) \cap (R \,/\, T)$$

$$\overline{R \,/\, S} = \overline{R} \circ S^{\circ} \qquad /\text{-COMPLEMENT}$$

$$R \,/\, \mathsf{id} = R \qquad /\text{-NEUTRAL}$$

$$R \,/\, \bot = \top \qquad /\text{-ABSORPTION}$$

$$\top \,/\, R = \top$$

$$(S \,/\, R) \circ R \subseteq S \qquad /\text{-CANCELLATION}$$

$$S \subseteq (S \circ R) \,/\, R$$

$$(R \,/\, S) \,/\, T = R \,/\, (T \,/\, S) \qquad /\text{-ASSOCIATIVITY}$$

$$(R \,/\, S) \circ f = R \,/\, (f^{\circ} \circ S) \qquad /\text{-FUSION}$$

$$f^{\circ} \circ (R \,/\, S) = (f^{\circ} \circ R) \,/\, S$$

Bibliography

R. Abraham and M. Erwig. GoalDebug: A spreadsheet debugger for end users. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 251–260. IEEE, 2007.

S. M. Abramov and R. Glück. The universal resolving algorithm: Inverse computation in a functional language. In *Proceedings of the 5th International Conference on Mathematics of Program Construction (MPC 2000)*, volume 1837 of *LNCS*, pages 187–212. Springer, 2000.

Y. Adachi. Intellisheet: a spreadsheet system expanded by including constraint. In *Proceedings of the 2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001)*, pages 173–179. IEEE, 2001.

K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9:69–86, 2010.

A. Anjorin, A. Cunha, H. Giese, F. Hermann, A. Rensink, and A. Schürr. Benchmarx. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014)*, volume 1133 of *CEUR Workshop Proceedings*, pages 82–86. CEUR-WS, 2014.

ATLAS group. ATL user guide. http://wiki.eclipse.org/ATL/User_Guide.

J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.

R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering (ICSE 1991)*, pages 158–165. IEEE / ACM, 1991.

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP 2010)*, pages 193–204. ACM, 2010.

R. S. Bird and O. de Moor. *Algebra of programming*. Prentice Hall International series in computer science. Prentice Hall, 1997. ISBN 978-0-13-507245-5.

A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2006)*, pages 338–347. ACM, 2006.

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 407–419. ACM, 2008.

A. Boronat, J. A. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE 2006)*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.

C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. In *Proceedings of the 10th International Symposium on Database Programming Languages (DBPL 2005)*, pages 27–41, 2005.

J. Bradfield and P. Stevens. Recursive checkonly QVT-R transformations with general when and where clauses via the modal mu calculus. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*, volume 7212 of *LNCS*, pages 194–208. Springer, 2012.

J. Bradfield and P. Stevens. Enforcing QVT-R with mu-calculus and games. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*, volume 7793 of *LNCS*, pages 282–296. Springer, 2013.

P. Buneman, S. Khanna, and W. C. Tan. On propagation of deletions and annotations through views. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2002)*, pages 150–158. ACM, 2002.

F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL transformations using transformation models and model finders. In *Proceedings of the 14th International Conference on Formal Engineering Methods (ICFEM 2012)*, volume 7635 of *LNCS*, pages 198–213. Springer, 2012.

J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2012.

I. Cervesato. *The Deductive Spreadsheet*. Cognitive Technologies. Springer, 2013. ISBN 978-3-642-37746-4.

A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. JTL: A bidirectional and change propagating transformation language. In *Proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *LNCS*, pages 183–202. Springer, 2010.

E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

A. Cunha and J. Visser. Transformation of structure-shy programs with application to XPath queries and strategic functions. *Science of Computer Programming*, 76(6): 516–539, 2011.

A. Cunha, A. Garis, and D. Riesco. Translating between Alloy specifications and UML class diagrams annotated with OCL. *Software and System Modeling*, 2013.

A. Cunha, N. Macedo, and T. Guimarães. Target oriented relational model finding. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014)*, volume 8411 of *LNCS*, pages 17–31. Springer, 2014.

J. Cunha, J. P. Fernandes, J. Mendes, H. Pacheco, and J. Saraiva. Bidirectional transformation of model-driven spreadsheets. In *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT 2012)*, volume 7307 of *LNCS*, pages 105–120. Springer, 2012.

K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the 2nd*

International Conference on Theory and Practice of Model Transformations (ICMT 2009), volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

J. de Lara and E. Guerra. Formal support for QVT-Relations with coloured Petri nets. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009)*, volume 5795 of *LNCS*, pages 256–270. Springer, 2009.

Z. Diskin. Algebraic models for bidirectional model synchronization. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *LNCS*, pages 21–36. Springer, 2008.

Z. Diskin. Model synchronization: Mappings, tiles, and categories. In *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2011)*, volume 6491 of *LNCS*, pages 92–165. Springer, 2011.

Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *Journal of Object Technology*, 10:6: 1–25, 2011.

J. Edwards, D. Jackson, and E. Torlak. A type system for object models. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2004)*, pages 189–199. ACM, 2004.

H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *LNCS*, pages 72–86. Springer, 2007.

J. N. Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, December 2009.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), 2007.

J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional programming (ICFP 2008)*, pages 383–396. ACM, 2008.

P. J. Freyd and A. Scedrov. *Categories, allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990. ISBN 978-0-444-70368-2.

M. F. Frias. *Fork Algebras in Algebra, Logic and Computer Science*, volume 2 of *Advances in Logic*. World Scientific, 2002. ISBN 978-981-02-4876-5.

M. F. Frias, C. L. Pombo, and N. Aguirre. An equational calculus for Alloy. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *LNCS*, pages 162–175. Springer, 2004.

D. Fylstra, L. Lasdon, J. Watson, and A. Waren. Design and use of the Microsoft Excel Solver. *Interfaces*, 28(5):29–55, 1998.

M. Garcia. Formalization of QVT-Relations: OCL-based static semantics and Alloy-based validation. In *Proceedings of the Second Workshop on MDSD Today*, pages 21–30. Shaker Verlag, 2008.

A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In *Proceedings of the 1st International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 90–105. Springer, 2002.

H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and System Modeling*, 8(1):21–43, 2009.

G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

J. Greenyer and E. Kindler. Comparing relational model transformation technologies: implementing Query/View/Transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.

J. Greenyer, S. Pook, and J. Rieke. Preventing information loss in incremental model synchronization by reusing elements. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011)*, volume 6698 of *LNCS*, pages 144–159. Springer, 2011.

E. Guerra and J. de Lara. An algebraic semantics for QVT-Relations check-only transformations. *Fundamenta Informaticae*, 114(1):73–101, 2012.

Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró. Quick fix generation for DSMLs. In *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011)*, pages 17–24. IEEE, 2011.

S. J. Hegner. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence*, 40:63–125, 2004.

F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, and Y. Xiong. Correctness of model synchronization based on triple graph grammars. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2011)*, volume 6981 of *LNCS*, pages 668–682. Springer, 2011.

F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong, S. Gottmann, and T. Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software and System Modeling*, pages 1–29, 2013.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 205–216. ACM, 2010.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 480–483. IEEE, 2011.

M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 371–384. ACM, 2011.

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1–2):89–118, 2008.

Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger. Dagstuhl seminar on bidirectional transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011.

Z. Huzar, L. Kuzniarz, G. Reggio, and J.-L. Sourrouille. Consistency problems in UML-based software development. In *Proceedings of the UML Modeling Languages and Applications 2004 Satellite Activities*, volume 3297 of *LNCS*, pages 1–12. Springer, 2004.

ikv++ technologies ag. Medini QVT. http://projects.ikv.de/qvt/.

D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012. ISBN 978-0-262-01715-2.

M. Johnson, R. Rosebrugh, and R. Wood. Algebras and update strategies. *Journal of Universal Computer Science*, 16(5):729–748, 2010.

F. Jouault and I. Kurtev. Transforming models with ATL. In *Proceedings of the Satellite Events at the MoDELS 2005 Conference (MoDELS 2005)*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.

F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006)*, pages 1188–1195. ACM, 2006.

F. Jouault and M. Tisi. Towards incremental execution of ATL transformations. In *Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT 2010)*, volume 6142 of *LNCS*, pages 123–137. Springer, 2010.

F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.

M. Kassoff, L.-M. Zen, A. Garg, and M. R. Genesereth. PrediCalc: a logical spreadsheet management system. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 1247–1250. ACM, 2005.

S. Kawanaka and H. Hosoya. biXid: a bidirectional transformation language for XML. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*, pages 201–214. ACM, 2006.

A. Keller. Choosing a view update translator by dialog at view definition time. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB 1986)*, pages 467–474. Morgan Kaufmann publishers, 1986.

A. Keller and J. D. Ullman. On complementary and independent mappings on databases. In *Proceedings of the Annual SIGMOD Meeting (SIGMOD 1984)*, pages 143–148. ACM, 1984.

M. Kleiner, M. D. D. Fabro, and P. Albert. Model search: Formalizing and automating constraint solving in MDE platforms. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, volume 6138 of *LNCS*, pages 173–188. Springer, 2010.

M. Konopasek and S. Jayaraman. *The TK! Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education*. Osborne/McGraw-Hill, 1984.

M. Kuhlmann and M. Gogolla. From UML and OCL to relational logic and back. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2012)*, volume 7590 of *LNCS*, pages 415–431. Springer, 2012.

J. A. Larson and A. P. Sheth. Updating relational views using knowledge at view definition and view update time. *Information Systems*, 16(2):145–168, 1991.

G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of XQuery. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2007)*, pages 21–30. ACM, 2007.

N. Macedo. Translating Alloy specifications to the point-free style. Master's thesis, Universidade do Minho, 2010.

N. Macedo and A. Cunha. Automatic unbounded verification of Alloy specifications with Prover9. *CoRR*, abs/1209.5773, 2012.

N. Macedo and A. Cunha. Implementing QVT-R bidirectional model transformations using Alloy. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*, volume 7793 of *LNCS*, pages 297–311. Springer, 2013.

N. Macedo and A. Cunha. Least-change bidirectional model transformation with QVT-R and ATL. *Software and System Modeling*, 2014.

N. Macedo, H. Pacheco, and A. Cunha. Relations as executable specifications: taming partiality and non-determinism using invariants. In *Proceedings of the 13th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2012)*, volume 7560 of *LNCS*, pages 146–161. Springer, 2012.

N. Macedo, T. Guimarães, and A. Cunha. Model repair and transformation with Echo. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, pages 694–697. IEEE, 2013a.

N. Macedo, H. Pacheco, A. Cunha, and J. N. Oliveira. Composing least-change lenses. *Electronic Communications of the EASST*, 57, 2013b.

N. Macedo, A. Cunha, and T. Guimarães. Exploring scenario exploration. Submitted, 2014a.

N. Macedo, A. Cunha, and H. Pacheco. Towards a framework for multi-directional model transformations. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014)*, volume 1133 of *CEUR Workshop Proceedings*, pages 71–74. CEUR-WS, 2014b.

N. Macedo, H. Pacheco, A. Cunha, and N. R. Sousa. Bidirectional spreadsheet formulas. In *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2014)*. IEEE, 2014c.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 47–58. ACM, 2007.

L. Meertens. Designing constraint maintainers for user interaction. Manuscript available at `http://www.kestrel.edu/home/people/meertens`, 1998.

T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

A. Milicevic and D. Jackson. Preventing arithmetic overflows in Alloy. In *Proceedings of the 3rd International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *LNCS*, pages 108–121. Springer, 2012.

V. Montaghami and D. Rayside. Extending Alloy with partial instances. In *ABZ*, volume 7316 of *LNCS*, pages 122–135. Springer, 2012.

A. D. Morgan. On the syllogism, part III (1958). In P. L. Heath, editor, *On the syllogism: and other logical writings*, Rare masterpieces of philosophy and science. Routledge, 1966.

S.-C. Mu and J. N. Oliveira. Programming from Galois connections. In *Proceedings of the 12th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2011)*, volume 6663 of *LNCS*, pages 294–313. Springer, 2011.

S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 2–20. Springer, 2004.

J. P. Near and D. Jackson. An imperative extension to Alloy. In *Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977 of *LNCS*, pages 118–131. Springer, 2010.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2012. ISBN 978-3-540-43376-7.

J. N. Oliveira. Transforming data by calculation. In *Proceedings of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, pages 134–195, 2007.

J. N. Oliveira. Extended static checking by calculation using the pointfree transform. In *Revised Tutorial Lectures of the International LerNet ALFA Summer School*, volume 5520 of *LNCS*, pages 195–251. Springer, 2009.

OMG. *MOF 2.0 Query/View/Transformation Specification (QVT), Version 1.1*, January 2011a. Available at http://www.omg.org/spec/QVT/1.1/.

OMG. *OMG Unified Modeling Language (UML), Version 2.4.1*, August 2011b. Available at http://www.omg.org/spec/UML/2.4.1/.

OMG. *OMG Object Constraint Language (OCL), Version 2.3.1*, January 2012. Available at http://www.omg.org/spec/OCL/2.3.1/.

OMG. *OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1*, June 2013. Available at http://www.omg.org/spec/MOF/2.4.1/.

H. Pacheco. *Bidirectional Data Transformation by Calculation*. PhD thesis, Universidade do Minho, 2012.

H. Pacheco and A. Cunha. Generic point-free lenses. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC 2010)*, volume 6120 of *LNCS*, pages 331–352. Springer, 2010.

H. Pacheco and A. Cunha. Calculating with lenses: optimising bidirectional transformations. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011)*, pages 91–100. ACM, 2011.

H. Pacheco, A. Cunha, and Z. Hu. Delta lenses over inductive types. *Electronic Communications of the EASST*, 49, 2012.

H. Pacheco, N. Macedo, A. Cunha, and J. Voigtländer. A generic scheme and properties of bidirectional transformations. *CoRR*, abs/1306.4473, 2013.

H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In *Proceedings of the 22th ACM SIGPLAN Workshop on Partial evaluation and Program Manipulation (PEPM 2014)*, pages 39–50. ACM, 2014.

B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

J. Puissant, R. Straeten, and T. Mens. Resolving model inconsistencies using automated regression planning. *Software and System Modeling*, pages 1–21, 2013.

D. Rayside, F. S.-H. Chang, G. Dennis, R. Seater, and D. Jackson. Automatic visualization of relational logic models. *Electronic Communications of the EASST*, 7, 2007.

A. Reder and A. Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 220–229. ACM, 2012.

I. Sasano, Z. Hu, S. Hidaka, K. Inaba, H. Kato, and K. Nakano. Toward bidirectionalization of ATL with GRoundTram. In *Proceedings of the 4th International Conference*

on Theory and Practice of Model Transformations (ICMT 2011), volume 6707 of *LNCS*, pages 138–151. Springer, 2011.

D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

A. Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994)*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

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

S. Sendall and J. Küster. Taming model round-trip engineering. In *Proceedings of the Workshop on Best Practices for Model-Driven Software Development*, 2004.

P. Stevens. A landscape of bidirectional model transformations. In *Proceedings of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *LNCS*, pages 408–424. Springer, 2007.

P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.

P. Stevens. A simple game-theoretic approach to checkonly QVT relations. *Software and System Modeling*, 12(1):175–199, 2013.

P. Stevens. Bidirectionally tolerating inconsistency: Partial transformations. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014)*, volume 8411 of *LNCS*, pages 32–46. Springer, 2014.

R. V. D. Straeten, J. P. Puissant, and T. Mens. Assessing the Kodkod model finder for resolving model inconsistencies. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011)*, volume 6698 of *LNCS*, pages 69–84. Springer, 2011.

A. Tarski and S. R. Givant. *A Formalization of Set Theory Without Variables*, volume 41 of *AMS Colloquium Publications*. American Mathematical Society, 1987. ISBN 978-0-8218-1041-5.

Tata Research Development and Design Centre. ModelMorf. http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm.

J. F. Terwilliger, A. Cleve, and C. Curino. How clean is your sandbox? - towards a unified theoretical framework for incremental bidirectional transformation. In *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT 2012)*, volume 7307 of *LNCS*, pages 1–23. Springer, 2012.

E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.

K. Voigt. *Structural Graph-based Metamodel Matching*. PhD thesis, University of Desden, 2011.

J. Voigtländer. Bidirectionalization for free! (pearl). In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 165–176. ACM, 2009.

J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 181–192. ACM, 2010.

M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Gradual refinement. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC 2010)*, volume 6120 of *LNCS*, pages 397–425. Springer, 2010.

E. D. Willink, H. Hoyos, and D. S. Kolovos. Yet another three QVT languages. In *Proceedings of the 6th International Conference on Theory and Practice of Model Transformations (ICMT 2013)*, volume 7909 of *LNCS*, pages 58–59. Springer, 2013.

Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 164–173. ACM, 2007.

Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the 7th joint meeting of the European Soft-*

ware Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2009), pages 315–324. ACM, 2009.

Index