An extensible dynamically typed object orientated language with an application to model transformations

Laurence R. Tratt

August 2005

Thesis submitted in partial fulfilment for the degree of
Doctor of Philosophy

King’s College London

Abstract

Dynamically typed object orientated languages such as Python are increasingly seen as a viable implementation technology for software systems. Despite the run-time flexibility that such systems present, few present any means of extending the base language. Although languages such as LISP provide features for extending the base language via a macro system, few modern languages are capable of compile-time meta-programming, and of those that do, many of the most powerful are statically typed functional languages. In this thesis I first present a novel dynamically typed object orientated language *Converge*, which can be extended via its compile-time meta-programming facility. This facility can then be used to extend Converge's syntax, allowing Domain Specific Languages (DSLs) to be embedded directly within Converge.

I then use Converge to tackle the problem of model transformations. Model transformations are of increasing importance in the development of large systems, whose models need to be manipulated into many different forms. Model transformations written in general programming languages are typically bloated, buggy, and inflexible and perform beneath reasonable expectations. The difficulties of implementing model transformations have hampered practical progress in this area. In this thesis I show a large scale example of a model transformation approach *MT* implemented as a Converge DSL. I then use this as a basis for a novel change propagating model transformation approach *PMT* which explores practical approaches to this challenging problem.

Acknowledgements

I would like to thank those who have helped me in my research work during the last five years. Two people in particular have had a big impact on this work. Tony Clark guided me through some of the early stages of this work, but more importantly opened my eyes to the possibilities out there, and who put up with my constant questions and chattering. John Fisher played a similar guiding rôle, and suffered a similar verbal attack, during my school days. I don't think I'd have got to this point without their guidance.

My thanks to all my friends from over the years who've given me support in one form or another – there really are too many to mention everyone here, so I apologise in advance to those who I've forgotten to mention. Arlene Ong & Kelly Androutsopoulos have been constant companions at King's, and I don't know quite how different this would all have been without the pair of them. It might have been a bit quieter, but also lot duller. To my friends outside of the King's bubble who will never read this – Eliot, Martin, Steve, Vasa, amongst many others – you can all claim prize money of £1000 if you report this offer to me within 7 days of the publication of this thesis.

To both bodies that funded me – the EPSRC for my first year and, indirectly, Tata Consultancy Services thereafter – thank you. I assure you both that I have put your money to the best possible use.

This thesis would not have been possible without industrial quantities of metal. It would be remiss of me not to attempt to embarrass my future self by reminding myself of this. I thus thank all those who've recorded discordant and insanely distorted guitars over a backdrop of caveman drumming, and topped it all off with pre-*Homo sapiens* vocals. I'd like to particularly thank the tiny minority who've taken this route and used it to create memorable tunes — you rock.

To my parents, who opened up the opportunities that led me to this point and have supported me all the way through, you've made this all possible — Mum, Dad, the cheque is in the post, honest. Since I will never be able to repay you what I owe, it's probably not worth me even starting to try, so I'll say simply: I love you both. This thesis is dedicated to Granny who would have loved to have seen me see this through.

Contents

Chapter 1.

Introduction

1.1. Overview

1.1.1. An extensible programming language

When developing complex software systems in a General Purpose Language (GPL), it is often the case that one comes to a problem which is not naturally expressible in the GPL used to develop that system. In such cases, the user has little choice but to find a suitable workaround, and encode their solution in as practical a fashion as they are able. Whilst such workarounds and encodings are often trivial, they can on occasion be exceedingly complex. In such cases the system can become far less comprehensible than the user may have wished. Although Steele argues that 'a main goal in designing a language should be to plan for growth' [Ste99], most modern GPLs only allow growth through the addition of libraries. The ability of a user to extend, or augment, their chosen programming language is thus severely restricted.

Domain Specific Languages (DSLs) are an attempt to work around the lack of expressivity in a GPL by presenting the user with a mini-language targeted to the particular domain they are working in. Mernik *et. al* [MHS03] define DSLs as 'languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general purpose programming languages in their domain of application'. Traditionally DSLs – for example the UNIX `make` program – have been implemented as entirely stand alone applications. Hudak contrasts the consequent high costs of traditional DSL implementation with Domain Specific Embedded Languages (DSELs) [Hud98]. DSELs contrast with traditional DSLs in that they are a language within a language; in other words the DSL is embedded within a GPL. In so doing, the DSEL can pick up many of the benefits of the surrounding GPL. However Hudak specifically limits his visions to DSLs embedded in strongly typed functional languages such as Haskell, relying on the particular feature sets that such languages offer.

Wilson argues that programming languages need to allow their syntaxes to be extended if powerful DSLs are to be exploited to their maximum potential [Wil05]. Hudak's vision is thus fundamentally limited since he expressly forbids any form of syntax extension to the host GPL. Part of the reason for this may be that few modern languages are capable of syntax extension. Although LISP's macro facilities are well known, its syntactic minimalism is far removed from modern programming languages and whilst the syntax is inherently flexible, it is not possible to change it in a completely arbitrary fashion. Nemerle [SMO04] is a statically typed OO language in the Java / C# vein, which includes a macro system that permits a limited form of syntax extension. Bravenboer and Visser perhaps come closest to the ideal vision of syntax extension with the `MetaBorg` system which allows language grammars to be extended in an arbitrary fashion [BV04]. However `MetaBorg` is a heterogeneous system meaning that the language being extended is generally different than the language doing the extension. In order to use such a system, one needs to be expert in three entirely separate systems (the language being extended, the language doing the extension and the 'emulation' of the language being extended) in order to produce a quality implementation, which is a significant barrier to use.

A primary aim of this thesis is to present an extendable programming language. Since the GPLs that I use most frequently for my research [Tra05] are dynamically typed Object Orientated (OO) languages such as Python [vR03], this thesis further aims to present an extendable dynamically typed object orientated language. Dynamically typed OO languages such as Python, Ruby [TH00] and Smalltalk [GR89] are increasingly recognised as having an important rôle to play in the development ecosphere, particularly for the rapid development of software whose requirements evolve and change as the software itself develops [Ous98]. Although they have traditionally been labelled somewhat dismissively as 'scripting languages', modern dynamic language implementations can often lead to programs which are close in run-time performance to their statically typed counterparts, whilst having a significantly lower development cost [Pre00].

In contrast to a heterogeneous system such as `MetaBorg`, a language which successfully meets this thesis's aims would need to be entirely homogeneous in nature. In order to achieve this, the language thus needs some way to execute arbitrary code at compile-time. To the best of my knowledge, the only dynamically typed OO language capable of this is Dylan [BP99], which is a heterogeneous system since its macro language is distinct from the main language. Relatively recently languages such as the multi-staged MetaML [Tah99] and Template Haskell (TH) [SJ02] have shown that statically typed functional languages can house powerful compile-time meta-programming facilities where the run-time and compile-time languages are one and the same. Whereas lexing macro systems typically introduce an entirely new language to a system, and LISP macro systems need the compiler to recognise that macro definitions are different from normal functions, languages such as

TH move the macro burden from the point of definition to the macro call point. In so doing, macros suddenly become as any other function within the host language, making this form of compile-time meta-programming in some way distinct from more traditional macro systems. Importantly these languages also provide powerful, but usable, ways of coping with the syntactic richness of modern languages.

Since languages such as MetaML and TH are concerned with different aspects of program development (such as statically determinable type-safety), it is less than clear whether or not a dynamically typed OO language could satisfactorily house a similar compile-time system. In this thesis I present the Converge programming language, which can be seen in many ways as a Python derivative, both syntactically and semantically. However, Converge is a more experimental multi-paradigm language than Python and its ilk. It has been designed, in part, to explore if, and how, various language features can be integrated together. In this thesis I present the main Converge language along with its TH-derived compile-time meta-programming facilities, explaining the impact this has had on the language's design since it is important that the addition of such a feature does not unduly complicate other areas of the language. I then show how Converge allows its syntax to be directly extended, thus allowing DSLs to be embedded in Converge in an entirely natural fashion. In order to validate Converge's approach to DSL implementation, the following subsection details a substantial problem, which is then implemented within Converge.

1.1.2. Model transformations

In recent years the movement towards developing software with the use of models has increased rapidly. Organizations are increasingly seizing the opportunity to move their intellectual property, business logic, and processes, from source code into models, allowing them to focus on the important aspects of their systems, which have traditionally been buried – and sometimes lost – in the mélange resulting from the use of general purpose languages (GPLs) such as Java and C++. For the purposes of this thesis, models can be assumed to be UML [BJR00] models, or similar. This increasingly sophisticated use of models has led to the desire to transform models into various different forms. Needs range from the mundane (e.g. simple data format conversion) to the traditional (e.g. model compilers) to the innovative (e.g. transformations which can propagate changes after an initial transformation).

Model transformations are the key to solving this very fundamental problem, and are vital if the use of modelling is to reach its full potential [BG02, GLR+02, Whi02]. A simple definition of a model transformation is that it is a program which mutates one model into another; in other words, something akin to a programming language compiler. Of course, if this simple description accurately described model transformations, then we would be faced with a relatively simple and uninterest-

ing problem to solve – GPLs and traditional techniques would almost certainly suffice to solve this problem satisfactorily, as they do with many other problems.

In practise writing model transformations is difficult, particularly so when GPLs are the only tool available to write them. Whilst such languages present good environments for solving many classes of problems, model transformations make frequent use of techniques and features which are either absent or cumbersome to use in GPLs. Such features thus need to be encoded in a roundabout fashion in the host language. For example, models are most naturally represented as graphs; encoding backtracking over a graph, of the kind frequently needed by model transformations, in a GPL is a surprisingly challenging task. Performance issues also feature – for example the potential size of models can necessitate against the eager evaluation of arbitrary structures. Encoding suitable techniques using the facilities available in a GPL is of course possible, but is tedious, error-prone and can lead to inefficient execution. More fundamentally it prevents the transformation writer from concentrating on the important aspects that they need to express; it hinders those who later wish to understand code which relies on knowledge both of the problem being solved and the elaborate encodings used to solve it; and lessens the potential for reuse.

To alleviate these problems, a number of different approaches dedicated to model transformations have recently been proposed. Most approaches have been created with the assumption that existing GPL approaches are unsatisfactory. However few, if any, approaches are explicit about this assumption and none analyse traditional approaches sufficiently. Perhaps because of this, most proposed model transformation approaches are somewhat ‘hit and miss’ in terms of tackling the problem more successfully than existing approaches. Significantly, I believe that most model transformation approaches are largely similar to each other. Without reasonable evaluation of all the potentially major different types of model transformation approach, it is hard to be sure that any particular approach is as good as can be reasonably achieved. There is also a tendency to assume that there is a suitable ‘one size fits all’ solution to the problem because of the narrowness of the solutions being attempted. At this stage in the development of the area, it seems sensible to assume that different solutions may be required to tackle different aspects of the problem.

It is my contention that the difficulty of implementing model transformation systems is one of the chief reasons for the relative simplicity of most current model transformation approaches. Only a small proportion of proposed approaches appear to be implemented; of those that do have an implementation, some are too limited to perform any meaningful task. Since model transformations are an inherently practical topic, implementations are vital for assessing and evolving new ideas. A long and labour intensive idea-implement-assess cycle seriously inhibits such experimentation. The area of model transformations thus finds itself in something of a vicious cycle: as a relatively new area,

experimentation is vital for discovering the merits of different approaches and techniques, yet the difficulties of creating implementations inhibits experimentation.

In respect to model transformations, this thesis has two complementary aims. The first is to demonstrate how a complex DSL fits within Converge, and how it is implemented. The second is to explore the model transformations field by investigating new types of model transformations.

Types of model transformation

In this thesis various types of model transformations are identified (see section 2.3), with two being of particular significance. These two types can be summarised as follows:

Stateless model transformations take in a source model and produce a target model in a similar vein to a programming language compiler. Once the transformation has been run it is complete, and the only action to be taken when rerunning the transformation is to create an entirely fresh target model from the source model.

Change propagating model transformations are only relevant after an initial transformation from a source to target model. Subsequent to such an initial transformation they are capable of propagating changes made to the source model to the target model in a non-destructive fashion.

The majority of existing model transformation approaches are only capable of expressing stateless model transformation. Although stateless model transformations are widely recognised as being important, change propagating model transformations are also of considerable interest.

In this thesis I present a stateless model transformation language MT which serves as an example of implementing a complex DSL within Converge. I then present a novel change propagating model transformation language PMT.

1.2. Overall aims of the thesis

Consistent with the issues outlined in section 1.1, this thesis has the following complementary aims:

1. To provide an extensible dynamically typed OO programming language which allows DSLs to be embedded within it.

2. To provide a non-trivial example of a DSL within the extendable programming language.

3. To examine new approaches for expressing stateless and change propagating transformations.

1.3. Overall thesis structure

In order to satisfy the stated aims, this thesis is organised into four main parts:

1. An analysis and review of compile-time meta-programming systems, and model transformations.
2. I present the design of a new imperative programming language named Converge, designed to facilitate the implementation of DSLs.
3. Converge is used to express a simple, but powerful rule-based stateless model transformation system.
4. The rule-based approach is then extended to define a novel change propagating model transformation approach.

1.4. Contributions

The main contributions of this thesis are as follows:

- The design of the Converge programming language.
- A clear identification of significant types of model transformations.
- The use of Converge to implement a practical rule-based stateless model transformation system, which severs as a non-trivial example of using Converge to implement a DSL.
- The use of Converge to explore practical approaches to change propagating transformations.

1.5. Detailed synopsis

Chapter 2 introduces the concepts of DSLs and model transformations. This chapter motivates the need to consider model transformations as a distinct and unique problem via an analysis of the problem they aim to solve. This leads to a categorization of some significant types of model transformation, and the establishment of a simple method that model transformation approaches conform to. By defining the problem space thus, a wide solution space is also defined.

Chapter 3 reviews and analyses the major approaches to compile-time meta-programming and model transformations. The analysis leads to two choices being made. Firstly I present a dynamically typed OO language capable of compile-time meta-programming (chapter 4), and secondly I

choose to provide new approaches for expressing stateless and change propagating transformations (chapters 5 and 6).

Chapter 4 presents the design of a new dynamically typed OO language Converge designed to facilitate the implementation of DSLs. Converge is an imperative programming language, capable of compile-time meta-programming, and with an extendable syntax.

The chapter concludes with a user extension to Converge which allows simple modelling languages to be embedded within Converge. As well as being a practical demonstration of Converge's features, this facility is used extensively throughout the remainder of the thesis.

Chapter 5 constructs a rule-based stateless model transformation approach. This serves as a non-trivial example of implementing a DSL in Converge.

Chapter 6 details a more sophisticated model transformation solution aiming to tackle the problem of change propagation. The rule based approach of the previous chapter is augmented with mechanisms for ensuring that transformations can be preserved after an initial transformation, and are capable of propagating changes made to a source model subsequent to its initial transformation. In so doing, a number of insights into the challenges of change propagation, and solutions to several of these problems, are presented.

Chapter 7 Conclusions.

1.6. Previous availability of material

1.6.1. Publications

Several parts of this thesis have appeared in identifiable form in previous publications, all authored solely by myself. An early version of parts of chapters 2 and 3 can be found in the following journal publication:

> Laurence Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112-122, May 2005.

A slightly augmented version of chapter 4 appeared in the following symposium publication:

> Laurence Tratt. Compile-time meta-programming in a dynamically typed OO language. Proc. Dynamic Languages Symposium, October 2005.

Large parts of chapters four have previously appeared in the following two technical reports:

Laurence Tratt. Compile-time meta-programming in Converge. Technical report TR-04-11, Department of Computer Science, King's College London, December 2004.

Laurence Tratt. The Converge programming language. Technical report TR-05-01, Department of Computer Science, King's College London, February 2005.

Chapter five appeared largely verbatim in the following technical report:

Laurence Tratt. The MT model transformation language. Technical report TR-05-02, Department of Computer Science, King's College London, May 2005.

1.6.2. Software

This thesis has led to the creation of several pieces of software. All software in this thesis can be downloaded from `http://convergepl.org/`.

1.7. Thesis conventions

Please note that some code has had to be reformatted in order to make it fit on the printed page.

Also note that whilst error messages and so on frequently give complete pathnames for the files they refer to, in the interests of brevity these have generally been cut down to show only the leaf name of the file involved.

Chapter 2.

Background

This chapter is intended to fill in the background information necessary for the work presented in this thesis. The concept of DSLs and model transformations are explained and motivated in more detail. Also spread throughout this section are the definitions of a number of existing and unfamiliar terms – and explanations of unfamiliar concepts – which are used throughout this thesis.

2.1. Domain specific languages

Mernik *et. al* [MHS03] define DSLs as 'languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general purpose programming languages in their domain of application'. The canonical example of a DSL is the widely available UNIX program `make`, which allows dependencies between files to be expressed. If a file `x` upon which a file `f` depends, has been updated it will force `f` to be recomputed. Although newer versions of `make` have added (sometimes incompatible) features upon this basic vision, in essence `make` is only capable of expressing simple dependencies.

A complete real world example of input to `make` is as follows:

```
echo: echo.c echo.h
  cc -o echo echo.o
```

In this fragment, the `echo` binary will be relinked (with a user-specified command) if either the `echo.c` or `echo.h` files has been changed since the last link. `make` also ensures that all C files (files whose names end in '`.c`') are recompiled if they have been changed since their last compilation. The expressive power of the `make` DSL for its chosen domain can be gauged by comparing the simple input to the following GPL pseudo-code:

```
if not futil.exists("echo.o") or (futil.last_modified("echo.c") > \
  futil.last_modified("echo.o")):
  sys.shell("cc -c -o echo.o echo.c")

if not futil.exists("echo") or (futil.last_modified("echo.c") > \
  futil.last_modified("echo")) or (futil.last_modified("echo.h") > \
```

```
futil.last_modified("echo")):
sys.shell("cc -o echo echo.o")
```

As this example clearly shows, for its intended domain, `make` allows users to express file dependencies in a concise fashion compared to a GPL alternative. As this hopefully suggests, the aim of a DSL is not to provide a generic solution to a wide category of problems; rather a DSL should aim to provide a succinct way of expressing solutions to a very specific problem. It should also be noted that although this example, and most of the other DSLs mentioned in this thesis, are computer related, in general DSLs can be written for any domain. For example, one could construct a DSL designed to allow banks to express the changing rates of interest on their accounts. van Deursen *et. al* provide a comprehensive review of much of the material about DSLs [vDKV00].

2.2. Model transformations

Although transformations in general are a subject that has seen much research over the past decades, model transformations are a relatively new area of research. Furthermore much of the research on non-model transformations has been targeted at very specific applications of transformations, nullifying much of its applicability to model transformations which aim to encompass a much broader and general scope [GLR+02]. Therefore, in order to define what a model transformation is, reference to previous transformations work is of limited use.

Model transformations have an intuitive meaning: they are programs which change one model into another. Although this simple definition encompasses one of the most important tasks of model transformations, it fails to capture other tasks which they model transformations aim to facilitate. This definition also gives little sense as to *how* model transformations might work, or how one might go about creating a model transformation. In this section, I present a simple example of a model transformation which allows many of the different aspects of model transformations to be highlighted.

2.2.1. Transforming between two similar modelling languages

Figure 2.1 shows the metamodels of two similar modelling languages which will be used in most of the examples in this chapter. A metamodel is literally the model of a model; it is a set of constraints which determine its valid instances. A metamodel and a model are analogous to a BNF grammar and a particular textual input. In the interests of brevity I do not formally define the semantics of these languages, assuming that equivalent elements in either modelling language have the same semantics unless otherwise stated. The modelling language *ML1* in figure 2.1(a) supports directed associations and package inheritance; the latter is a mechanism for structuring models as found in e.g. Appukuttan *et. al* [ACE+02]. For the purposes of this example, a package A which inherits from a package B is

considered to posses a copy of all the elements in B. Figure 2.1(b) shows the modelling language *ML2* which does not provide support for package inheritance but allows bidirectional associations. Because of the large overlap between the two metamodels, many models can be instances of either metamodel; however many models will make use of the conflicting features of one or the other modelling language and are thus not directly interchangeable. This is representative of the real world where two modelling tools store and manipulate models in only marginally different fashions, yet still end up preventing users from interchanging their models between them[1].

Figure 2.2(a) shows a typical example of package inheritance in a simple model of a company – different aspects of the company have been separated into different packages to aid comprehension. The *Company* package then inherits the relevant sub-packages, meaning it contains all the relevant parts of the company model. Since this model makes use of package inheritance, it must be an instance of the *ML1* modelling language; any tool which understands the *ML2* modelling language will not be able to interpret the model correctly. Intuitively all that is needed to obtain the *ML2* equivalent of the model in figure 2.2(a) is to flatten the package inheritance by copying elements from the super-packages into the sub-package. Figure 2.2(b) shows such a model created by hand (with the super-packages not copied over since they are redundant).

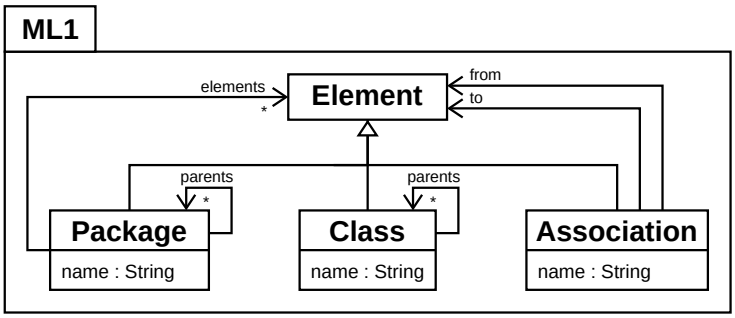
In this example creating the *ML2* model from the *ML1* model by hand is simple and relatively quick. However it is clear that this is not a scalable approach. Performing a similar task on larger models would be both tedious and error-prone; it would be a daunting prospect to frequently repeat this task. The problem at hand is thus how to provide users with a practical means of automating this task.

2.2.2. Encoding the example in a GPL

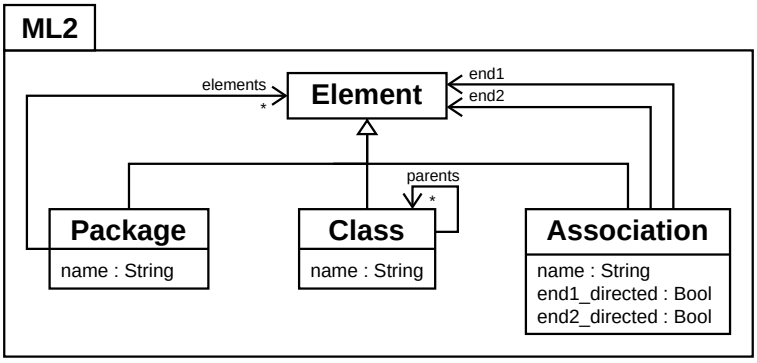
Any first attempt at an automatic model transformation is likely to be created in a GPL. A relatively, but not completely, naïve non-OO attempt, expressed in a fairly high-level pseudo-code, might look as follows:

```
func transform(element : ML1.Element) : M2.Element:
  if type(element) == ML1.Package:
    package_elements := []
    for package_element in element.elements:
      package_elements.append(transform(package_element))
    parents_temp = elements.parents.copy()
    for parent in parents_temp:
      for parent_element in parent.elements:
        package_elements.append(transform(parent_element))
        parents_temp.extend(parent_element.parents)
    return new M2.Package(element.name, package_elements)
```

[1] As purely anecdotal evidence, in 2002 I informally evaluated the interoperability of approximately eight UML modelling tools. I was surprised to find that few tools could load models saved by other tools, and frankly shocked to discover that some tools could not even be relied upon to load in their own saved models in all cases.



(a) The *ML1* modelling language.



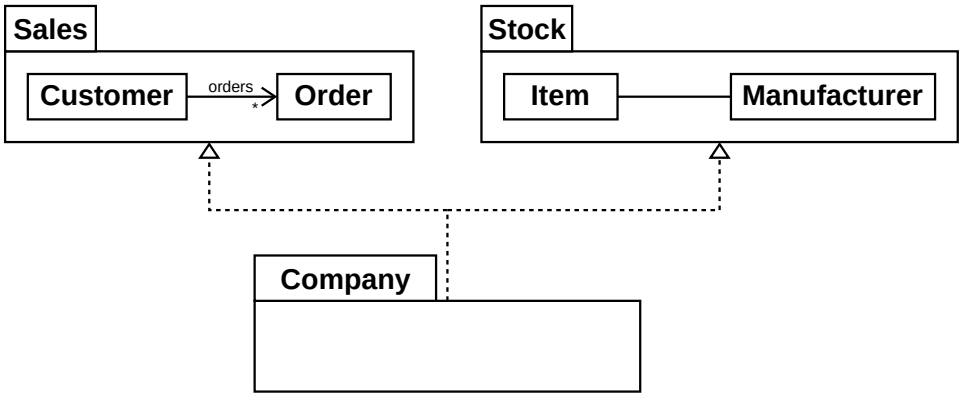
(b) The *ML2* modelling language.

Figure 2.1.: Language metamodels.

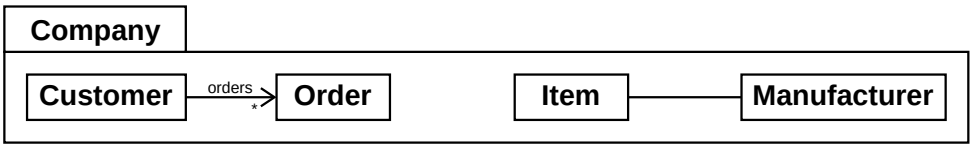
```
  elif type(element) == ML1.Class:
    parents := []
    for parent in element.parents:
      parents.append(transform(parent))
    return new ML2.Class(element.name, parents))
  elif type(element) == ML1.Association:
    return new ML2.Association(element.name, transform(element.from),
                               transform(element.to), true, false)

func main(model_in : Seq{ML1.Element}) : Void:
  model_out := []
  for element in model_in:
    model_out.append(transform(element))
```

The essential idea here is to transform every element in an instance of the *ML1* language into its counterpart in the *ML2* language; elements from inherited packages are brought into the child packages and the package inheritance itself disappears. This approach has two immediate, and closely related, problems: elements can easily be duplicated during the transformation e.g. if a class S is specialized by two other classes then two copies of S will appear in the target model; cycles in the model created by associations between two classes cause the transformation to loop without termination. Both problems are related to the fact that models are graph structures.

If one ignores the flaws in this particular example, then two stylistic issues can be discerned. Most obviously, the entire transformation has been squeezed into one function. Clearly this is not a scalable approach. However, factoring out the code from the body of each branch of the `if` statement into separate functions reveals another limitation of this approach. The expressions in the condition of



(a) Company model with package inheritance.



(b) Company model without package inheritance.

Figure 2.2.: Example models.

each branch of the `if` statement are an inherent part of each *rule* in the transformation, since they determine whether the rule can be executed or not. Separating the body of each branch into a function creates a dichotomy between the two aspects of the rule.

A more sophisticated approach which uses a cache to determine which elements have already been transformed, and uses the overloading facilities of many OO languages allows one to encode a more rule-based approach (see section 3.1) to overcome the problems outlined thus far:

```
class ML1_To_ML2:
  func transform(model_in : Seq{ML1.Element}) : Seq{ML2.Element}:
    self.processed_elements = [] // These two sequences will always
    self.processed_results = []  // be of the same length
    model_out := []
    for element in model_in:
      model_out.append(self.transform_element(element))
    return model_out

  func transform_element(element : ML1.Element) : ML2.Element:
    if self.processed_elements.index(element) != -1:
      processed_element = self.processed_results[
                            self.processed_elements.index(element)]
    else:
      processed_element = self.transform_rule(element)
      self.processed_results.append(processed_element)
    return processed_element

  func transform_rule(element : ML1.Package) : ML2.Element:
    package_elements := []
    for package_element in element.elements:
      package_elements.append(self.transform_element(package_element))
    for parent in element.parents:
      for parent_element in parent.elements:
        package_elements.append(self.transform_element(parent_element))
    return new M2.Package(element.name, package_elements)

  func transform_rule(element : ML1.Class) : ML2.Element:
```

```
    parents := []
    for parent in element.parents:
      parents.append(self.transform_element(parent))
    return new ML2.Class(element.name, parents))

  func transform_rule(element : ML1.Association) : ML2.Element:
    return new ML2.Association(element.name, transform(element.from),
                               transform(element.to), true, false)
```

Our simple example is now successfully encoded – we now no longer transform elements twice nor can the transformation enter into infinite cycles. By overloading the *transform_rule*, the reliance on a large `switch`-style statement has been removed. However this latter success is somewhat illusionary because of the lack of expressive power afforded by this approach. The only form of constraint that each rule can express is about the type of source model element it can transform. This would not be sufficient to express, for example, a rule which transforms packages whose names begin with a '_' in a different way from packages whose names do not begin with a '_' (such a rule may be used to enforce naming conventions). Complex constraints such as this are often a part of model transformations; method overloading does not provide sufficient expressive power.

Despite the lack of generality of the overloading approach, this transformation is still a considerable improvement over its flawed predecessor. A significant problem however with this transformation is its relative size to our naïve solution, with a large amount of boiler plate code and general machinery[2] added in order to get the transformation to work correctly. Worryingly, the necessary machinery is not confined to certain aspects of the transformation – it pervades every aspect. Whilst this machinery is of manageable proportions for a small transformation, one can surmise that such an approach will swiftly lead to the substance of the underlying transformation being swamped as the transformation grows larger. Thus whilst we have a solution for the original problem, it seems unlikely that such an approach will scale appropriately.

2.2.3. A change propagating example

In the previous section, I motivated the need for model transformations by exploring the need to perform a transformation between models stored in different tools, and the difficulties in trying to write such a transformation in a GPL. I classify that transformation example as a *unidirectional stateless* transformation. It is unidirectional because it can only transform an instance of *ML1* into an instance of *ML2*. It is stateless because running the transformation when the source model has changed results in the creation of an entirely new target model even if it is an exact duplicate of the model that already exists. Although such a transformation can be of practical use in integrating together different tools, it tackles only part of the problem.

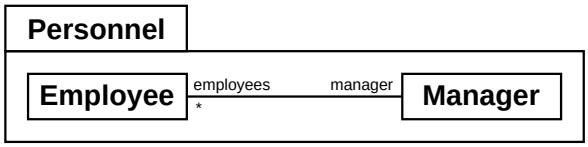
[2] I use the term 'machinery' to denote code which is necessary to construct a running system but which detracts from the users' focus in creating the system.

One enticing future scenario is when tools which specialize in different aspects of modelling can be used together throughout the development life cycle [Tra05] e.g. a UML modelling tool *UT* and a Java modelling tool *JT*. In such a scenario, a model is not just transformed between different tools once, but may be edited multiple times in each tool. For example, an initial model may be created in *UT*, transformed and subsequently edited in *JT*, before high-level architectural changes are applied in *UT* which one expects to see reflected in *JT*. A similar, although more linear, scenario involving incremental model development is explained in Becker *et. al* [BHW04]. The general aim underlying such scenarios is to allow the user to leverage the particular specialities of different tools at varying points in the development life cycle.

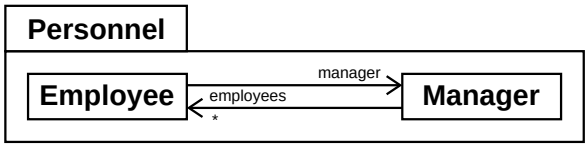
The significant challenge raised by this scenario can be seen in figure 2.3. Imagine first that one has the model in figure 2.3(a) (an instance of the *ML2* modelling language) in a tool *MT2*. One then transforms this model into an instance of the *ML1* modelling language for use in another tool *MT1*. The result of the transformation from *ML2* to *ML1* is shown in figure 2.3(b), which contains two directed associations. Now if the user changes the model in *MT2*, what might the result be on the model in *MT1*? Being stateless in nature, the example presented in the previous section would simply erase whatever was in *MT1* and create an entirely fresh model.

A more sophisticated approach would be for the transformation to attempt to perform the minimum alteration to the target model to propagate the changes, leaving it otherwise intact. In order to do this, the transformation needs to somehow recognise those elements in the model in *MT1* which relate to those in *MT2* and use, or change, them appropriately. This initially seems fairly trivial – for example the *Employee* class is obviously shared in both models. However, consider the bidirectional association in *MT2* which is non-trivially related to two directed associations in *MT1* – how should a transformation recognise such a relationship? One could search in an *MT2* model for a pair of directed associations whose names appear to correspond to that of a bidirectional association in *MT1*, but such correspondences may be pure coincidence (the user being free to name associations as they so wish), which would lead to an incorrect change propagation. One important type of propagation results from the deletion of an element in *MT2* which should result in the appropriate deletion of elements in *MT1*. Unfortunately, no matter how clever a property-based calculation might be in determining element equivalence, if the transformation has no record of which elements in *MT2* relate to those in *MT1* it will be unable to perform such a deletion reliably. At best it will not delete elements in *MT1* that it should; at worst it may result in the accidental deletion of elements in *MT1*.

The scenario is further complicated by the fact that it is rarely acceptable for the transformations between tools to reconstruct a model from scratch if it already exists. In other words, although the original transformation from a model in *UT* to a model in *JT* creates the target model from scratch,



(a) The *Personnel* package in the *ML2* language.



(b) The *Personnel* package in the *ML1* language.

Figure 2.3.: Models with different types of associations.

subsequent transformations need to alter the models already present rather than wiping the model and treating every transformation as if it were an initial transformation (even if it perfectly recreates said model). There are two main reasons for this. Firstly continually creating large models from scratch can be prohibitively inefficient, particularly if only a small portion of one model has been changed. Secondly, the user may in the target model manually create elements which do not directly relate to the source model (e.g. in the *UT* and *JT* example, this could involve adding Java specific details into the model in *JT*). Subsequent updates must not destroy manually added elements, or the links to them, simply because they are not a direct part of the transformation.

It is important to note that the scenario given here is deliberately limited compared to the general case. It calls only for changes in *UT* to be propagated to *JT*, not vice versa. A solution for the general case would utilize a *bidirectional* transformation that could also propagate any relevant changes made in *JT* to *UT*. True bidirectional transformations present a number of challenges above and beyond those tackled in this thesis, and by most current model transformation technologies (see section 3). Consequently they are largely ignored in this thesis – however all of the challenges listed in this thesis apply equally to bidirectional transformations.

2.2.4. A method for model transformations

Based on examples such as those just presented, a simple method for model transformations can be discerned which can significantly aid understanding of the general problem. It also allows the comparison of different approaches by describing where, and how well, any approach fits into the method. Because, as shall be seen in chapter 3, there are various different categories of model transformations, this method is intentionally high level and therefore applicable to the majority of practical approaches. For example, in a simplistic approach encoded in a GPL this method would apply to the entire program; in a rule based approach, this method could be seen to be applied to each rule. The example in

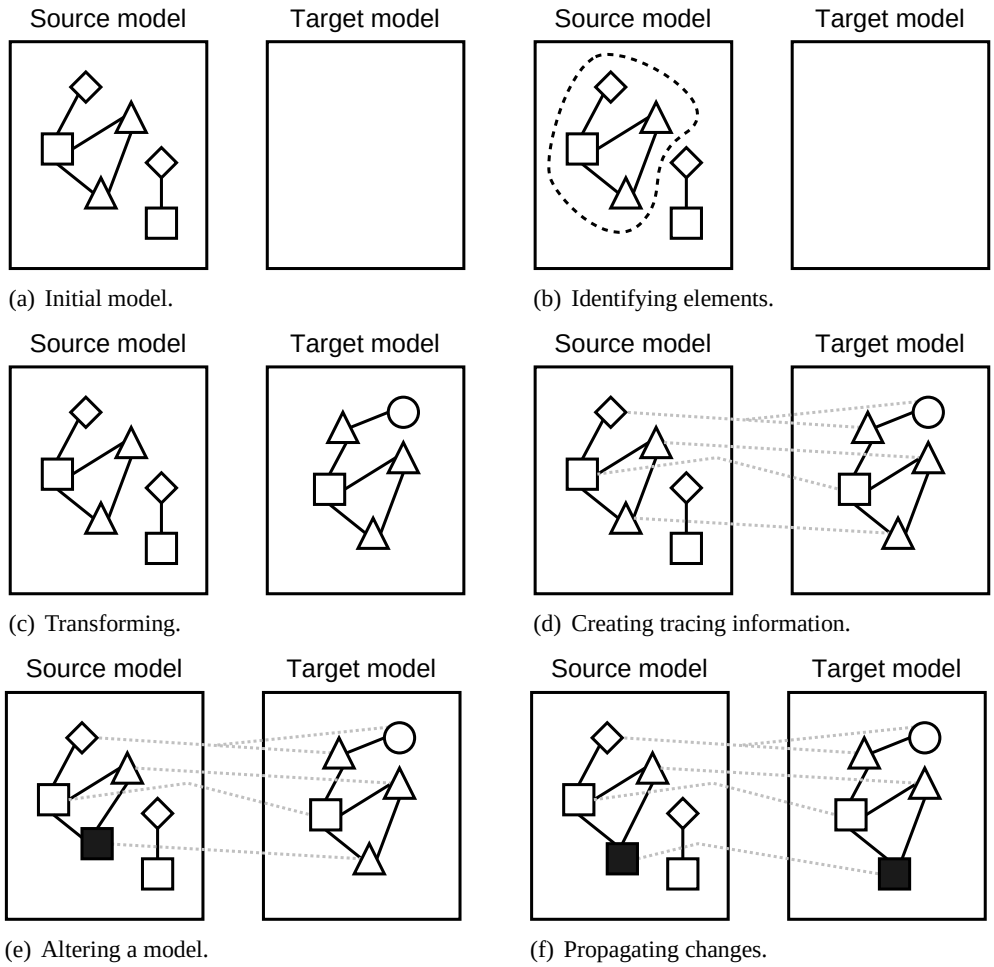


Figure 2.4.: Transforming a model.

figure 2.4 is intended to help visualise these parts:

1. Searching a model to identify appropriate elements to transform.
2. Transforming elements.
3. The retention (in some manner) of tracing information recording which elements in a model are related by the transformation to elements in other models.
4. Detecting updates in one model involved in the transformation and performing relevant operations in the transformations other affected models.

Whilst a minimal approach to model transformations need only perform parts one and two, a complete approach would be capable of performing all parts: a model transformation technology which limits itself to merely taking in one model and producing another model out fails to tackle all the required problems outlined in section 2.2.3. However, although the method is comprised of four main parts, it is not necessary for these parts to correspond to distinct phases of execution. Parts one and

two are often partially intertwined and it would be surprising if parts two and three in particular were implemented as separate phases because the required information for part three will be determined by what happens in part two.

2.2.5. Challenges raised by the examples

These simple examples are intended to give the reader an idea not only of the overall problem that model transformations are attempting to tackle, but also the issues raised in tackling the problem. In short the two main challenges of the 'what' and the 'how' can be summarised as follows:

1. The desire to reduce the necessary, but largely irrelevant, machinery which can swamp the essence of any given model transformation.

2. The need not only to transform an initial instance of a metamodel *ML1* into an instance of a metamodel *ML2,* but also to propagate subsequent changes made to the *ML1* model non-destructively to the *ML2* model.

A third challenge could be considered to be the desire to create bidirectional transformations. However the problem of bidirectional transformations is not explicitly considered in this thesis chiefly because it requires, at a minimum, practical solutions to the problems listed in the two challenges above.

2.3. Notable categories of model transformation

There are many categorisation criteria that one can apply to model transformations. For example one could categorize the way they are used, the paradigm they exploit [**?**], the features they provide [GGKH03] and so on; chapter 3 details some existing categorizations in more detail. Already in this chapter a few types of model transformation have been of particular note and, since they recur throughout this thesis, it is useful to have fixed terms to refer to them. Starting from the intuitive / naïve notion of a model transformation being a program which 'consumes an instance of the metamodel *ML1* and produces an instance of the metamodel *ML2*', the following types of model transformations are particularly significant:

Uni/bidirectional transformations. Implicit in the naïve notion of a model transformation is the idea that the transformation is unidirectional. In other words, the transformation is incapable of taking an instance of the metamodel *ML2* in and producing an instance of the metamodel *ML1* out. There are several reasons why a particular transformation is unidirectional, two of

the most important being: the transformation loses information and hence there is insufficient detail in instances of *B* alone to perform a full reversal; bidirectional transformations tend to be considerably more difficult to write than unidirectional transformations. Note that this simple definition of bidirectionality does not necessarily imply that a reverse transformation will perfectly recreate the original source model.

In the interests of simplicity, throughout this thesis I generally refer to 'source' and 'target' models although this should not be taken to mean that a transformation between two models labelled thus implies that only unidirectional transformations can exist between the two.

Multi-domain transformations In this thesis, as in most work in this area, the general assumption is that two models or *domains* are involved in a transformation – however it is important to realise that in the context of this thesis, this simplification is purely to aid exposition. Multi-domain transformations, though rarer than those involving only two domains, are important tools. A simple example is a model diff transformation (analogous to the UNIX `diff` tool [HM76]) which takes in two models and produces a third.

The reason for the use of the term 'domain' is to allow one other notable type of transformation: so-called 'update in place' transformations. This is a transformation which alters its source model into the target model, rather than operating a target model which is entirely separate from the source model.

Stateless transformations. An example of a real world stateless transformation is a compiler: in simplified terms, it takes in a source file, transforms it, and writes out a binary file. Once done, the transformation is complete, and if the source file changes the entire transformation is rerun in an identical fashion regardless of the existence of the binary – an existing binary file will simply be overwritten. Note that the stateless classification does not imply that transformations need necessarily be uni-directional: for example, decompilers can reverse the compilation process (albeit imperfectly) in exactly the same stateless fashion as a compiler.

Change propagating transformations. These are transformations which can not only perform a one-off transformation but can propagate subsequent changes from some or all of its constituent domains without the need to rerun the entire transformation. In the context of this thesis there is an extra implication on change propagating transformations which is that they propagate their changes non-destructively; in other words, they do not blindly overwrite the target when propagating changes. Since this is a far trickier proposition than a stateless transformation, relatively few such transformations exist in practise at the moment, although as seen in section 2.2.3 there is a real need for such transformations in a modelling context. Compuware's OptimalJ

tool [OJ04] provides a practical example of this, where a UML model is transformed into an EJB specific model; changes to either the model or the code are reflected in the other.

Note that these types of model transformations are not necessarily mutually exclusive: one can, for example, have a bidirectional, multi-domain change propagating transformation. The terms defined above are used throughout the rest of this thesis.

2.4. Model transformations scope

In this chapter, most of the examples have been small and artificial, to aid exposition. Examples of larger model transformations abound – some are currently used by real users, some are in development, and some require more advanced technologies than are currently available. To give the reader a rough feel for the scope of the problem we are talking about, some representative examples are, in approximate ascending order of complexity:

- A simple model refactoring of the kind found in many Integrated Development IDE's such as Eclipse [Ecl04], where changing a methods name causes all references to that method to be renamed appropriately.
- Transforming a model that uses multiple inheritance into one that only uses single inheritance by creating intermediate interfaces [CEM+04].
- A data conversion transformation between two models whose meta-models are fairly similar in the aspects being transformed e.g. UML to BPEL [AGGI04].
- A model compiler that takes a UML model (e.g. class diagram and statecharts) and transforms it into a model for a specific programming language[3] e.g. Java.
- An abstracting transformation which operates between models held in two tools, one of which is an abstraction of the other, and which propagates changes between the two automatically.

Further examples can be found in e.g. [ACR+03, BDJ+03].

2.5. Change propagation

Consider the challenge of expressing change propagating transformations in a GPL. Referencing the model transformation method of section 2.2.4, one can see that the first novel aspect of change

[3]The underlying meta-model is likely to closely follow the abstract syntax for the language involved.

propagating transformations – the creation of appropriate tracing information – is relatively easy to perform in a GPL. In essence, whenever an element is created in the target model, an appropriate piece of tracing information is created relating the relevant source and target elements.

Consider now what happens when the source model involved in the transformation is altered. There are two immediate issues to consider when propagating changes from the source model to the target model. Firstly, should the entire transformation be rerun? Secondly, how do we propagate the relevant changes whilst maintaining any additions made by the user to the target model? The first issue can be considered to be a performance issue, and is thus not of great import. The second issue however covers a much more fundamental problem.

Recall that in the examples presented in section 2.2.1, standard object creation was used to populate the target model. This means that if such a transformation, or any part thereof, is rerun then entirely new model elements will be created from scratch, rather than existing elements being altered into their appropriate new form. To make this example concrete, recall the following function in the *ML1* to *ML2* transformation which transforms packages:

```
func transform_rule(element : M1.Package) : M2.Element:
  package_elements := []
  for package_element in element.elements:
    package_elements.append(self.transform_element(package_element))
  for parent in element.parents:
    for parent_element in parent.elements:
      package_elements.append(self.transform_element(parent_element))
  return new M2.Package(element.name, package_elements)
```

Assume that, following an initial transformation, we have added a new element to an *ML1* package, and have managed to identify that only the above function need be rerun in order that the corresponding *ML2* element is changed accordingly. When rerun in the form presented above, an entirely new package element will be created – thus two packages which represent the same thing will now be in existence. Schemes that, for example, delete all elements from previous iterations (so that when a change is propagated, all elements from the initial transformation are removed to avoid duplication) go some way to solving the problem, but also unveil another problem. This relates to the requirement that new elements manually created by the user in the target model be left unchanged when changes are propagated. Manually added elements may well have had links to or from elements created in previous transformation iterations. Simply replacing old model elements with new model elements destroys all links to and from the manually added elements. Instead transformations need to detect and update old model elements when appropriate.

In the case of the `transform_rule` function, a suitable change propagating equivalent may look along the lines of the following:

```
func transform_rule(element : ML1.Package) : M2.Element:
  package_elements := []
  for package_element in element.elements:
```

```
    package_elements.append(self.transform_element(package_element))
  for parent in element.parents:
    for parent_element in parent.elements:
      package_elements.append(self.transform_element(parent_element))
  existing_package = NULL
  for processed_element in self.processed_elements:
    if type(processed_element) == ML1.Package and
     processed_element.name == element.name:
      existing_package = processed_element
      break
  if existing_package != NULL:
    existing_package.name = element.name
    existing_package.elements = package_elements
  else:
    return new ML2.Package(element.name, package_elements)
```

The intention here is that the function first searches to find if an appropriate element exists in the target model and, if it does, that element is updated with the correct new information. If no such element exists, one is created. There are two problems with this particular approach. Firstly it relies on identifying equivalent elements in the source and target models by their name which, as detailed in section 2.2.3, is not a generally applicable strategy. The second problem is much more significant — the transformation has been significantly complicated by the addition of code to cope with change propagation. Separate branches are needed to deal with the creation of new elements, and the update of existing elements.

There is another potential solution for GPLs which support meta-classes (see Forman and Danforth [FD98]) where the object creation mechanism can be controlled by users. If all model elements are instances of a suitable meta-class, then instantiating a model element would require passing it a key as well as the values of the elements attributes. The meta-class can then check against a repository to see whether an element with the same key has already been created, and if so returns that element with its attributes updated appropriately rather than creating a new element; otherwise a new element is created. However this mechanism only works for model elements: sets, sequences and other built-in types present a serious problem if the user manually alters an instance of one in the target model. Furthermore since the meta-class mechanism is not available in many widely used OO languages (e.g. neither C++, Java or C# has such support), this mechanism can not be considered to be generally applicable. In such languages, cumbersome workarounds that avoid the standard object creation need to be employed.

This section has so far largely avoided a tackling an important practical element of many change propagating transformations: the generation of suitable keys. The concept of a key has hitherto been vaguely defined. In the context of change propagating transformations, elements in the target model have a key which is an identifier based in part on attributes from the source elements. In essence, given a particular set of source elements involved in a transformation, one should always be able to generate the same key. This becomes a complex affair in GPLs when e.g. multiple target

elements are generated from the same set of source elements in the presence of a loop — how does one generate unique keys. Furthermore since a target elements key is based on attributes from *all* source elements relevant to the target element, then this can complicate the program flow since this information potentially needs to be passed to all parts of a transformation rule. Section 6.1.4 explains the concept of keys in change propagating transformations in more detail, and also details alternative mechanisms to keys.

The issues noted in this subsection are severe enough that I am not aware of any published instances of change propagating model transformations written in a GPL.

Chapter 3.

Review

This chapter has two main parts. The first is a review of compile-time meta-programming systems. The second is a review of the major model transformation approaches proposed thus far (section 3.3.2).

Since terms related to programming language paradigms occur frequently in this chapter, I first present brief definitions of some relevant terms. I then give a brief overview of specification orientated approaches to model transformations; whilst these are of relatively little practical use, they were an important precursor to implementation orientated approaches. Finally the bulk of the chapter details several different model transformation approaches.

3.1. Programming language paradigms

When talking about different programming languages and model transformation approaches, I use certain terms in order to give the reader an impression of approximately where they lie in relation to each other. Since not all of these terms are likely to be known to all readers – and because some of these terms have various definitions attached to them – I present a brief explanation of the more contentious or unfamiliar here to clarify their later usage. Note that many of these terms are not mutually exclusive; indeed many of these terms can be applied in conjunction to certain languages.

Declarative / imperative As shall be seen in section 3.3.2, existing approaches to model transformations, and to programming languages in general, can be broadly categorized into two camps: those taking a *declarative* approach, and those opting for an *imperative* approach. The terms declarative and imperative can sometimes be rather contentious, and I use them with no small hesitation – they can also be rather crude mechanisms for pigeon holing different approaches. With that warning in mind, it is important to realize that in the wider context of programming languages there is a generally accepted consensus as to which of the two approaches best describes most languages. Crudely put, a language is considered to be imperative if it has side

effects and if it forces the programmer to be explicit about the sequence of steps to be taken when it is executed; languages that are side effect free and do not force the programmer to be explicit about the execution sequence are considered to be declarative. In essence, declarative languages allow the programmer to state the outcome of a computation without explicitly stating the steps necessary in order to achieve said outcome; in contrast, imperative languages force the programmer to state the steps of a computation which hopefully achieves the desired outcome.

Typically, functional languages such as Haskell [Jon03] and logic languages such as Prolog [SS94] are considered to be declarative, whilst languages such as Java [GJSB00], C++ [Str97] and Python [vR01] are considered to be imperative. Because there is a grey area in between these two terms, languages such as XSLT [W3C99b] – which is side-effect free and has an implicit approach to function call / pattern application, but is explicit in some aspects about the computation sequence – can be argued to conform to either paradigm. Wherever possible I try to suitably qualify these terms when talking about languages that can reasonably be considered to lie somewhere between the two paradigms. As this may suggest, it is often the case that the two paradigms to co-exist within the same environment.

Strongly / weakly typed Strongly typed languages are those where data have an intrinsic type which must be respected at all times. This is most easily explained by considering its inverse: a weakly typed language. For example C `kernighan88c` allows users to give an arbitrary type to any memory address. In C, one can incorrectly consider the data at a particular memory address to be of an incorrect type leading to bizarre errors. In a strongly typed language such an operation will cause an error.

Note that strong typing does not stipulate when the checking for type correctness may occur; it may be at compile-time or run-time.

Statically / dynamically typed Statically typed languages are those that enforce type correctness at compile-time. Haskell is an example of a strongly statically typed language, in that any type errors will result in a program which does not compile. Dynamically typed languages enforce type correctness at run-time. Python is a strongly dynamically typed language. Thus the expression `2 + "x"` will result in a compile-time error in Haskell, but a run-time error in Python. In both cases however the type of objects is respected.

Note that some languages combine aspects of static and dynamic typing. For example Java has partial static typing, but type casts force some type checks to be performed at run-time. Also note that static checking does not imply strong typing; C, for example, is a weakly statically

typed language.

Rule based Rule based languages allow the user to define multiple independent rules of the form `guard => action` i.e. 'if-then'. In most GPLs the process of execution is based on calling specific named functions in a sequence determined by the developer. This contrasts with rule based languages where a given piece of input data is checked against each rules' guard; the first rule whose guard matches then has its body called. Rule based languages thus use what is termed *forward chaining* as their fundamental execution method; once a guard is matched and an action is performed, the system does not revert back to a previous state nor does control flow backtrack to a previous point. Forward chaining of this sort is an inherently data-driven process, although note that the action part of a rule can be declarative or imperative in nature. Examples of rule based languages include ELAN [BKK+96] and XSLT.

Logic based Logic based languages are in some senses similar to rule based languages, in that they define a series of largely independent clauses (which are broken down into facts and rules) with the order that clauses are executed being determined by the languages engine and not the developer. The runtime strategy however is effectively to use a *backward chaining* strategy as opposed to forward chaining. Rule based languages start with a system state and try to continually apply rules to the system, often changing the system state in the process. Logic systems operate in the opposite fashion: they start with a goal, and attempt to prove that the system satisfies the goal, creating intermediate data as appropriate to satisfy this goal. This process effectively starts at the lowest level where it has known facts it can prove about the existing system, and works backward trying to prove new goals until the overall goal is satisfied. The canonical example of a logic language is Prolog.

Constraint solving Constraint solving involves the specification of multiple constraints that have variables which are quantified over infinite variables [Bar99]. The constraint solving algorithm then attempts to combine all constraints in a system to find acceptable solution(s). Sketchpad is the original example of a constraint solving system [Sut63].

Constraint logic based A relevant variation on logic based programming is Constraint Logic Programming (CLP). Essentially this involves the merger of the constraint solving and logic programming paradigms into one. CLP overcomes two particular problems often associated with standard logic programming [FHK+92]. Most significantly, CLP allows data to be interpreted; in other words new datatypes can be created. By providing different constraint solving mechanisms, CLP also allows users to sidestep the often significant performance issues associated

with the general purpose depth-first search rule of logic programming. Interestingly, CLP systems are often modified logic systems. See e.g. [JM94] for more details.

3.2. Compile-time meta-programming

Compile-time meta-programming allows the user of a programming language a mechanism to interact with the compiler to allow the construction of arbitrary program fragments by user code. As Steele argues, 'a main goal in designing a language should be to plan for growth' [Ste99] – compile-time meta-programming is a powerful mechanism for allowing a language to be grown in ways limited only by a users imagination. For example, it allows users to add new features to a language [SeABP99], or apply application specific optimizations [SCK03].

In the following subsections I review material relevant to compile-time meta-programming.

3.2.1. Token level macro facilities

By far the most common programming language macro facility in use today is the C PreProcessor (CPP). In their comprehensive analysis of C preprocessor usage, Ernst et. al note that although the CPP is not a fundamental part of the language 'C ... is incomplete without its macro processor' [EBN02]. The CPP operates as a pre-compilation stage that expands macros and allows conditional compilation before the C compiler itself is executed. Generally the separate existence of these two stages is transparent to the user, although typically either stage can be individually invoked.

The CPP operates at the token level, sharing its tokenizing strategy with the C language. Macro definitions consist of a name, arguments and a body and are only permitted to occupy one logical line in the source file. Once introduced, subsequent tokens which match the name of a macro are automatically replaced by the body of the macro with suitable argument replacement.

Because the CPP is entirely ignorant of the syntactic context it is operating in, one can quickly run into unexpected situations. The need to develop and use conventions is paramount to avoid serious problems such as variable capture, and unexpected macro replacement. Variable capture is a particularly insidious problem, which is most often noticed when a macro expands to manipulate a particular named variable; at such a point, if the user passes in a variable of the same name as an argument to the macro then the two clash and unexpected results arise (see Dybvig et. al [DHB92] for an in depth examination of this problem).

In common with most token level macro facilities (e.g. the Unix M4 macro processor), the CPP provides useful features but at a cost: its use must be carefully controlled to prevent unexpected side effects. This is largely due to the fact that most such facilities operate with limited knowledge (CPP)

or total ignorance (M4) of the syntactic environment in which they are operating. Although the CPP is widely used, it is well-known for causing bizarre programming headaches due to unexpected side effects of its use (see e.g. [CMA93, Baw99, EBN02]).

3.2.2. Syntax level macro facilities

The LISP family of languages, such as Scheme [KCR98], have long had powerful macro facilities allowing program fragments to be built up at compile-time. Such macro facilities suffered for many years from the problem of variable capture; fortunately modern implementations of hygienic macros [DHB92] allow macros to be used safely. LISP and Scheme programs make frequent use of macros, which are an integral and vital feature of the language.

Brabrand and Schwartzbach differentiate between two main categories of macros [BS00]: those which operate at the syntactic level and those which operate at the lexing level. Scheme's macro system works at the syntactic level: it operates on Abstract Syntax Trees (AST's), which structure a programs representation in a way that facilitates making sophisticated decisions based on a nodes context within the tree. Macro systems operating at the lexing level are inherently less powerful, since they essentially operate on a text string, and have little to no sense of context.

The macro language provided by Scheme is powerful and (unlike many traditional LISP implementations) reliable. However it has spawned few imitators. Although one could suggest many reasons for this, perhaps the most crucial is related to the fact that few other languages share Scheme and LISP's highly regular, sparse syntax; a LISP grammar is many times smaller than that of any programming language in wide spread use today. In no small part due to this, LISP is able to use expressions themselves as a data structure *s-expressions* (or s-exp for short). In other words, this means that a macro call is a simple operation that first of all provides substitution in one s-exp (the macro) and then splices the resulting s-exp into another s-exp (the macro caller). In general, manipulating a language with a complex grammar in such a way is far more difficult. As Weise and Crew note, such attempts generally lead to heavily convoluted and hard to maintain code that has to manipulate and create ASTs [WC93].

Weise and Crew propose a new style of macro language (implemented for C) where macros are C-like functions, with added syntax for macro related facilities, which take in and produce ASTs. A special operator allows abstract syntax fragments to be expressed in the standard C concrete syntax, rather than relying upon the explicit creation of an AST via procedure calls. Inside the fragments limited variable replacement can be made by using the `$` operator to refer to variables outside of the fragment. In so doing, Weise and Crew are able to provide hygienic macros for a language with a relatively complex grammar in a relatively natural manner. However their solution is somewhat

hampered by the fact that the macros themselves require considerable added syntax over the base language C, and that because the macro solution is incomplete, certain types of macro are impossible to express.

3.2.3. MetaML and Template Haskell

Despite the power of syntactic macro systems, and the wide-spread usage of the CPP, relatively few programming languages other than LISP and C explicitly incorporate such systems (of course, a lexing system such as the CPP can be used with other text files that share the same lexing rules). One of the reasons for the lack of macro systems in programming languages is that whilst lexing systems are recognised as being inadequate, modern languages do not share LISP's syntactic minimalism. This creates a significant barrier to creating a system which matches LISP's power and seamless integration with the host language [BP99].

Relatively recently languages such as the multi-staged MetaML [Tah99] and TH [SJ02] have shown that statically typed functional languages can house powerful compile-time meta-programming facilities where the run-time and compile-time languages are one and the same. Whereas lexing macro systems typically introduce an entirely new language to proceedings, and LISP macro systems need the compiler to recognise that macro definitions are different from normal functions, languages such as TH move the macro recognition burden from the point of definition to the macro call point. In so doing, macros suddenly become as any other function within the host language, making this form of compile-time meta-programming in some way distinct from more traditional macro systems. Importantly these languages also provide powerful, but usable, ways of coping with the syntactic richness of modern languages.

MetaML was the first proposal to show how a modern language can incorporate powerful compile-time meta-programming facilities. MetaML is a multi-stage language; that is, it can generate and compile arbitrary program fragments even at run-time. In MetaML, macros are normal functions that are indistinguishable from any other and hence are first class, unlike Scheme macros (although Bawden has proposed a first-class macro system for Scheme [Baw00]). The use of a macro however requires an explicit 'splice' operator that evaluates its arguments at compile time and inserts the results into the AST. Since a goal of MetaML is to ensure the type correctness not only of program generators, but also generated programs themselves, the language is severely restricted. For example MetaML can not, as standard, introduce entirely new definitions (although Ganz *et. al* propose a solution for this [GST01]). TH takes the most important aspects of MetaML – quasi-quoting and splicing – and refines them in the context of Haskell. TH is a two-stage language in that it can only generate and compile program fragments at compile-time. TH integrates its features more tightly

into the host language, and places less restrictions on the subsequent generated programs. However Template Haskell does contain one obvious limitation: macro definitions and macro calls must exist in different modules[1].

3.2.4. OO languages

Few OO languages have any form of macro facility. The dynamically typed OO language Dylan has a macro facility which is similar to Scheme's [BP99]. However Dylan's macro language is very different from the main language itself, leading to a very obvious seam between the two. The statically typed OO language Nemerle has a compile-time meta-programming system that is partially inspired by TH [SMO04]. Nemerle is also capable of a limited form of syntax extension. Nemerle's system is unusual in that it is partly homogeneous (normal Nemerle functions can be called at compile-time) and partly heterogeneous (in that top-level macro functions must be explicitly identified).

3.3. Model transformations

3.3.1. Transformation specifications

Two of the first works in the area of model transformations are that of Lano [LB98] and Evans [Eva98] who both define transformations with respect to an underlying semantics of class diagrams. The transformations they define are not directly executable, rather they specify a transformation. In essence this means that given two particular model instances the specification can determine the well-formedness of the two models with respect to one another. Later work such as that of the 2U group [CEK01] and Akehurst and Kent [AK02] refine the use of class diagrams and OCL for transformation specifications.

Although transformation specifications have many uses [QVT03a], they are of limited relevance in this thesis's context of transformation implementations. However it is important to note that while transformation specifications are often said not to be executable [QVT03a], this is slightly misleading. Specifications can provide a 'yes' / 'no' answer about the well-formedness of a given pair of models. In advanced cases, a specification may even be able to provide a detailed analysis of why two particular models are not well formed with respect to each other.

[1] This is largely an implementation restriction due to the fact that the existing TH implementation forces staged execution on an engine not originally intended for such a purpose.

3.3.2. Transformation technologies

In the following subsections I review the major technologies for expressing model transformations. Some of these technologies were specifically designed for writing model transformations; others have been adapted to suit this purpose. The technologies in this section are listed in approximate order from those technologies least specialized for model transformations to those most specialized.

3.3.3. XSLT

XSLT [W3C99b] is a rule based XML transformation technology which has gained a significant amount of attention over the past few years. XSLT initially seems a promising candidate in which to realise model transformations, because models are often stored as XML in order to interchange between tools (via the XMI standard [OMG03]). An XSLT rule takes the form of a simple pattern written in the XPath language [W3C99a] and a body which is an unusual mix of explicit and implicit sequencing. XSLT is also unusual in that both the data to be transformed and the transformation itself are represented in the same form – XML.

Peltier et al. [PBG01] based a model transformation framework upon XSLT but used it only at the lowest-level, citing general readability issues as well as specifics such as the lack of acceptable error reporting. As this experience suggests, XSLT suffers from a number of flaws which render it unsuitable for the majority of medium or large tasks that we are interested in. As noted in Bex et al. [BMN02], 'XSLT is highly adequate for the simple transformations it was intended for (recall that XSL was originally intended just for XML to HTML transformations)' but that it has serious shortcomings for more advanced transformations. One of the problems alluded to by Bex et al. about XSLT is its lack of power; it took several years before a formal proof was constructed that XSLT is Turing complete [Kep02][2] and – as both the relatively recent timing and need for existence of the proof may suggest – in practical usages one very often rapidly hits the limits as to the sorts of transformations XSLT can naturally express.

Because XSLT transformations are written in XML, they have to conform to both XML's syntax and XML's rigid well-formedness rules. XML's syntax is rather verbose compared to most programming languages. Whilst XML's well-formedness rules go some way to ensure that XML data has been correctly represented, they also force XSLT transformations to be somewhat more wordy than would otherwise be necessary. Because XSLT transformations must be well-formed XML files, there are also some seemingly valid transformations involving XML fragments that are in fact invalid because the XSLT transformation can only naturally deal with well-formed XML fragments. Ill-formed

[2] See `http://www.unidex.com/turing/utm.htm` for the Turing machine implementation the proof is based upon.

fragments, such as those which do not contain balanced elements, must be encoded using the `CDATA` mechanism thus circumventing much of XSLT's syntactic conventions. The net effect of XSLT's syntax is to cause the 'poor readability and the high cost of maintenance' noted by Peltier *et. al*.

A separate issue which makes expressing model transformations in XSLT less than ideal is that XML documents are represented as a tree structure; models are, in the general case, naturally representable as graphs. Although graphs can be represented by trees with link references between nodes, the difference in representation can lead to an unnatural representation of many types of model transformations [VP03, Var03]. To compound this issue, XSLT provides relatively poor support for references, making the following of references a heavyweight exercise that further clutters model transformations.

Of particular relevance to this thesis is the fact that XSLT transformations are inherently unidirectional and stateless. Furthermore when compared to the method of section 2.2.4, one can see that XSLT comes off poorly by virtue of the fact that it not even capable of creating tracing information relating source and target elements.

3.3.4. Graph transformations

A particular style of transformation which has seen heavy use in theoretical circles since their introduction in the late 60's are graph transformations; see [AEH+99] for a relevant, comprehensive overview of this area. Note that the term 'graph transformation' is misleading, as it refers to a particular category of rule-based transformation that is typically represented diagrammatically. Various other types of transformations operate on graphs but are not termed 'graph transformations' – Mens and van Gorp note that 'graph transformation is more a programming paradigm than a technique' [MG04].

Well known styles of graph transformations include the single and double push-out approaches, though there are several others. Graph transformation approaches, viewed at a suitably high level, operate in a similar fashion. Essentially the input graph is gradually transformed in-place into the output graph; rules identify subgraphs to transform, and then glue in a replacement graph. Rules are successively applied to the changing model until no more apply. Factors such as the handling of dangling references during replacement, and the order in which rules are tried differentiate various approaches. Graph transformations have a number of useful theoretical properties which make them attractive and, in the context of this thesis, the fact that models are well represented as graphs is particularly appealing [VP03].

Early work involving graph transformation and models largely centred on their use in defining the semantics of different modelling diagram types. In the continuing work of Gogolla *et. al* [GPP98,

Gog00, GZK03], graph transformations are used to transform UML models into instances of a 'core' UML, using the precise semantics of this core to define the semantics of the rest of the language. Although an intriguing approach, Gogolla's transformations tackle only small-scale problems.

Typed attributed graphs are a style of graph well suited to representing models and for reasoning about properties such as termination and confluence [HKT02]. Küster et al. [KHE03] define a general model transformation approach using graph transformation as the underlying mechanism, allowing them to draw upon some of the properties of graph transformations in a model transformation context. By grouping transformation rules into transformation units, it is reasonable to expect that such an approach will scale to larger problems than Gogolla's approach, but to date only small scale examples appear to have been attempted.

Levendovszkey et al. [LKM+02], Sendall [Sen03], Varró [VVP02, Var03] and Willink [Wil03] have all proposed model transformation approaches which are based upon simple graph transformation systems. Agrawal *et. al*'s more mature *GReAT* system [AKS03] is in a similar vein. These approaches all share in common that they define a visual language for defining unidirectional stateless transformations.

Change propagation

None of the graph transformation approaches mentioned thus far in this section has been capable of any form of change propagation. There are other instances in the literature relating to graph transformations and change propagation which I now describe.

In Braun and Marschall's language [BM03] the 'B' stands for 'bidirectional', although this appears to be a recent change of direction that is not yet fully realised – BOTL originally stood for 'Basic Object-oriented Transformation Language' [BM02]. Braun and Marschall present a small amount of theory intended to facilitate bidirectional transformations, but choose to restrict the transformations they consider to bijective transformations. A bijective relation is one that is injective and surjective. Informally, an injective relation means that distinct source objects must map to distinct target objects (commonly known as 'one-to-one'). A surjective relation means that each source object must map to a target object (commonly know as 'onto'). Looking back at even the simple examples of section 2.2.1, one can see that many useful model transformations fail to satisfy one or both of the injective and surjective criteria. Even if the BOTL approach were to be fully fleshed out, the fact that it is fundamentally incapable of expressing many simple model transformations severely limits its utility.

Triple graph grammars [Sch94] are a formalism specifically designed to facilitate bidirectional transformations. Several approaches reference triple graph grammars but, to the best of my knowledge, none have yet used this as the underlying formalism. This may be in part due to the fact that

triple graph grammar rules are considerably more difficult to create and comprehend than normal pair grammar rules, since they encode productions and correspondence rules in one. Kindler *et. al* outline a possible implementation of triple graph grammars and model transformations [KRW04], but the approach has yet to be realised. Becker *et. al* [BHW04] present a model transformation scheme which integrates some limited aspects of triple graph grammars into an approach that otherwise shares more in common with unidirectional stateless graph transformation approaches. With this they are able to perform some limited change propagation, although their scheme requires frequent manual intervention on the part of the user to resolve conflicts.

Rule organization and control structures

Most of the graph transformation approaches detailed in this section give little or no attention to facilities for organizing rules or control structures. These two points are connected in a way that may not be obvious. Since most approaches lack appropriate control structures, one often needs to copy rules making subtle modifications to get the same effect as if control structures were present. The proliferation of rules is then aggravated by the lack of facilities for organizing rules.

The lack of such features in research prototypes is perhaps not surprising. However although there are some suggestions for enhancing such facilities in graph transformation systems (for example, [SW98] defines an organization mechanism based on UML packages), even mature graph transformation systems such as PROGRES [SWZ99] have surprisingly weak organization facilities and control structures [MG04]. It is unclear whether this reflects a fundamental problem in the methodology, or merely a lack of development effort into practical matters.

Formal properties

Although graph transformations can be used to prove interesting properties about transformations, only a fairly small minority of useful transformations are currently amenable to such analysis [MDJ02]. In practise the formal properties of graph transformations are of relatively little use, and can not be considered to be a significant advantage over other approaches which do not make similar claims.

Conclusion

The sheer number of model transformation approaches based on graph transformations suggests that they hold promise for realizing model transformations. However current approaches are limited in scope and rather simplistic. Despite being a well established subject area the relative paucity of graph transformation implementations is surprising. Perhaps the two best known systems are PROGRES and FUJABA [NNZ00]. The former is a venerable and generalized candidate with roots stretching

back to nearly two decades; however it is very much a research vehicle and has seen relatively little development in recent years. The latter is more modern but is specialized for certain restricted styles of Java development.

Some explanation for the lack of tool support can perhaps be gleaned from the literature on graph transformation based model transformations. Men and van Gorp comment that 'it remains to be seen whether graph transformation alone suffices to express complex transformations' [MG04]. Czarnecki *et. al* [**?**] note that users often perceive graph transformations to be complex beasts, hence why they have seen relatively little real-world usage.

In conclusion, graph transformations have yet to show that they are a practical vehicle for model transformations. Furthermore, the fact that no current approach provides anything other than rudimentary support for change propagation reduces their relevance to this thesis.

3.3.5. Logic programming

An approach unique in the model transformation world is that described by Whittle [Whi02] who uses the rewriting logic based language Maude [CEL$^+$96]. Although the prime motivation of the approach is to automate simple unidirectional stateless transformations on simplified UML class diagrams, the concept of *difference matching* is introduced. The example given is the checking of a model D_2 as a valid refactoring of a model D_1. Differences between the two models are discovered, and transformation rules are invoked in order to reduce these differences; if the repeated application of rules reduces the differences between the models to the empty set, then the models are correct with respect to the transformation. Although difference matching is partly intended to alleviate the logical problem of instantiation of unbound variables, the concept could usefully be applied to non-logic based system. It should be noted that this concept is considerably different than the transformation specification approaches detailed in section 3.3.1. Furthermore the unusual nature of this feature means that it does not neatly fit into the method of section 2.2.4.

3.3.6. TXL

Cordy's TXL [Cor04] is a particularly interesting transformation language. Although originally intended for transforming instances of the programming language Turing, it has evolved into a language capable of transforming instances of arbitrary language grammars. In so doing, TXL has morphed into a hybrid rule-based / functional programming languages. Whilst rules can still be fired in a traditional rule-based manner, rules can also call specific other named rules. If the guard of a named call does not match then the source model is returned unmodified. TXL rules contain powerful guards, which consist of a relatively crude pattern augmented by a *when* clause containing an arbitrary ex-

pression. As its origins might suggest, TXL is largely geared towards transforming one programming instance into another instance of the same programming language. It is possible to transform between two different programming languages via a 'union grammar' which is a single grammar combining all relevant aspects of the grammars of the two languages in question.

Paige and Radjenovic performed some initial investigations of the feasibility of using TXL for model transformations [PR03]. They provide a small example of a transformation between simplified models of UML and Java. The example is carefully constructed so that a single grammar is sufficient to express both models, and thus the transformation is relatively simple.

TXL has many advantages as a transformation system. It is mature, efficient, and is demonstrably capable of succinctly expressing many useful transformations. It is one of the few transformation systems to have been used to process large volumes (Cordy records a case where several billion lines of code were transformed with TXL), and its pragmatic approach to rule-based transformations incorporating functional aspects is far more refined than any of the dedicated model transformation approaches reviewed in this chapter. Offset against its advantages are two significant issues relevant to model transformations. Firstly, TXL's support for transforming between two different languages is poor, relying on the artificial concept of union grammars. Such grammars require significant manual effort to create, and allow the transformation writer to create hybrid models which satisfy the union grammar but which conform to neither of the original languages; although this may on occasion be useful, it also opens up many possibilities for generating syntactically invalid output. Secondly, TXL is an inherently unidirectional stateless transformation mechanism; in terms of the method of section 2.2.4, it is similar to XSLT.

3.3.7. QVT

Model transformations are a vital factor in the realization of the OMG's MDA vision [BG02], which is based on the idea of progressively facilitating more and more software development with models. Since models appear in different forms at different stages of the MDA vision, the concept of a model transformation is key within MDA. For example: transforming models representing one technology into others which represent different technologies; abstracting and refining models; merging models; and so on. To this end a Request For Proposals (RFP) was issued by the OMG 'MOF 2.0 Queries / Views / Transformations (QVT)' [OMG02] in 2002 to seek a standard way of performing model transformations.

There were eight initial submissions to the RFP. When this thesis was in its early stages of writing, seven submissions remained on the table. Since that point many of the submissions have attempted to merge – this process is ongoing at the time of writing. None of the original submissions have yet been

'taken off the table', and since they cover a broad spectrum of solutions, with several indicative of the state of the art, an analysis of them is still highly relevant. However it is not my intention to enumerate all of the individual submissions; see Gardner et al. [GGKH03] in particular for a comparison of the individual submissions, and also Czarnecki and Helsen [**?**] who propose a feature classification scheme for transformation approaches, including several QVT submissions. In the following sections I use three QVT submissions – TRL, xMOF and the QVT-Partners submission – to examine some interesting points in the model transformation spectrum. The first two of these three submissions are bi-polar opposites; the third lies somewhere between the first two.

3.3.8. TRL

The Transformation Rule Language (TRL) language [OQV03] is in essence a standard rule-based imperative language specialized with features for dealing with UML model transformations. This specialization comes in several forms: concepts such as 'transformation rule' are raised to first-class status, meaning they do not need to be encoded using standard language constructs; some of the information recorded in the new first class constructs is used for additional purposes e.g. to create tracing information; extra syntax is provided for e.g. accessing the stereotype of a UML model element. Rules consist of a signature – comprising the types of the source and target model elements – and an imperative body. The syntax and semantics are essentially that of the Object Constraint Language (OCL) [OMG97] augmented with side-effects and a small handful of necessary control constructs. The benefit of this approach is its relative familiarity to users, and the knowledge that imperative programming languages traditionally lead to efficient implementations. However this argument is slightly dented by the relatively unusual concrete syntax and semantics which result from adding side-effects to a side-effect free constraint language.

Whilst TRL is adequate for specifying small transformations, it has several flaws which become apparent when attempting more complex tasks. For example, it suffers from many of the problems associated with writing model transformations in GPLs; this is compounded by the surprising realization that despite initial appearances TRL is not an OO language. Additionally, despite having several language constructs specifically designed to aid inspecting and manipulating models, TRL implicitly adopts a fixed meta-level system – representing models that are not of the type originally envisioned is difficult. Fundamentally TRL is only capable of expressing unidirectional stateless transformations – whilst tracing information can be automatically created from rules, the fundamentally imperative nature of the majority of the language and use of explicit object creation rules out practical change propagation.

TRL can thus be seen to be a fairly standard non-OO rule-based imperative language, augmented

with a few unusual features for its intended domain. Due to the very coarse grained nature of the rule signature mechanism, TRL relies heavily on the use of OCL constraints to detect elements of interest. In short, TRL is adequate for a certain style of limited model transformation, but its addition of very specialized model features coupled with a paucity of standard control and data mechanisms means it is not a practical vehicle for complex model transformations.

3.3.9. xMOF

The xMOF[3] language [CS03] is a constraint solving system for model transformations. An xMOF program consists of a number of OCL constraints about model elements involved in a transformation, with the aim of specifying bidirectional change propagating transformations. The xMOF engine then attempts to ensure that all models in the system satisfy the constraints. As far as the constraint writer is concerned there is thus no practical difference between an initial transformation and subsequent change propagation.

The chief advantage of the xMOF approach is seen to be that it rids the transformation designer of the need to perform the tedious and verbose book-keeping demanded by imperative approaches. Furthermore since the relationship between two models is not stated in terms of inputs and outputs, the transformation is implicitly bi-directional.

xMOF has some features which are of particular interest. Chief amongst these is its powerful but intuitive solution to the potential paradox inherent in bidirectional change propagating transformations – that is, if, after an initial transformation, one model is changed, in which directions should changes be propagated? A simplistic system might notice the difference between the two models and propagate changes from the unchanged model, thereby wiping out the changes made to the other model. A more complex system would be for the system to record which elements have been changed in which model and use that to determine the direction in which changes should be propagated. However unfettered change propagation is not always desirable and is not always possible in cases in which an infinite number of possibilities might satisfy some relations between models (for an analogy, consider what happens if, given specific values of x and z, one varies the value of y in the equation $x + y = z$ – there are an infinite number of pairs (x, z) which will satisfy the equation). xMOF therefore allows developers to specify the direction of equality in the presence of change propagation. A simple example of this is as follows. Consider the xMOF statement `name :== name` which means that the `name` attribute of the lhs and rhs models should be the same; if they differ, the name of the lhs should be used to update the rhs. The opposite effect can be achieved by `name ==: name`. Al-

[3] xMOF uses a number of terms in ways that conflict with generally accepted definitions; in the interests of simplicity I largely ignore the particular terms xMOF uses.

though xMOF requires such functionality for bidirectional change propagation, one can see that such a feature could be of use even in unidirectional change propagating transformations.

However xMOF has several severe disadvantages from a practical point of view. First and foremost, it places a burden on the user to make sure that the constraints they specify completely describe the transformation – if they fail to do this, the resulting system is likely to either produce arbitrary results each time it is run, or to run out of memory as it attempts to enumerate all matching values. Although this complaint can be levelled against any constraint solving system, it is arguably more critical in a modelling environment which contains richer and more complex datatypes than many similar systems. There are therefore a large number of different ways that a user can under specify their transformation; it is unclear that xMOF can report enough instances of this in advance to the user to significantly lessen the problem. A similar issue can be seen when considering transformation composition; although xMOF allows arbitrary transformations to be composed together, it places no restrictions on how constraints between different transformations interact. Since individual rules typically interact together in complex ways, adding another set of rules via composition can easily generate unexpected results.

A secondary problem with xMOF is that, perhaps surprisingly, its specification is entirely non-committal about execution strategies. Whilst this gives implementers scope for concentrating on aspects important to specific audiences, it also places a heavy burden on each implementer to develop the sophisticated inference rules necessary for analysing groups of constraints, and for producing an engine capable of finding solutions which satisfy all of them. This has a subtle knock-on effect for end users: since different execution engines will have different inference rules and so on, sets of constraints that may execute as expected in one xMOF system may not do so in another. However, at the time of writing, this can be considered to be a minor issue since there is currently only one xMOF implementation available.

Perhaps the most intractable issue with xMOF is that, by its very nature, and even with perfectly specified systems, it can take an unbounded amount of time to execute. Particularly in the case of large models, it is unclear that a solution of this type will execute in a reasonable time. Constraint programming, as detailed by e.g. Barták [Bar99] is a challenging and relatively unexplored area of research (despite existing for over four decades [Sut63]), which has shown potential in small, tightly defined areas, but there is little precedent for using it on a task of the order of complexity of model transformations. Conceptually however xMOF is interesting because it satisfies all parts of the method of section 2.2.4.

3.3.10. QVT-Partners approach

Having detailed two approaches at opposite ends of the model transformation spectrum in the preceding two sections, it is interesting to evaluate the QVT-Partners submission [QVT03a] as it can be seen as lying between the other two submissions. This section comes with a bias warning: it should be noted that the author was a major contributor to this submission.

The QVT-Partners approach makes a distinction between transformation specifications and implementations, providing support for both. Specifications can check whether two models are correct according to the specification. Implementations actually transform a given model into another. In the general case, specifications are not executable in the sense that they are capable of transforming one model into another; however the QVT-Partners approach makes the case that certain types of small specification can automatically be refined into implementations. The specification aspect of the approach shares much in common with the wholly specification orientated approaches noted in section 3.3.1.

The overall framework is a rule-based one, with a limited form of backtracking occurring when transformations are composed together. Transformations are formed of a number of domains, each domain being formed of a pattern and a constraint. Patterns are succinct ways of expressing powerful constraints about models and can be arbitrarily nested and composed. Patterns can contain unbound variables which are effectively wildcards that are assigned the value of whatever they match against. A transformation can contain an overarching OCL constraint which is able to tie together variables over multiple patterns.

Transformations can be composed together using three operators `disjunct`, `conjunct` and `not`. Composition can be used in two different fashions: a new transformation can be the composition of one or more other transformations; transformation implementations can utilise a limited form of composition in their expressions. In order that specific rules can be composed, rules have names which allow individual rules to be explicitly called. Both diagrammatic and textual notations are defined; the diagrammatic notation conveys less information and all but the most simple transformations make at least some use of the textual notation.

The QVT-Partners approach is interesting in several ways. By making heavy use of patterns, it is often able to express complex transformations succinctly and in a manner which is reasonably comprehensible. Its use of composition with limited backtracking allows complex transformations to be built which are still likely to run in a reasonable time.

However the approach is not without its limitations. For example the rule-application mechanism is ill-defined and confusing. Transformation composition is marred somewhat by the default semantics of *conjunct* which automatically and arbitrarily merge some model elements returned by

the composed transformation together leading to unexpected and confusing results. The pattern language provided is relatively threadbare, lacking vital features, and its definition ambiguous in several important areas. Significantly, there is no support provided at present for facilitating change propagating transformations. However there have also been a number of follow-up publications to the QVT-Partners submission that explore and elucidate various other areas around the submission e.g. [ACR+03].

3.3.11. Other approaches

DSTC approach

The CLP based model transformation approach of the DSTC QVT submission [DIC03] is in essence a Prolog-like language highly specialized for model transformations. Despite some superficial similarities to xMOF, the two approaches are in fact rather different – the DSTC approach is far more explicit than xMOF about many aspects of transformations thus removing many of the possibilities for users to write transformations that can not be executed. For example, transformations in the DSTC approach are inherently unidirectional. Transformation rules can create tracing (called tracking in the DSTC terminology) information but require the user to explicitly specify what elements must be involved in each trace. A concept of a 'key' is also defined which is a way of uniquely defining an object based on certain of the objects' properties. This concept is largely unused in the current approach; one can surmise that it is intended as part of a strategy to enable change propagation. Although the DSTC submission makes clear that its choice of a declarative approach is to enable change propagation – and despite the presence of relevant features such as tracking and keys – it is currently only capable of specifying unidirectional stateless transformations.

ATL

The ATL language [BDJ+03] takes a rule-based approach broadly similar to the QVT-Partners approach, albeit with significantly enhanced tool support. It does not possess a very specialised pattern language, relying chiefly on a slightly augmented version of OCL. Unlike the QVT-Partners approach, which places a strong emphasis on the imperative aspects of the solution, ATL downplays the non-declarative aspects of the language. Currently the approach only supports unidirectional stateless transformations.

Johann and Egyed's approach

Johann and Egyed briefly outline a framework for unidirectional change propagation model transformation approach [JE04]. As shall be seen in section 6.3.1, their proposed solution is only capable of propagating certain limited forms of changes. However their approach is notable for being the only unidirectional change propagating approach documented in this section.

3.3.12. Summary of model transformation approaches

Several interesting points can be taken away from the review of existing model transformation approaches:

1. Approaches are either essentially variant GPLs (e.g. TRL) or logic-based (e.g. xMOF). The QVT-Partners approach is unusual in that it has aspects of both styles, but it can be argued that it is really two different approaches under one umbrella.

2. Most approaches contain some discussion of change propagation. However with one or two exceptions (notably xMOF) very few approaches actually present any concrete material as to how they might support change propagating model transformations in practise.

3. Many approaches either lack a publicly available implementation, or have an implementation that only implements a subset of what is documented.

A corollary of the first two points is that the existing approaches only explore a handful of points within the overall solution space.

Many of the current model transformation approaches can be categorised as declarative. Those that are categorised as imperative (e.g. the QVT-Partners approach and TRL) are of limited use because they share the same issues as GPLs, as noted in section 2.5. At this point, it is important to note that although this implies – as do the majority model transformation documents – that only a fully declarative solution is capable of providing a practical solution to change propagating model transformations, this is not in fact the case. In fact, whilst several of the issues noted in section 2.5 are irksome when expressing change propagating transformations, explicit object creation is the only issue which fundamentally limits change propagation.

Although explicit object creation is the default in GPLs, there is no inherent reason why imperative approaches *have* to use explicit object creation. If the relationship between elements in the source and target models is specified declaratively then the details of the computation which leads to the relationship is irrelevant – the computation can be declarative or imperative. However currently only

solutions which have a fully declarative computation are able to specify the relationship between source and target elements sufficiently.

An interesting point to note is that whilst many model transformation approaches claim to be designed with change propagation in mind few of them, at the time of writing, have any practical support for such model transformations.

3.4. Research problem

Having in this chapter reviewed the major approaches to compile-time meta-programming and to model transformations, this section identifies two research problems which are tackled in the remainder of this thesis.

3.4.1. A DSL implementation technology

As noted by Hudak, implementing DSLs as stand alone applications is time-consuming [Hud98]. In response to this, Hudak proposes Domain Specific Embedded Languages (DSEL's), which take a different approach, embedding a DSL inside a larger, richer language. Unfortunately the style of DSEL that Hudak promotes is quite limited in nature. His approach relies on the built-in features of functional languages such as Haskell: higher-order functions, lazy evaluation and so on. Whilst these features can ease the expression of many DSELs, there is a limit to how much one can express without descending into cumbersome encodings. The fundamental limitation of Hudak's approach is that he expressly forbids any form of syntax extension to his host languages. Conversely Wilson argues that programming languages need to allow their syntaxes to be extended if powerful DSLs are to be exploited to their maximum potential [Wil05].

There exist other approaches to embedding DSLs within host languages in a fashion that permits syntax extension. LISP and Nemerle provide limited forms of syntax extension and were discussed in section 3.2. Bravenboer and Visser perhaps come closest to the ideal vision of syntax extension with the `MetaBorg` system which allows language grammars to be extended in an arbitrary fashion [BV04]. `MetaBorg` is a heterogeneous system in that the language being extended is generally different than the language doing the extension. Thus the latter must provide facilities ranging from parsers and parse tree datatypes to emulations of aspects of the extended languages compiler in order to present a system which can compete with LISP-esque macros for power. In order to use such a system, the person implementing the extension will need to be expert in three entirely separate systems (the language being extended, the language doing the extension and the 'emulation' of the language being extended) in order to produce a quality implementation. Because of this, the `MetaBorg` ap-

proach currently seems best suited for embedding DSLs that are small and localised in nature.

The research problem tackled by this thesis is thus to provide a programming language which allows its syntax to be extended in order to facilitate DSL development. Both Wilson, and Bravenboer and Visser note that no modern programming language contains sufficient facilities 'as is' to achieve this aim. Chapter 4 thus details my solution to this problem: a new imperative programming language named Converge, which supports syntax extension with its compile-time meta-programming facilities.

3.4.2. Issues with existing model transformation approaches

Although, as shown in section 3.3.12, there are many detailed points one can pick out from the analysis of existing model transformation approaches, two higher-level points in particular have relevance to the direction of this thesis. The first is fairly easily deduced:

1. Despite superficial differences, most existing model transformation approaches are relatively similar to one another, and are also largely simplistic in their approaches. For example, with the notable exception of xMOF, all existing approaches are only capable of expressing stateless model transformations.

Put in different terms, since model transformations are a relatively recent development there is little collective knowledge about even the most basic of building blocks. Unsurprisingly therefore, all existing approaches are therefore somewhat limited and simplistic in nature. This suggests that rather than expecting a new model transformation approach to present a complete, unified solution it is more important to focus on attempting novel approaches to even simple issues. In this way one would hope that in time the best solutions for various aspects of model transformations will be identified.

The second point has not, to the best of my knowledge, been directly articulated in the context of model transformations but is a well known issue in similar areas:

2. The relative expense and time necessary to implement a practical model transformation approach inhibits experimentation.

Hudak documents this issue in a more general, but highly applicable, context [Hud98]. Hudak highlights an unfortunate tendency that one can also see in model transformation approaches – as they grow in complexity they tend to acquire more and more features influenced by normal programming languages. This not only adds to the implementation burden, but the programming language-esque features tend to be inferior to their counterparts. This tendency can be seen in its most extreme form in TRL.

It is my contention that the two points here are closely linked. The difficulty of implementing model transformation approaches is one of the chief reasons behind the lack of exploration of different techniques and approaches, and hence the relatively simple and uniform approaches that are currently available. Since model transformations are an inherently practical topic, implementations are vital for assessing and evolving new ideas. A long and labour intensive idea-implement-assess cycle seriously inhibits such experimentation. Model transformations are thus an excellent candidate for implementation as DSLs within a programming language with an extendable syntax.

Specific model transformation approaches

As noted in section 3.3.12, although there exist fully imperative model transformation approaches (chiefly TRL), the majority of approaches can be categorised as declarative. The gap between these two extremes is currently under-explored. The QVT-Partners approach is almost alone in exploring this gap, but achieves only limited success since it is effectively an umbrella for two different approaches: a wholly declarative approach and a mostly imperative approach. Whilst there is some reuse of concepts between the two approaches, the overall effect is far from seamless from a user point of view.

Therefore the first choice I make about the specific model transformation approaches that I will investigate in this thesis is that they should fuse elements of both declarative and imperative approaches. Note that there is a deliberate synergy with the choice in section 3.4.1 to use an imperative language to implement embedded model transformation DSLs: one would hope to be able to reuse aspects of the host imperative language within the model transformations DSL.

The second choice that needs to be made relates to the types of model transformations to be attempted. Since one of the purposes of Converge is to reduce the implementation burden when creating model transformation approaches, it is important to present some evidence that it is useful for implementing more than one model transformation approach. I therefore choose to define a ‘standard’ unidirectional stateless model transformation approach in chapter 5. I then extend this approach to define a unidirectional change propagating model transformation approach in chapter 6.

3.4.3. Thesis aims

In summary, this thesis has the following aims:

1. To provide an extensible dynamically typed OO programming language which allows DSLs to be embedded within it.

2. To provide a non-trivial example of a DSL within the extendable programming language.

3. To examine new approaches for expressing stateless and change propagating transformations.

3.4.4. Assessment criteria

In this section I present the criteria by which Converge (chapter 4) and the model transformation approaches (chapters 5 and 6) can be judged by.

Assessment criteria for a DSL implementation technology

Reasonable criteria by which one can judge the success of a technology that aims to aid the implementation of DSLs are simple to state, but difficult to assess by. For example the fundamental criteria in the context of this thesis is whether the technology palpably reduces the required implementation effort. However, it is beyond the scope of this thesis to provide hard numbers in the form of comparative time measurements or lines of code since — such a task would be a major undertaking.

In the context of the overall thesis the only feasible way to assess the implementation technology proposed is an indirect one: through the model transformation approaches presented in chapters 5 and 6. If those approaches are seen to be useful and novel in and of themselves, then one can surmise that, at worst, the proposed implementation technology did not hamper their development and, at best, it actively helped their development.

Note that although it is hard to assess the success of a model transformation implementation technology in general, the specific technology proposed in this thesis is not only intended to be used for implementing model transformation approaches. In fact, Converge is proposed as a general GPL and can be assessed completely independently of model transformations. Therefore part of chapter 4 presents a comparison of Converge to other GPLs, and details how some of the lessons learned from its development could be used to augment mainstream GPLs with some of its more novel features.

Assessment criteria for a model transformation approach

A model transformation approach can be assessed on the following two criteria:

1. The scope of the problem it tackles (compared to the method of section 2.2.4).

2. The practicality of the solution.

The first of these criteria is relatively easy to assess, the second considerably less so. Nevertheless the second point is an important one: a powerful solution which requires of a potential user undue effort can not really be considered to present a realistic solution to what is an inherently practical problem.

Chapter 4.

The Converge programming language

This chapter presents the design of a new dynamically typed OO language Converge designed to facilitate the implementation of DSLs. Converge is a dynamically typed imperative programming language, capable of compile-time meta-programming, and with an extendable syntax. Although Converge has been designed with the aim of implementing different model transformation approaches as embedded DSLs in mind, it is also a GPL, albeit one with unusually powerful features.

This chapter comes in four main parts. The first part documents the basics of the Converge language itself. The second part details Converge's compile-time meta-programming and syntax extension facilities, including a section detailing suggestions for how some of Converge's novel features could be added to similar languages. The third part of the chapter explains Converge's syntax extension facility, and includes a simple example of syntax extension in use. The final part of the chapter documents a user extension which allows simple UML modelling languages to be embedded within Converge. As well as being a practical demonstration of Converge's features, this facility is used extensively throughout the remainder of the thesis.

4.1. Converge basics

This section gives a brief overview of the core Converge features that are relevant to the main subject of this thesis. Since most of the basic features of Converge are similar to other similar programming language, this section is intentionally terse. However it should allow readers familiar with a few other programming languages the opportunity to quickly come to grips with the most important areas of Converge, and to determine the areas where it differs from other languages.

4.1.1. Syntax, scoping and modules

Converge's most obvious ancestor is Python [vR03] resulting in an indentation based syntax, a similar range and style of datatypes, and general sense of aesthetics. The most significant difference is

that Converge is a slightly more static language: all namespaces (e.g. a modules' classes and functions, and all variable references) are determined statically at compile-time whereas even modern Python allows namespaces to be altered at run-time[1]. Converge's scoping rules are also different from Python's and many other languages, and are intentionally very simple. Essentially Converge's functions are synonymous with both closures and blocks. Converge is lexically scoped, and there is only one type of scope (as opposed to Python's notion of local and global scopes). Variables do not need to be declared before their use: assigning to a variable anywhere in a block makes that variable local throughout the block (and accessible to inner blocks). However if a variable is declared via the `nonlocal` keyword, then Converge searches for the first block containing an assignment of that variable, from the current block outwards; reading and assigning to the variable will then refer to the outer variable in the block containing the `nonlocal` definition, but not by default to further inner blocks. Variable references search in order from the innermost block outwards, ultimately resulting in a compile-time error if a suitable reference is not found. As in Python, fields within a class are not accessible via the default scoping mechanism: they must be referenced via the `self` variable which is automatically brought into scope in any *bound function* (functions declared within a class are automatically bound functions). Converge's justification for this is subtly different than Python's, which has this feature to aid comprehension; although this is equally true in Converge, without this feature, namespaces would not be statically calculable since an objects slots are not always known at compile-time.

Converge programs are split into modules, which contain a series of *definitions* (imports, functions, classes and variable definitions). Unlike Python, each module is individually compiled into a bytecode file by the Converge compiler `convergec` and linked by `convergel` to produce a static bytecode executable which can be run by the Converge VM. If a module is the *main module* of a program (i.e. passed first to the linker), Converge calls its `main` function to start execution. The following module shows a caching Fibonacci generating class, and indirectly shows Converge's scoping rules (the `i` and `fib_cache` variables are local to the functions they are contained within), printing `8` when run:

```
import Sys

class Fib_Cache:
  func init():
    self.cache := [0, 1]

  func fib(x):
    i := self.cache.len()
    while i <= x:
      self.cache.append(self.cache[i - 2] + self.cache[i - 1])
      i += 1
    return self.cache[x]
```

[1]Prior to version 2.1, Python's namespaces were determined almost wholly dynamically; this often lead to subtle bugs, and hampered the utility of nested functions.

```
func main():
  fib_cache := Fib_Cache()
  Sys.println(fib_cache.fib(6))
```

Compiling and running this fragment looks as follows:

```
$ converge convergec -o fib.cvb fib.cv
$ converge convergel -o fib fib.cvb lib/libconverge.cvl
$ converge fib
8
```

As in Python, Converge modules are executed from top to bottom when they are first imported. This is because functions, classes and so on are normal objects within a Converge system that need to be instantiated from the appropriate built-in classes – therefore the order of their creation can be significant e.g. a class *must* be declared before its use by a subsequent class as a superclass. Note that this only affects references made at the modules top-level – references e.g. inside functions are not restricted thus.

4.1.2. Functions

Converge uses the term function both in its traditional programming sense of a stand-alone function (or 'procedure'), and also for functions which reside in classes (often called methods). The reason for this is that 'normal' functions and 'methods' are not restricted in Converge to only their traditional rôles: 'normal' functions can reside in classes and 'methods' can reside outside of classes. When it is important to distinguish between the two, Converge has two distinct types: *unbound functions* ('normal' functions) and *bound functions* ('methods'). Bound functions expect to have an implicit first argument of the self object[2]; however they can not have arguments applied to it directly. Extracting a bound function from an object creates a *function binding* which wraps a bound function and a reference to the self object into an object which can then have arguments applied to it. Function bindings can be manually created by instantiating the `Func_Binding` class, which allows bound functions to be used with arbitrary self objects.

In normal use, Converge automatically assumes that the keyword `func` introduces an unbound function if it is used outside class, and a bound function if used inside a class. Using the `bound_func` or `unbound_func` keywords overrides this behaviour. Functions, bound or unbound, can have zero or more parameters; prefixing the final parameter in a function with a `*` denotes the 'var args' parameter.

An important feature of functions is their `apply` slot which applies a list of objects as parameters to the function. This allows argument lists of arbitrary size to be built and applied at run-time.

[2]Note that unlike Python, Converge does not force the user to explicitly list `self` as a function parameter.

4.1.3. Goal-directed evaluation

An important, if less obvious, influence to Converge is Icon [GG96a]. Since Icon is likely to be unfamiliar to be most readers, a brief overview of Icon is instructive in understanding why it possesses an unusual, and interesting, feature set. Icon's chief designer was Ralph Griswold, and is a descendant of the SNOBOL series of programming languages – whose design team Griswold had been a part of – and SNOBOL's short-lived successor SL5. SNOBOL4 in particular was specifically designed for the task of string manipulation, but an unfortunate dichotomy between pattern matching and the rest of the language, and the more general problems encountered when trying to use it for more general programming issues ensured that, whilst successful, it never achieved mass acceptance; SL5 suffered from almost the opposite problem by having an over-generalized and unwieldy procedure mechanism. See Griswold and Griswold [GG93] for an insight into the process leading to Icon's conception. Since programs rarely manipulate strings in isolation, post-SL5 Griswold had as his aim to build a language which whilst being aimed at non-numeric manipulation also was usable as a general programming language. The eventual result was Icon [GG96a, GG96b], a language still in use and being developed to this day. In order to fulfil the goal of practical string manipulation, the premises on which Icon is founded are not only fundamentally different from those normally associated with GPLs, but are also tightly coupled with one another.

As Icon, Converge is an expression-based language, with similar notions of expression *success* and *failure*. In essence, expressions which succeed produce a value; expressions which fail do not produce a value and percolate the failure to their outer expression. For example the following fragment:

```
func main():
  x := 1 < 2
  y := 2 < 1
  Sys.println(x)
  Sys.println(y)
```

leads to the following output:

```
2
Traceback (most recent call last):
  File "expr.cv", line 5, column 13, in main
Unassigned_Var_Exception: Var has not yet been assigned to.
```

This is because when the expression `2 < 1` is evaluated, it fails (since 2 is not less than 1); the failure percolates outwards and prevents the assignment of a value to the variable `y`. Note that failure does not percolate outwards to arbitrary points: failure can not cross *bound expressions*. A bound expression thus denotes a 'stop point' for backtracking. The most obvious point at which bound expressions occur is when expressions are separated by newlines in an Converge program although bound expressions occur in various other points. For example, each branch of an `if` expression is bound, which prevents the failure of a branch causing the entire `if` expression to be re-evaluated.

Converge directly inherits Icon's bound expression rules which largely preserve traditional imperative language evaluation strategies, even in the face of backtracking.

Success and failure are the building blocks for goal-directed evaluation, which is essentially a limited form of backtracking suitable for imperative programming languages. Functions which contain the `yield` statement are *generators* and can produce more than one return value. The `yield` statement is an alternative type of function return which effectively freezes the current functions closure and stack, and returns a value to the caller; if backtracking occurs, the function is resumed from its previous point of execution and may return another value. Generators complete by using the `return` statement. Since the `return` statement returns the `null` object if no expression is specified, generators typically use `return fail` to ensure that the completion of the generator does not cause one final loop of the caller — `return`'ing the `fail` object causes a function to percolate failure to its caller immediately.

The most frequent use of generators is seemingly mundane, and occurs in the following idiom, which uses the `iterate` generator on a list to print each list element `l` on a newline:

```
l := [3, 9, 27]
for x := l.iterate():
  Sys.println(x)
```

In simple terms, the `for` construct evaluates its condition expression and after each iteration of the loop backtracks in an attempt to pump the condition for more values. This idiom therefore subsumes the verbose encoding of iterators found in most OO languages.

Generators can be used for much more sophisticated purposes. Consider first the following generator which generates all Fibonacci numbers from 1 to `high`:

```
func fib(high):
  a, b := [0, 1]
  while b < high:
    yield b
    a, b := [b, a + b]

  return fail
```

The `for` construct exhaustively evaluates its condition (via backtracking) until it can produce no more values. Therefore the following fragment prints all Fibonacci values from 1 to 100000:

```
for f := fib(100000):
  Sys.println(f)
```

The conjunction operator & conjoins two or more expressions; the failure of any part of the expression causes backtracking to occur. Backtracking resumes the most recent generator which is still capable of producing values, only resuming older generators when more recent ones are exhausted. Thus backtracking in Converge is entirely deterministic because the sequence in which alternatives are tried is explicitly specified by the programmer – this makes the evaluation strategy significantly different than that found in logic languages such as Prolog. If all expressions in a conjunction suc-

ceed, the value of the final expression is used as the value of the conjunction. If failure occurs, and there are no generators capable of producing more values to be resumed, then the conjunction itself fails.

Combining the `for` construct with the `&` operator can lead to terse, expressive examples such as the following which prints all Fibonacci numbers wholly divisible by 3 between 1 and 100000:

```
for Sys.println(f := fib(100000) & f % 3 == 0 & f)
```

A brief explanation of this can be instructive. Firstly `f := fib(100000)` pumps the `fib` generator and assigns each value it returns to the variable `f`. Since it is contained within the first expression of the `&` operator, when the `fib` generator completes, its failure causes the `f := ...` assignment to fail, which causes the entire `&` operator to fail thus causing the `for` construct to fail and complete. Secondly `f % 3 == 0` checks whether `f` modulo 3 is equal to 0 or not; if it is not, failure occurs and backtracking occurs back to the `fib` generator. Since `f % 3 == 0`, if it succeeds, always evaluates to 0 (`==` evaluates to its right hand argument on success), the final expression of `f` produces the value of the variable which `Sys.println` then prints.

Neither Icon or Converge possess standard boolean logic since equivalent functionality is available through other means. The conjunction operator acts as an ‘and’ operator. Although the disjunction operator `|` is generally used as ‘or’, it is in fact a generator that successively evaluates all its expressions, producing values for those expressions which succeed. Thus in most circumstances the `|` operator neatly preserves the normal expectation of ‘or’ – that it evaluates expressions in order only until it finds one which succeeds – whilst also providing useful extra functionality.

This section has detailed the most important aspects of Converge’s Icon-esque features, but for a more thorough treatment of these features I recommend Icon’s manual [GG96a] — virtually all the material on goal-directed evaluation is trivially transferable from Icon to Converge. Gudeman [Gud92] presents a detailed explanation of goal-directed evaluation in general, with its main focus on Icon, and presents a denotational semantics for Icon’s goal-directed evaluation scheme. Proebsting [Pro97] and Danvy et al. [DGR01] both take subsets of Icon chosen for their relevance to goal-directed evaluation, compiling the fragments into various programming languages (Danvy *et. al* also specify their Icon subset with a monadic semantics); both papers provide solid further reading on the topic.

While loops

Converge also contains a `while` construct. The difference between the `for` and `while` constructs is initially subtle, but is ultimately more pronounced than in most languages. In essence, each time a `for` loop completes, the construct backtracks to the condition expression and pumps it for a new

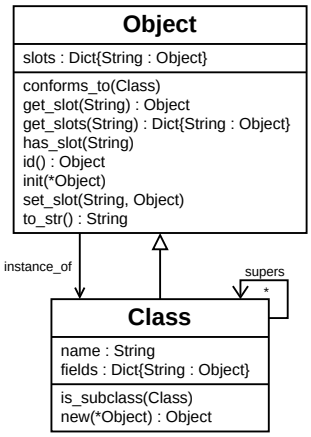


Figure 4.1.: Core Converge data model.

value. In contrast, a `while` construct evaluates its expression anew after each iteration. This means that if the condition of a `while` construct is a generator it can only ever generate a maximum of one value before it is discarded. To emphasise this, the following code endlessly repeats, printing 1 on each iteration:

```
while f := fib(100000):
  Sys.println(f)
```

4.1.4. Data model

Converge's OO features are reminiscent of Smalltalk's [GR89] everything-is-an-object philosophy, but with a prototyping influence that was inspired by Abadi and Cardelli's theoretical work [AC96]. The internal data model is derived from ObjVLisp [Coi87]. Classes are provided as a useful and common convenience but are not fundamental to the object system in the way they are to most OO languages. The system is bootstrapped with two base classes `Object` and `Class`, with the latter being a subclass of the former and both being instances of `Class` itself: this provides a full metaclass ability whilst avoiding the class / metaclass dichotomy found in Smalltalk [BC89, DM95]. The core data model can be seen in figure 4.1. Note that the `slots` field in the `Object` class is conceptual and can only be accessed via the `get_slot` and `get_slots` functions.

In the Smalltalk school of OO, objects consist of a slot for each attribute in the class; calling a method on an object looks for a suitable method in the object's instantiating class (and possibly its superclasses). In contrast Converge, by default, creates objects with a slot for each field in a class, including methods. This therefore moves method overriding in classes to object creation time, rather than the more normal invocation time. This is possible since, as in Python, a function's name is the only factor to be taken into account when overriding. Object creation in Converge thus has a higher overhead than in most OO languages; this is offset by the fact that calling a function in an

object is faster (since classes and super-classes do not need to be searched). The reason for this design decision is to ensure that all objects in a Converge system are 'free objects' in that they can be individually manipulated without directly affecting other objects, a feature which can prove useful when manipulating and transforming objects. This behaviour also mirrors the real world where, for example, changing a car's design on paper does not change actual cars on the road; it does not however reflect the behaviour of non-prototyping OO languages. For example, in Converge adding (or deleting) a method in a class does not automatically affect objects which are instances of that class, whereas in Python all of the classes instances would appear to grow (or lose) a method. Note that this flexibility also allows objects to be dynamically reclassified without additional language features; this contrasts with more static languages where additional language features need to be added to allow this feature (see Drossopoulou *et. al* for a concrete proposal [DDDCG02]). Although not explored further in this thesis, such a feature is highly desirable for so-called 'update in place' transformations.

From a practical point of view it is important to note that in normal use most users will be unaware of the difference between Converge's object creation scheme and its more normal counterparts since common usage does not involve directly manipulating the meta-system. Note that this entire area of behaviour can be overridden by using meta-classes and the meta-object protocol (section 4.1.7).

Metaclasses

In similar fashion to ObjVLisp, metaclasses are otherwise normal objects which possess a `new` slot. `Class` is the default metaclass; individual classes can instantiate a different class via the `metaclass` keyword. Metaclasses typically subclass `Class`, although this is not a requirement. A simple example of a useful metaclass is the following singleton metaclass [GHJV94] which allows classes to enforce that at most one instance of the class can exist in the system. Noting that `exbi` (EXtract and BInd) can be viewed as being broadly equivalent to other languages `super` keyword, the `Singleton` class is defined and used as follows:

```
class Singleton(Class):
  func new():
    if not self.has_slot("instance"):
      self.instance := exbi Class.new()
    return self.instance

class M metaclass Singleton:
  pass
```

Note that the `new` function in `Class` automatically calls the `init` function on the newly created object, passing it all the arguments that were passed to `new`.

Built-in data types

Converge provides a similar set of built-in data types to Python: strings, integers, dictionaries (key / value pairs) and sets. Dictionary keys and set elements should be immutable (though this is not enforced, violating this expectation can lead to unpredictable results), and must define `==` and `hash` functions, the latter of which should return an integer representing the object. All built-in types are subclasses of `Object`, and can be sub-classed by user classes (although the current implementation restricts user classes to sub-classing a maximum of one built-in type).

4.1.5. Comparisons and comparison overloading

Converge defines a largely standard set of binary operators. The lack of standard boolean logic in Converge means that the `not` operator is slightly unusual and is not classed as a comparison operator. Rather than `not` taking in a boolean value and returning its negated value, the `not` operator evaluates its expression and, if it fails, `not` succeeds and produces the `null` object. If the expression succeeds, the value produced is discarded and the `not` operator fails.

Objects can control their involvement in comparisons by defining, or overriding, the functions which are called by the various comparison operators. Functions are passed an object for comparison, and should fail if the comparison does not hold, or return the object passed to them if it does. Comparison operators are syntactic sugar for calling a function of the same name in the left hand side object (e.g. the `==` operator looks up the `==` slot in an object).

Note that although the Converge grammar (appendix A) bundles the `is` operator into the `comparison_op` production, it is unlike the other comparison operators in that it tests two objects for equality of their identities, and can not be overridden by user objects.

4.1.6. Exceptions

Converge provides exception handling that is largely similar to Python. The `raise` expression raises an exception, printing a detailed stack-trace, the type of the exception and a message from the exception object itself. All exceptions must be instances of the `Exception` class in the `Exceptions` module. The `try ... catch` construct is used to test and capture exceptions.

4.1.7. Meta-object protocol

Converge implements a simple but powerful Meta-Object Protocol (MOP) [KdRB91], which allows objects to control all behaviour relating to slots. The default MOP is contained within the `Object` class and comprises the `get_slot`, `get_slots`, `has_slot` and `set_slot` functions. These

can be arbitrarily overridden to control which slots the object claims it has, and what values such slots contain. Note that *all* accesses go through these functions; if they are overridden in a subclass, the user must exercise caution to call the 'master' MOP functions in the `Object` class to prevent infinite loops. The following example shows a MOP which returns a default value of `null` for unknown slot names:

```
class M:
  func get_slot(n):
    if not self.has_slot(n):
      return null
    return exbi Object.get_slot(n)
```

4.1.8. Differences from Python

Converge deliberately presents a feature set which can be used in a fashion similar to Python. Programmers used to Python can easily use Converge in a Python-esque fashion although they will miss out on some of Converge's more advanced features. The chief differences from Python are that Converge is a more static language, able to make stronger guarantees about namespaces, and that Converge is an expression based language rather than Python's statement based approach. Converge has a more uniform object system, and less reliance on a battery of globally available built-in functions than Python.

One small change from Python to Converge is a generalization of the somewhat confusingly named `finally` branch which can be attached to Python's `for` and `while` loops. The `finally` branch is executed if the loop construct terminates naturally (i.e. `break` is not called). Converge renames the `finally` branch to `exhausted` and also allows a `broken` branch to be added which will be called if a `break` is encountered. A slightly contrived example of this feature is as follows:

```
high := 10000
for x := fib(high):
  if x % 9 == 0:
    break
exhausted:
  Sys.println("No Fibonacci numbers wholly divisible by 9 upto ", high)
broken:
  Sys.println("Fibonacci number ", x, " wholly divisible by 9")
```

4.1.9. Differences from Icon

Converge's expression system is highly similar to Icon. Provided they can adjust to the Python-esque veneer, Icon programmers will have little difficulty exploiting Converge's expression system and implementation of goal-directed evaluation. There are however two significant differences in Converge's functions and generators.

Firstly, whereas Icon functions which do not have a `return` expression at the end of a function have an implicit `return fail` added, Converge functions instead default to `return null` (as

do Python functions). Icon takes its approach so that generators do not accidentally return an extra object when they should instead fail, and Converge originally took the same approach as Icon. However in practise it is quite common, when developing code, to write incomplete functions — often one part of the code not initially filled in is the function's final `return` expression. Such functions then cause seemingly bizarre errors since they do not return a value, causing assignments in calling functions to fail and so on (indeed, this happened surprisingly frequently in the early stages of Converge development). Since the proportion of generators to normal functions is small, it seems more sensible to optimise the safety of normal functions at the expense of the safety of generators. As can be seen from section 4.1.3, generators in Converge generally have `return fail` as their final action in order to emulate Icon's behaviour.

Secondly, Converge does not propagate generation across a `return` expression. In Icon, if `f` is a generator then `return f()` turns the function containing the `return` expression into a generator itself which produces all the values that `f` produces. Converge does not emulate this behaviour, which somewhat arbitrarily turns `return` into a sort of `for` construct in certain situations that can only be determined by knowing whether the expression contains a generator. The same behaviour can be obtained in Converge via the following idiom:

```
for yield f()
return fail
```

Finally, two important features present in Icon are absent in Converge. One is the concept of string-scanning expressions, which are a specialised form of string matching; such a concept is not general enough for Converge but, if required, could be expressed as a DSL (see section 4.4). Second is Icon's reversible assignment operator `<-`. Reversible assignment acts as normal assignment except in the presence of backtracking, which will restore the variable being assigned to its original value. Whilst conceptually a useful idea, this is used reasonably infrequently in practise and is thus not included in Converge.

4.1.10. Implementation

The current Converge implementation consists of a Virtual Machine (VM) written in C, and a compiler written in Converge itself (the current compiler was bootstrapped several generations ago from a much simpler Python version). The VM has a simplistic semi-conservative garbage collector which frees the user from memory management concerns. The VM uses a continuation passing technique at the C level to make the implementation of goal-directed evaluation reasonably simple and transparent from the point of view of extension modules. Its instruction set is largely based on Icon's, although the VM implementation itself shares more in common with modern VM's such as Python's.

This thesis is not overly concerned with the implementation of the VM and compiler. Interested

readers are encouraged to visit `http://convergepl.org/` where the VM and compiler can be downloaded and inspected.

4.1.11. Parsing

An aspect of Converge and its implementation that is particularly important throughout this thesis is its ability to easily parse text. Converge implements a parser toolkit (the Converge Parser Kit or CPK) which contains a parsing algorithm based on that presented by Earley [Ear70]. Earley's parsing algorithm is interesting since it accepts and parses any Context Free Grammar (CFG) — this means that grammars do not need to be written in a restricted form to suit the parsing algorithm, as is the case with traditional parsing algorithms such as LALR. By allowing grammars to be expressed without concern for many of the traditional parsing concerns, a barrier to DSL development is removed. Practical implementations of Earley parsers have traditionally been scarce, since the flexibility of the algorithm results in slower parsing times than traditional parsing algorithms. The CPK utilises some (though not all) of the additional techniques developed by Aycock and Horspool [AH02] to improve its parsing time, particularly those relating to the ε production. Even though the CPK contains an inefficient implementation of the algorithm, on a modern machine, and even with a complex grammar, it is capable of parsing in the low hundreds of lines per second which is sufficient for the purposes of this thesis. The performance of more sophisticated Earley parsers such as Accent [Sch05] suggest that the CPK's performance could be raised by approximately an order of magnitude with relatively little effort.

Parsing in Converge is preceded by a tokenization (also known as lexing) phase. The CPK provides no special support for tokenization, since the built-in regular expression library makes the creation of custom tokenizers trivial. Tokenizers are expected to return a list of objects, each of which has slots `type`, `value`, `src_file` and `src_offset`. The first two slots represent the type (i.e. `ID`) and value (i.e. `height`) of a token and must be strings; the latter two slots record both the file and character offset within the file that a particular token originated in. The tokenizer for Converge itself is somewhat unusual in that it needs to understand about indentation in order that the grammar can be expressed satisfactorily. Each increase in the level of indentation results in a `INDENT` token being generated; each decrease results in a `DEDENT` followed by a `NEWLINE` token. Each newline on the same level of indentation results in a `NEWLINE` token.

The CPK implements an EBNF style system – a BNF system with the addition of the Kleene star. CPK production rules consist of a rule name, and one or more alternatives. Each alternative consists of tokens, references to other rules and groupings. Currently the only form of grouping accepted is the Kleene star. Since this thesis contains several grammars writing in the CPK, the grammar of the

CPK itself is as follows:

⟨*grammar*⟩ ::= ⟨*rule*⟩*

⟨*rule*⟩ ::= ‘`ID`’ ⟨*rule_alternative*⟩*

⟨*rule_alternative*⟩ ::= ‘`::=`’ ⟨*rule_elem*⟩*
| ‘`::=`’ ⟨*rule_elem*⟩* ‘`%PRECEDENCE`’ ‘`INT`’

⟨*rule_elem*⟩ ::= ⟨*atom*⟩
| ⟨*grouping*⟩

⟨*grouping*⟩ ::= ‘`{`’ ⟨*atom*⟩ ‘`}*`’

⟨*atom*⟩ ::= ‘`"`’ ‘`TOKEN`’ ‘`"`’
| ‘`ID`’

Since Earley grammars can express any CFG, grammars can be ambiguous — that is, given inputs can satisfy the grammar in more than one way. In order to disambiguate between alternatives when building the parse tree, the CPK allows grammar rules to have a precedence attached to them; if more than one rule has been used to parse a given group of tokens, the rule with the highest precedence is used[3].

In order to use the CPK, the user must provide it with a grammar, the name of a start rule within the grammar, and a sequence of tokens. The result of a CPK parse is an automatically constructed parse tree, which is represented as a nested Converge list of the form [*`production name, token or list`*$_1$*`, ..., token or list`*$_n$]. The following program fragment shows a CPK grammar for a simple calculator:

```
GRAMMAR := """
S ::= E
E ::= E "+" E  %precedence 10
  ::= E "*" E  %precedence 30
  ::= "(" E ")"
  ::= N "INT"  %precedence 10
N ::= "-"
  ::=
"""
```

Assuming the existence of a suitable `tokenize` function, an example program which uses this grammar to parse input is as follows:

```
import CPK.Grammar, CPK.Parser

func calc_parse(input):
  grammar := Grammar.Grammar(GRAMMAR, "S")
  tokens := tokenize(input)
  parser := Parser.Parser(grammar)
```

[3]Note that there is another, much rarer, type of ambiguity involving alternatives which contain different number of tokens. These are currently always resolved in favour of the alternative containing the least number of tokens, no matter its precedence. This generally gives the expected behaviour, but can cause problems in some rare cases. This limitation is purely due to a naïve implementation.

```
tree := parser.parse(tokens)
Sys.println(tree)
```

The parse tree is printed out as:

```
["S", ["E", ["E", ["N"], <INT 5>], <+>, ["E", ["E", ["N"], <INT 2>], <*>, ["E",
["N"], <INT 3>]]]]
```

This is somewhat easier to visualize when using the `parse_tree` function in a `Parser` instance to format the list as a tree:

```
S->
  E ->
    E ->
      N ->
      INT <5>
    +
    E ->
      E ->
        N ->
          -
        INT <2>
      *
      E ->
        N ->
        INT <3>
```

The full Converge grammar can be seen in appendix A.

4.1.12. Related work

This section has made several comparisons between Converge, and Icon and Python in particular. These are not repeated in this subsection.

The Unicon project [JMPP03] is in the reasonably advanced stages of extending Icon with object orientated features. It differs significantly from Converge in maintaining virtually 100% compatibility with Icon. Unicon's extensions to Icon, effectively being a bolt-on to the original, mean the resulting language features are not as closely integrated as they are in Converge. Godiva [Jef02], which claims as a goal to be a 'very high level dialect of Java', also incorporates goal-directed evaluation. In reality, Godiva's claim to be a dialect of Java is slightly tenuous: whilst it shares some syntax, the semantics are substantially different. Neither Unicon nor Godiva have a meta-circular data model (see section 4.5.2), and both are less dynamic languages than Converge.

4.2. Compile-time meta-programming

4.2.1. Background

Compile-time meta-programming provides the user of a programming language with a mechanism to interact with the compiler to allow the construction of arbitrary program fragments by user code. In this section I detail an extension to the core Converge language which adds compile-time meta-

programming facilities similar to TH. Since this is the first time that facilities of this nature have been added to a dynamically-typed OO language such as Converge, section 4.3 details the implications of adding such a feature to similar languages.

4.2.2. A first example

The following program is a simple example of compile-time meta-programming, trivially adopted from its TH cousin in [COST04]. `expand_power` recursively creates an expression that multiplies `n` `x` times; `mk_power` takes a parameter `n` and creates a function that takes a single argument `x` and calculates x^n; `power3` is a specific power function which calculates n^3:

```
func expand_power(n, x):
  if n == 0:
    return [| 1 |]
  else:
    return [| $<<x>> * $<<expand_power(n - 1, x)>> |]

func mk_power(n):
  return [|
    func (x):
      return $<<expand_power(n, [| x |])>>
  |]

power3 := $<<mk_power(3)>>
```

The user interface to compile-time meta-programming is inherited from TH: quasi-quote expressions `[| ... |]` build abstract syntax trees - ITree's in Converge's terminology - that represent the program code contained within them, and the splice annotation `$<<...>>` evaluates its expression at compile-time (and before VM instruction generation), replacing the splice annotation itself with the ITree resulting from its evaluation. When the above example has been compiled into VM instructions, `power3` essentially looks as follows:

```
power3 := func (x):
  return x * x * x * 1
```

By using the quasi-quotes and splicing mechanisms, we have been able to synthesise at compile-time a function which can efficiently calculate powers without resorting to recursion, or even iteration. Note how apart from the quasi-quotes and splicing mechanisms no extra features have been added to the base language – unlike LISP style languages, all parts of a Converge program are first-class elements regardless of whether they are executed at compile-time or run-time.

This terse explanation hides much of the necessary detail which can allow readers who are unfamiliar with similar systems to make sense of this synthesis. In the following sections, I explore the interface to compile-time meta-programming in more detail, explaining the system step by step.

4.2.3. Splicing

The key part of the 'powers' program is the splice annotation in the line `power3 := $<<mk_power(3)>>`. The top-level splice tells the compiler to evaluate the expression between the chevrons at compile-time, and to include the result of that evaluation in the module for ultimate bytecode generation. In order to perform this evaluation, the compiler creates a temporary or 'dummy' module which contains all definitions up to, but excluding, the definition the splice annotation is a part of; to this temporary module a new splice function (conventionally called `$$splice$$`) is added which contains a single expression `return` *`splice expr`*. This temporary module is compiled to bytecode and injected into the running VM, whereupon the splice function is called. Thus the splice function 'sees' all the definitions prior to it in the module, and can call them freely – there are no other limits on the splice expression. The splice function must return a valid ITree which the compiler uses in place of the splice annotation.

Evaluating a splice expression leads to a new 'stage' in the compiler being executed. Converge's rules about which references can cross the staging boundary are simple: only references to top-level module definitions can be carried across the staging boundary (see section 4.2.5). For example the following code is invalid since the variable `x` will only have a value at run-time, and hence is unavailable to the splice expression which is evaluated at compile-time:

```
func f(x): $<<g(x)>>
```

Although the implementation of splicing in Converge is more flexible than in TH – where splice expressions can only refer to definitions in imported modules – it raises a new issue regarding forward references. This is tackled in section 4.2.9.

Note that splice annotations within a file are executed strictly in order from top to bottom, and that splice annotations can not contain splice annotations.

Permissible splice locations

Converge is more flexible than TH in where it allows splice annotations. A representative sample of permissible locations is:

Top-level definitions. Splice annotations in place of top-level definitions must return an ITree, or a list of ITree's, each of which must be an assignment.

Function names. Splice annotations in place of function names must return a `Name` (see section 4.2.6).

Expressions. Splice annotations as expressions can return any normal ITree. A simple example is `$<<x>> + 2`. We saw another example in the 'powers' program with `power3 :=`

`$<<mk_power(3)>>`.

Within a block body. Splice annotations in block bodies (e.g. a functions body) accept either a single ITree, or a list of ITree's. Lists of ITree's will be spliced in as if they were expressions separated by newlines.

A contrived example that shows the last three of these splice locations (in order) in one piece of code is as follows:

```
func $<<create_a_name()>>():
  x := $<<f()>> + g()
  $<<list_of_exprs()>>
```

At compile-time, this will result in a function named by the result of `create_a_name` and containing 1 or more expressions, depending on the number of expressions returned in the list by `list_of_exprs`.

Note that the splice expressions must return a valid ITree for the location of a splice annotation. For example, attempting to splice in a sequence of expressions into an expression splice such as `$<<x>> + 2` results in a compile-time error.

4.2.4. The quasi-quotes mechanism

In the previous section we saw that splice annotations are replaced by ITree's. In many systems the only way to create ITree's is to use a verbose and tedious interface of ITree creating functions which results in a 'style of code [which] plagues meta-programming systems' [WC93]. LISP's quasi-quote mechanism allows programmers to build up LISP S-expressions (which, for our purposes, are analogous to be ITree's) by writing normal code prepended by the backquote ` notation; the resulting S-expression can be easily manipulated by a LISP program. Unfortunately LISP's syntactic minimalism is unrepresentative of modern languages, whose rich syntaxes are not as easily represented and manipulated.

MetaML and, later TH, introduce a quasi-quotes mechanism suited to syntactically rich languages. Converge inherits TH's Oxford quotes notation `[| ...|]` notation to represent a quasi-quoted piece of code. A quasi-quoted expression evaluates to the ITree which represents the expression inside it. For example, whilst the raw Converge expression `4 + 2` evaluates to, and prints out as, `6`, `[| 4 + 2 |]` evaluates to an ITree which prints out as `4 + 2`. Thus the quasi-quote mechanism constructs an ITree directly from the users input - the exact nature of the ITree is of immaterial to the casual ITree user, who need not know that the resulting ITree is structured along the lines of *add(int(4), int(2))*.

To match the fact that splice annotations in blocks can accept sequences of expressions to splice in,

the quasi-quotes mechanism allows multiple expressions to be expressed within it, split over newlines. The result of evaluating such an expression is, unsurprisingly, a list of ITree's.

Note that as in TH, Converge's splicing and quasi-quote mechanisms cancel each other out: `$<<[| x |]>>` is equivalent to *x* (though not necessarily vice versa).

Splicing within quasi-quotes

In the 'powers' program, we saw the splice annotation being used within quasi-quotes. The explanation of splicing in section 4.2.3 would seem to suggest that the splice inside the quasi-quoted expression in the `expand_power` function should lead to a staging error since it refers to variables `n` and `x` which were defined outside of the splice annotation. In fact, splices within quasi-quotes work rather differently to splices outside quasi-quotes: most significantly the splice expression itself is *not* evaluated at compile-time. Instead the splice expression is copied as-is into the code that the quasi-quotes transforms to. For example, the quasi-quoted expression `[| $<<x>> + 2 |]` leads to an ITree along the lines of *add(x, int(2))* – the variable `x` in this case would need to contain a valid ITree. As this example shows, since splice annotations within quasi-quotes are executed at run-time they can access variables without staging concerns.

This feature completes the cancelling out relationship between splicing and quasi-quoting: `[| $<<x>> |]` is equivalent to *x* (though not necessarily vice versa).

4.2.5. Basic scoping rules in the presence of quasi-quotes

The quasi-quote mechanism can be used to surround any Converge expression to allow the easy construction of ITree's. Quasi-quoting an expression also has another important feature: it fully respects lexical scoping. Take the following contrived example of module `A`:

```
func x(): return 4
func y(): return [| x() * 2 |]
```

and module `B`:

```
import A, Sys
func x(): return 2
func main(): Sys.println($<<A.y()>>)
```

The quasi-quotes mechanisms ensures that since the reference to `x` in the quasi-quoted expression in `A.y` refers lexically to `A.x`, that running module `B` prints out `8`. This example shows one of the reasons why Converge needs to be able to statically determine namespaces: since the reference of `x` in `A.y` is lexically resolved to the function `A.x`, the quasi-quotes mechanism can replace the simple

reference with an *original name*[4] that always evaluates to the slot `x` within the specific module `A` wherever it is spliced into, even if `A` is not in scope (or a different `A` is in scope) in the splice location.

Some other aspects of scoping and quasi-quoting require a more subtle approach. Consider the following (again contrived) example:

```
func f(): return [| x := 4 |]

func g():
  x := 10
  $<<f()>>
  y := x
```

What might one expect the value of `y` in function `g` to be after the value of `x` is assigned to it? A naïve splicing of `f()` into `g` would mean that the `x` within `[| x := 4 |]` would be captured by the `x` already in `g` – `y` would end with the value `4`. If this was the case, using the quasi-quote mechanism could potentially cause all sorts of unexpected interactions and problems. This problem of variable capture is well known in the LISP community, and hampered LISP macro implementations for many years until the concept of hygienic macros was invented [KFFD86]. A new subtlety is now uncovered: not only is Converge able to statically determine namespaces, but variable names can be α-renamed without affecting the programs semantics. This is a significant deviation from the Python heritage. The quasi-quotes mechanism determines all bound variables in a quasi-quoted expression, and preemptively α-renames each bound variable to a guaranteed unique name that the user can not specify; all references to the variable are updated similarly. Thus the `x` within `[| x := 4 |]` will not cause variable capture to occur, and the variable `y` in function `g` will be set to `10`.

There is one potential catch: top-level definitions (all of which are assignments to a variable, although syntactic sugar generally obscures this fact) can not be α-renamed without affecting the programs semantics. This is because Converge's dynamic typing means that referencing a slot within a module can not generally be statically checked at compile-time. Thus α-renaming top-level definitions would almost certainly lead to run-time 'slot missing' exceptions being raised as the user attempts to reference a definition *`D`* within a module. Although the current compiler does not catch this case, since the user is unlikely to have cause to quasi-quote top-level definitions, barring it should be of little practical consequence.

Whilst the above rules explain the most important of Converge's scoping rules in the presence of quasi-quotes, upcoming sections add extra detail to the basic scoping rules explained in this section.

4.2.6. The `CEI` interface

At various points when compile-time meta-programming, one needs to interact with the Converge compiler. The Converge compiler is entirely contained within a package called `Compiler` which

[4] This terminology is borrowed from TH, but with a much different implementation.

is available to every Converge program. The `CEI` module within the `Compiler` package is the officially sanctioned interface to the Compiler, and can be imported with `import Compiler.CEI`.

ITree functions

Although the quasi-quotes mechanism allows the easy, and safe, creation of many required ITree's, there are certain legal ITree's which it can not express. Most such cases come under the heading of 'create an arbitrary number of *X*' e.g. a function with an arbitrary number of parameters, or an `if` expression with an arbitrary number of `elif` clauses. In such cases the `CEI` interface presents a more traditional meta-programming interface to the user that allows ITree's that are not expressible via quasi-quotes to be built. The downside to this approach is that recourse to the manual is virtually guaranteed: the user needs to know the name of the ITree element(s) required (each element has a corresponding function with a lower case name and a prepended 'i' in the `CEI` interface e.g. `ivar`), what the functions requirements are etc. Fortunately this interface needs to be used relatively infrequently; all uses of it in this thesis will be explicitly explained.

Names

Section 4.2.3 showed that the Converge compiler sometimes uses names for variables that the user can not specify using concrete syntax. The same technique is used by the quasi-quote mechanism to α-rename variables to ensure that variable capture does not occur. However one of the by-products of the arbitrary ITree creating interface provided by the `CEI` interface is that the user is not constrained by Converge's concrete syntax; potentially they could create variable names which would clash with the 'safe' names used by the compiler. To ensure this does not occur, the `CEI` interface contains several functions – similar to those in recent versions of TH – related to names which the user is forced to use; these functions guarantee that there can be no inadvertent clashes between names used by the compiler and by the user.

In order to do this, the `CEI` interface deals in terms of instances of the `CEI.Name` class. In order to create a variable, a slot reference etc, the user must pass an instance of this class to the relevant function in the `CEI` interface. New names can be created by one of two functions. The `name(`*x*`)` function validates *x*, raising an exception if it is invalid, and returning a `Name` otherwise. The `fresh_name` function guarantees to create a unique `Name` each time it is called (this is the interface used by the quasi-quotes mechanism). This allows e.g. variable names to be created safely with the idiom `var := CEI.ivar(CEI.name("var_name"))`. `fresh_name` takes an optional argument *x* which, if present, is incorporated into the generated name whilst still guaranteeing the uniqueness of the resulting name; this feature aids debugging by allowing the user to trace the origins of a fresh

name. Note that the `name` interface opens the door for dynamic scoping (see section 4.2.8).

4.2.7. Lifting values

When meta-programming, one often needs to take a normal Converge value (e.g. a string) and obtain its ITree equivalent: this is known as *lifting* a value.

Consider a debugging function `log` which prints out the debug string passed to it; this function is called at compile-time so that if the global `DEBUG_BUILD` variable is set to `fail` there is no run-time penalty for using its facility. The `log` function is thus a safe means of performing what is often termed 'conditional compilation'. Noting that `pass` is the Converge no-op, a first attempt at such a function is as follows:

```
func log(msg):
  if DEBUG_BUILD:
    return [| Sys.println(msg) |]
  else:
    return [| pass |]
```

This function fails to compile: the reference to the `msg` variable causes the Converge compiler to raise the error:

```
Var 'msg' is not in scope when in quasi-quotes (consider using
$<<CEI.lift(msg)>>).
```

Rewriting the offending piece of code to the following gives the correct solution:

```
return [| Sys.println($<<CEI.lift(x)>>) |]
```

What has happened here is that the string value of `msg` is transformed by the `lift` function into its abstract syntax equivalent. Constants are automatically lifted by the quasi-quotes mechanism: the two expressions `[| $<<CEI.lift("str")>> |]` and `[| "str" |]` are therefore equivalent.

Converge's refusal to lift the raw reference to `msg` in the original definition of `log` is a significant difference from TH, whose scoping rules would have caused `msg` to be lifted without an explicit call to `CEI.lift`. To explain this difference, assume the `log` function is rewritten to include the following fragment:

```
return [|
  msg := "Debug: " + $<<CEI.lift(msg)>>
  Sys.println(msg)
|]
```

In a sense, the quasi-quotes mechanism can be considered to introduce its own block: the assignment to the `msg` variable forces it to be local to the quasi-quote block. This needs to be the case since the alternative behaviour is nonsensical: if the assignment referenced to the `msg` variable outside the quasi-quotes then what would the effect of splicing in the quasi-quoted expression to a different context be? The implication of this is that referencing a variable within quasi-quotes would have a significantly different meaning if the variable had been assigned to within the quasi-quotes or outside

it. Whilst it is easy for the Converge compiler writer to determine that a given variable was defined outside the quasi-quotes and should be automatically lifted in (or vice versa), from a user perspective TH's behaviour can be unnecessarily confusing. Converge's quasi-quote mechanism originally had the same behaviour in this respect as TH, but this resulted in fragile and hard to follow code. To avoid such problems, Converge forces variables defined outside of quasi-quotes to be explicitly lifted into it. This also maintains a simple symmetry with Converge's main scoping rules: assigning to a variable in a block makes it local to that block.

4.2.8. Dynamic scoping

Sometimes the quasi-quote mechanisms automatic α-renaming of variables is not what is needed. For example consider a function `swap(x, y)` which should swap the values of the two variables passed as strings in its parameters. In such a case, we *want* the result of the splice to capture the variables in the spliced environment. Because the quasi-quotes mechanism only renames variables which it can determine statically at compile time, any variables created via the idiom `CEI.ivar(CEI.name(x))` and spliced into the quasi-quotes will not be renamed. The following succinct definition of `swap` takes advantage of this fact:

```
func swap(x, y):
  x_var := CEI.ivar(CEI.name(x))
  y_var := CEI.ivar(CEI.name(y))
  return [|
    temp := $<<x_var>>
    $<<x_var>> := $<<y_var>>
    $<<y_var>> := temp
  |]
```

Note that the variable `temp` within the quasi-quotes *will* be α-renamed and thus will be effectively invisible to the code that it is spliced into, but that the two variables referred to by `x` and `y` will be scoped by their splice location. The `swap` function can be used thus:

```
a := 10
b := 20
$<<swap("a", "b")>>
```

Dynamic scoping also tends to be useful when a quasi-quoted function is created piecemeal with many separate quasi-quote expressions. In such a case, variable references can only be resolved successfully when all the resulting ITree's are spliced together since references to the function's parameters and so on will not be determined until that point. Since it is highly tedious to continually write `CEI.ivar(CEI.name("foo"))`, Converge provides the special syntax `&foo` which is equivalent. Notice that this notation prefixes a variable name, irrespective of the value it contains. Thus it would not be possible to rewrite parts of the `swap` function as e.g. `x_var := &x`.

4.2.9. Forward references and splicing

In section 4.2.3 we saw that when a splice annotation outside quasi-quotes is encountered, a temporary module is created which contains all the definitions up to, but excluding, the definition holding the splice annotation. This is a very useful feature since compile-time functions used only in one module can be kept in that module. However this introduces a real problem involving forward references. A forward reference is defined to be a reference to a definition within a module, where the reference occurs at an earlier point in the source file than the definition. If a splice annotation is encountered and compiles a subset of the module, then some definitions involved in forward references may not be included: thus the temporary module will fail to compile, leading to the entire module not compiling. Worse still, the user is likely to be presented with a highly confusing error telling them that a particular reference is undefined when, as far as they are concerned, the definition is staring at them within their text editor.

Consider the following contrived example:

```
func f1(): return [| 7 |]

func f2(): x := f4()

func f3(): return $<<f1()>>

func f4(): pass
```

If `f2` is included in the temporary module created when evaluating the splice annotation in `f3`, then the forward reference to `f4` will be unresolvable.

The solution taken by Converge ensures that, by including only a minimal subset of definitions in the temporary module, most forward references do not raise a compile-time error. We saw in section 4.2.5 that the quasi-quotes mechanism uses Converge's statically determined namespaces to calculate bound variables. That same property is now used to determine an expressions free variables.

When a splice annotation is encountered, the Converge compiler does not immediately create a temporary module. First it calculates the splice expressions free variables; any previously encountered definition which has a name in the set of free variables is added to a set of definitions to include. These definitions themselves then have their free variables calculated, and again any previously encountered definition which has a name in the set of free variables is added to the set of definitions to include. This last step is repeated until an iteration adds no new definitions to the set. At this point, Converge then goes back in order over all previously encountered definitions, and if the definition is in the list of definitions to include, it is added to the temporary module. Recall that the order of definitions in a Converge file can be significant (see section 4.1.4): this last stage ensures that definitions are not reordered in the temporary module. Note also that free variables which genuinely do not refer to any definitions (i.e. a mistake on the part of the programmer) will pass through this scheme unmolested

and will raise an appropriate error when the temporary module is compiled.

Using this method, the temporary module that is created and evaluated for the example looks as follows:

```
func f1(): return [| 7 |]
func $$splice$$(): return f1()
```

There are thus no unresolvable forward references in this example.

There is a secondary, but significant, advantage to this method: since it reduces the number of definitions in temporary modules it can lead to an appreciable saving in compile time, especially in files containing multiple splice annotations.

4.2.10. Compile-time meta-programming in use

In this chapter thus far we have seen several uses of compile-time meta-programming. There are many potential uses for this feature, many of which are too involved to detail in the available space. For example, one of the most exciting uses of the feature has been in conjunction with Converge's extendable syntax feature (see section 4.4), allowing powerful DSLs to be expressed in an arbitrary concrete syntax. One can see similar work involving DSLs in e.g. [SCK03, COST04].

In this section I show two seemingly mundane uses of compile-time meta-programming: conditional compilation and compile-time optimization. Although mundane in some senses, both examples open up potential avenues not currently available to other dynamically typed OO languages.

Conditional compilation

Whereas languages such as Java attempt to insulate their users from the underlying platform an application is running on, languages such as Python and Ruby allow the user access to many of the lower-level features the platform provides. Many applications rely on such low-level features being available in some fashion. However for the developer who has to provide access to such features a significant problem arises: how does one sensibly provide access to such features when they are available, and to remove that access when they are unavailable?

The `log` function on page 76 was a small example of conditional compilation. Let us consider a simple but realistic example that is more interesting from an OO perspective. The POSIX `fcntl` (File CoNTrol) feature provides low-level control of file descriptors, for example allowing file reads and writes to be set to be non-blocking; it is generally only available on UNIX-like platforms. Assume that we wish to provide some access to the `fcntl` feature via a method within file objects; this method will need to call the raw function within the provided `fcntl` module iff that module is available on the current platform.

In Python for example, there are two chief ways of doing this. The first mechanism is for a `File` class to defer checking for the existence of the `fcntl` module until the `fcntl` method is called, raising an exception if the feature is not detected in the underlying platform. Callers who wish to avoid use of the `fcntl` method on platforms lacking this feature must catch the appropriate exception. This rather heavy handed solution goes against the spirit of *duck typing* [TH00], a practise prevalent in languages such as Ruby and Python. In duck typing, one checks for the presence of a method(s) which appear to satisfy a particular API without worrying about the type of the object in question. For example, for a method that requires a file object to read from, rather than testing that the object passed is an instance of the `File` class, the method simply checks that the input object has a `read` slot. Whilst perhaps unappealing from a theoretical point of view, this approach is common in practise due to the low-cost flexibility it leads to. To ensure that duck typing is possible in our `fcntl` example, we are forced to use exception handling and the dynamic selection of an appropriate sub-class:

```
try:
  import fcntl
  _HAVE_FCNTL = True
except exceptions.ImportError:
  _HAVE_FCNTL = False


class Core_File:
  # ...

if _HAVE_FCNTL:
  class File(Core_File):
    def fcntl(op, arg):
      return fcntl.fcntl(self.fileno(), op, arg)
else:
  class File(Core_File):
    pass
```

Whilst this allows for duck typing, this idiom is far from elegant. The splitting of the `File` class into a core component and sub-classes to cope with the presence of the `fcntl` functionality is somewhat distasteful. This example is also far from scalable: if one wishes to use the same approach for more features in the same class then the resultant code is likely to be highly fragile and complex.

Although it appears that the above idiom can be encoded largely ‘as is’ in Converge, we immediately hit a problem due to the fact that module imports are statically determined. Thus a direct Converge analogue would compile correctly only on platforms with a `fcntl` module. However by using compile-time meta-programming one can create an equivalent which functions correctly on all platforms and which cuts out the ugly dynamic sub-class selection.

The core feature here is that class fields are permissible splice locations (see section 4.2.3). A splice which returns an ITree that is a function will have that function incorporated into the class; if the splice returns `pass` as an ITree then the class is unaffected. So at compile-time we first detect for the presence of a `fcntl` module (the `VM.loaded_module_names` function returns a list con-

taining the names of all loaded modules); if it is detected, we splice in an appropriate `fcntl` method otherwise we splice in the no-op. This example make use of two hitherto unencountered features. Firstly, using an `if` construct as an expression requires a different syntax (to work around parsing limitations associated with indentation based grammars); the construct evaluates to the value of the final expression in whichever branch is taken, failing if no branch is taken. Secondly the modified Oxford quotes `[d| ...|]` – *declaration quasi-quotes* – act like normal quasi-quotes except they do not α-rename variables; declaration quotes are typically most useful at the top-level of a module. The Converge example is as follows:

```
$<<if VM.loaded_module_names().contains("FCntl") {
  [d|
    import FCntl
    _HAVE_FCNTL := 1
  |]
}
else {
  [d| _HAVE_FCNTL := 0 |]
}>>

class File:
  $<<if _HAVE_FCNTL {
    [d|
      func fcntl(op, arg):
        return FCntl.fcntl(self.fileno(), op, arg)
    |]
  }
  else {
    [| pass |]
  }>>
```

Although this example is simplistic in many ways, it shows that compile-time meta-programming can provide a conceptually neater solution than any purely run-time alternative since it allows related code fragments to be kept together. It also provides a potential solution to related problems. For example portability related code in dynamically typed OO languages often consists of many `if` statements which perform different actions depending on a condition which relates to querying the platform in use. Such code can become a performance bottleneck if called frequently within a program. The use of compile-time meta-programming can lead to a zero-cost run-time overhead. Perhaps significantly, the ability to tune a program at compile-time for portability purposes is the largest single use of the C preprocessor [EBN02] – compile-time meta-programming of the sort found in Converge not only opens similar doors for dynamically typed OO languages, but allows the process to occur in a far safer, more consistent and more powerful environment than the C preprocessor.

4.2.11. Run-time efficiency

In this section I present the Converge equivalent of the TH compile-time `printf` function given in [SJ02]. Such a function takes a format string such as `"%s has %d %s"` and returns a quasi-quoted function which takes an argument per '`%`' specifier and intermingles that argument with the main text

string. For our purposes, we deal with decimal numbers `%d` and strings `%s`.

The motivation for a TH `printf` is that such a function is not expressible in base Haskell. Although Converge functions can take a variable number of arguments (as Python, but unlike Haskell), having a compile-time version still has two benefits over its run-time version: any errors in the format string are caught at compile-time; an efficiency boost.

This example assumes the existence of a function `split_format` which given a string such as `"%s has %d %s"` returns a list of the form `[PRINTF_STRING, " has ", PRINTF_INT, " ", PRINTF_STRING]` where `PRINTF_STRING` and `PRINTF_INT` are constants.

First we define the main `printf` function which creates the appropriate number of parameters for the format string (of the form `p0`, `p1` etc.). Parameters must be created by the `CEI` interface. An `iparam` has two components: a variable, and a default value (the latter can be set to `null` to signify the parameter is mandatory and has no default value). `printf` then returns an anonymous quasi-quoted function which contains the parameters, and a spliced-in expression returned by `printf_expr`:

```
func printf(format):
  split := split_format(format)
  params := []
  i := 0
  for part := split.iterate():
    if part == PRINTF_INT | part == PRINTF_STRING:
      params.append(CEI.iparam(CEI.ivar(CEI.name("p" + i.to_str())), null))
      i += 1
  return [|
    func ($<<params>>):
      Sys.println($<<printf_expr(split, 0)>>)
  |]
```

`printf_expr` is a recursive function which takes two parameters: a list representing the parts of the format string yet to be processed; an integer which signifies which parameter of the quasi-quoted function has been reached.

```
func printf_expr(split, param_i):
  if split.len() == 0:
    return [| "" |]
  param := CEI.ivar(CEI.name("p" + param_i.to_str()))
  if split[0].conforms_to(String):
    return [| $<<CEI.lift(split[0])>> + $<<printf_expr(split[1:], param_i)>> |]
  elif split[0] == PRINTF_INT:
    return [| $<<param>>.to_str() + $<<printf_expr(split[1:], param_i + 1)>> |]
  elif split[0] == PRINTF_STRING:
    return [| $<<param>> + $<<printf_expr(split[1 : ], param_i + 1)>> |]
```

`printf_expr` recursively calls itself, each time removing the first element from the format string list, and incrementing the `param_i` variable iff a parameter has been processed. This latter condition is invoked when a string or integer '%' specifier is encountered; raw text in the input is included as is, and as it does not involve any of the functions' parameters, does not increment `param_i`. When the format string list is empty, the recursion starts to unwind.

When the result of `printf_expr` is spliced into the quasi-quoted function, the dynamically scoped references to parameter names in `printf_expr` become bound to the quasi-quoted functions' parameters. As an example of calling this function, `$<<printf("%s has %d %s")>>` generates the following function:

```
func (p0, p1, p2):
  Sys.println(p0 + " has " + p1.to_str() + " " + p2 + "")
```

so that evaluating the following:

```
$<<printf("%s has %d %s")>>("England", 39, "traditional counties")
```

results in `England has 39 traditional counties` being printed to screen.

This definition of `printf` is simplistic and lacks error reporting, partly because it is intended to be written in a similar spirit to its TH equivalent. Converge comes with a more complete compile-time `printf` function as an example, which uses an iterative solution with more compile-time and run-time error-checking. Simple benchmarking of the latter function reveals that it runs nearly an order of magnitude faster than its run-time equivalent[5] – a potentially significant gain when a tight loop repeatedly calls `printf`.

4.2.12. Compile-time meta-programming costs

Although compile-time meta-programming has a number of benefits, it would be naïve to assume that it has no costs associated with it. However although Converge's features have been used to build several small programs, and two systems of several thousand lines of code each, it will require a wider range of experience from multiple people working in different domains to make truly informed comments in this area.

One thing is clear from experience with LISP: compile-time meta-programming in its rawest form is not likely to be grasped by every potential developer [Que96]. To use it to its fullest potential requires a deeper understanding of the host language than many developers are traditionally used to; indeed, it is quite possible that it requires a greater degree of understanding than many developers are prepared to learn. Whilst features such as extendable syntax (see section 4.4) which are layered on top of compile-time meta-programming may smooth off many of the usability rough edges, fundamentally the power that compile-time meta-programming extends to the user comes at the cost of increased time to learn and master.

In Converge one issue that arises is that code which continually dips in and out of the meta-programming constructs can become rather messy and difficult to read on screen if over-used in any one area of code. This is due in no small part to the syntactic considerations that necessitate a

[5] This large differential is in part due to the fact that the current Converge VM imposes a relatively high overhead on function application.

move away from the clean Python-esque syntax to something closer to the C family of languages. It is possible that the integration of similar features into other languages with a C-like syntax would lead to less obvious syntactic seams.

4.2.13. Error reporting

Perhaps the most significant unresolved issue in compile-time meta-programming systems relates to error reporting [COST04]. Although Converge does not have complete solutions to all issues surrounding error reporting, it does contain some rudimentary features which may give insight into the form of more powerful error reporting features both in Converge and other compile-time meta-programming systems.

The first aspect of Converge's error reporting facilities relates to exceptions. When an exception is raised, detailed stack traces are printed out allowing the user to inspect the sequence of calls that led to the exception being raised. These stack traces differ from those found in e.g. Python in that each level in the stack trace displays the file name, line number and column number that led to the error. Displaying the column number allows users to make use of the fine-grained information to more quickly narrow down the precise source of an exception. Converge is able to display such detailed information because when it parses text, it stores the file name, line number and column number of each token. Tokens are ordered into parse trees; parse trees are converted into ASTs; ASTs are eventually converted into VM instructions. At each point in this conversion, information about the source code elements is retained. Thus every VM instruction in a binary Converge program has a corresponding debugging entry which records which file, line number and column number the VM instruction relates to. Whilst this does require more storage space than simpler forms of error information, the amount of space required is insignificant when the vast storage resources of modern hardware are considered.

Whilst the base language needs to record the related source offset of each VM instruction, the source file a VM instruction relates to is required only due to compile-time meta-programming. Consider a file `A.cv`:

```
func f():
  return [| 2 + "3" |]
```

and a file B.cv:

```
import A

func main():
  $<<A.f()>>
```

When the quasi-quoted code in `A.f` is spliced in, and then executed an exception will be raised about the attempted addition of an integer and a string. The exception that results from running `B` is as

follows:

```
Traceback (most recent call last):
  File "A.cv", line 2, column 13, in main
Type_Exception: Expected instance of Int, but got instance of String.
```

The fact that the `A` module is pinpointed as the source of the exception may initially seem surprising, since the code raising the exception will have been spliced into the `B` module. This is however a deliberate design choice in Converge. Although the code from `A.f` has been spliced into `B.main`, when `B` is run the quasi-quoted code retains the information about its original source file, and not its splice location. To the best of my knowledge, this approach to error reporting in the face of compile-time meta-programming is unique. As points of comparison, TH is not able to produce any detailed information during a stack-trace and SCM Scheme [Jaf03] pinpoints the source file and line number of run-time errors as that of the macro call site. In SCM Scheme if the code that a macro produces contains an error, all the user can work out is which macro would have led to the problem — the user has no way of knowing which part of the macro may be at fault.

Converge allows customization of the error-reporting information stored about a given ITree. Firstly Converge adds a feature not present in TH: nested quasi-quotes. An outer quasi-quote returns the ITree of the code which would create the ITree of the nested quasi-quote. For example the following nested code:

```
Sys.println([| [| 2 + "3" |] |].pp())
```

results in the following output:

```
CEI.ibinary_add(CEI.iint(2, "ct.cv", 484), CEI.istring("3", "ct.cv", 488),
  "ct.cv", 486)
```

Nested quasi-quotes provide a facility which allows users to analyse the ITrees that plain quasi-quotes generate: one can see in the above that each ITree element contains a reference to the file it was contained within (`ct.cv` in this case) and to the offset within the file (`484` and so on). The `CEI` module provides a function `src_info_to_var` which given an ITree representing quasi-quoted code copies the ITree[6] replacing the source code file and offsets with variables `src_file` and `src_offset`. This new ITree is then embedded in a quasi-quoted function which takes two arguments `src_file` and `src_offset`. When the user splices in and then calls this function, they update the ITree's relation to source code files and offsets. Using this function in the following fashion:

```
Sys.println(CEI.src_info_to_var([| [| 2 + "3" |] |]).pp())
```

results in the following output:

```
unbound_func (src_file, src_offset){
    return CEI.ibinary_add(CEI.iint(2, src_file, src_offset),
```

[6] In the current implementation, the `src_info_to_var` actually mutates ITrees, but for reasons explained in section 4.3.3 this will not be possible in the future.

```
    CEI.istring("3", src_file, src_offset), src_file, src_offset)
}
```

In practice when one wishes to customise the claimed location of quasi-quoted code, the nested quasi-quotes need to be cancelled out by a splice. For example, to change source information to be offset 77 in the file `nt.cv` we would use the following code:

```
return $<<CEI.src_info_to_var([| [| 2 + "3" |] |], "nt.cv", 77>>
```

Whilst this appears somewhat clumsy, it is worth noting that by adding only the simple concept of nested quasi-quotes, complex manipulation of the meta-system is possible.

Converge's current approach is not without its limitations. Its chief problem is that it can only relate one source code location to any given VM instruction. There is thus an 'either / or' situation in that the user can choose to record either the definition point of the quasi-quoted code, or change it to elsewhere (e.g. to record the splice point). It would be of considerable benefit to the user if it is possible to record all locations which a given VM instruction relates to. Assuming the appropriate changes to the compiler and VM, then the only user-visible change would be that `src_info_to_var` would append `src_file` and `src_offset` information within a given ITree, rather than overwriting the information it already possessed.

4.2.14. Related work

Perhaps surprisingly, the template system in C++ has been found to be a fairly effective, if crude, mechanism for performing compile-time meta-programming [Vel95, COST04]. The template system can be seen as an ad-hoc functional language which is interpreted at compile-time. However this approach is inherently limited compared to the other approaches described in this section.

The dynamic OO language Dylan – perhaps one of the closest languages in spirit to Converge – has a similar macro system [BP99] to Scheme. In both languages there is a dichotomy between macro code and normal code; this is particularly pronounced in Dylan, where the macro language is quite different from the main Dylan language. As explained in the introduction, languages such as Scheme need to be able to identify macros as distinct from normal functions (although Bawden has suggested a way to make macros first-class citizens [Baw00]). The advantage of explicitly identifying macros is that there is no added syntax for calling a macro: macro calls look like normal function calls. Of course, this could just as easily be considered a disadvantage: a macro call is in many senses rather different than a function call. In both schemes, macros are evaluated by a macro expander based on patterns – neither executes arbitrary code during macro expansion. This means that their facilities are limited in some respects – furthermore, overuse of Scheme's macros can lead to complex and confusing 'language towers' [Que96]. Since it can execute arbitrary code at compile-time Converge does not suffer from the same macro expansion limitations, but whether moving the syntax burden

from the point of macro definition to call site will prevent the comprehension problems associated with Scheme is an open question.

Whilst there are several proposals to add macros of one sort or another to existing languages (e.g. for Java alone one can find proposals from Bachrach and Playford's Java macro system [BP01] and Tatsubori *et. al* [TCIK99]), the lack of integration with their target language thwarts practical take-up.

Nemerle [SMO04] is a statically typed OO language, in the Java / C# vein, which includes a macro system mixing elements of Scheme and TH's systems. Macros are not first-class citizens, but AST's are built in a manner reminiscent of TH. The disadvantage of this approach is that calculations often need to be arbitrarily pushed into normal functions if they need to be performed at compile-time.

Comparisons between Converge and TH have been made throughout this section – I do not repeat them here. MetaML is TH's most obvious forebear and much of the terminology in Converge has come from MetaML via TH. MetaML differs from TH and Converge by being a multi-stage language. Using its 'run' operator, code can be constructed and run (via an interpreter) at run-time, whilst still benefiting from MetaML's type guarantees that all generated programs are type-correct. The downside of MetaML is that new definitions can not be introduced into programs. The MacroML proposal [GST01] aims to provide such a facility but – in order to guarantee type-correctness – forbids inspection of code fragments which limits the features expressivity.

Significantly, with the exception of Dylan, I know of no other dynamically typed OO language in the vein of Converge which supports any form of compile-time meta-programming natively.

4.3. Implications for other languages and their implementations

I believe that Converge shows that compile-time meta-programming facilities can be added in a seamless fashion to a dynamically-typed OO language and that such facilities provide useful functionality not available previously in such languages. In this section I first pinpoint the relatively minimal requirements on language design necessary to allow the safe and practical integration of compile-time meta-programming facilities. Since the implementation of such a facility is quite different from a normal language compiler, I then outline the makeup of the Converge compiler to demonstrate how an implementation of such features may look in practice. Finally I discuss the requirements on the interface between user code and the languages' compiler.

4.3.1. Language design implications

Although Converge's compile-time meta-programming facilities have benefited slightly from being incorporated in the early stages of the language design, there is surprisingly little coupling between the

base language and the compile-time meta-programming constructs. The implications on the design of similar languages can thus be boiled down to the following two main requirements:

1. It must be possible to determine all namespaces statically, and also to resolve variable references between namespaces statically. This requirement is vital for ensuring that scoping rules in the presence of compile-time meta-programming are safe and practical (see section 4.2.5). Slightly less importantly, this requirement also allows functions called at compile-time to be stored in the same module as splices which call them whilst avoiding the forward reference problem (see section 4.2.9).

2. Variables within namespaces other than the outermost module namespace must be α-renameable without affecting the programs semantics. This requirement is vital to avoid the problem of variable capture.

Note that there is an important, but non-obvious, corollary to the second point: when variables and slot names overlap then α-renaming can not take place. In section 4.2.5 we saw that, in Converge, top-level module definitions can not be renamed because the variable names are also the slot names of the module object. Since Converge forces all accesses of class fields via the `self` variable, Converge neatly sidesteps another potential place where this problem may arise. Fortunately, whilst many statically typed languages allow class fields to be treated as normal variables (i.e. making the `self.` prefix optional) most dynamically typed languages take a similar approach to Converge and should be equally immune to this issue in that context.

Only two constructs in Converge are dedicated to compile-time meta-programming. Practically speaking both constructs would need to be added to other languages:

1. A splicing mechanism. This is vital since it is the sole user mechanism for evaluating expressions at compile-time.

2. A quasi-quoting mechanism to build up AST's. Although such a facility is not strictly necessary, experience suggests that systems without such a facility tend towards the unusable [WC93].

4.3.2. Compiler structure

Typical language compilers follow a predictable structure: a parser creates a parse tree; the parse tree may be converted into an AST; the parse tree or AST is used to generate target code (be that VM bytecode, machine code or an intermediate language). Ignoring optional components such as optimizers, one can see that normal compilers need only two or three major components (depending on

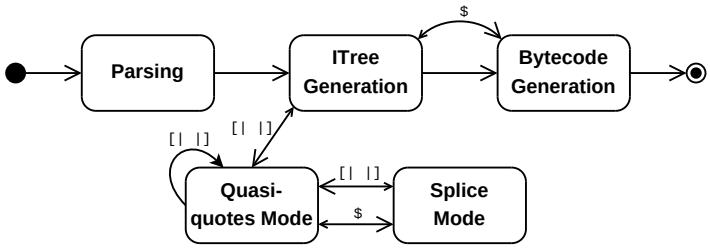


Figure 4.2.: Converge compiler states.

the inclusion or omission of an explicit AST generator). Importantly the process of compilation involves an entirely linear data flow from one component to the next. Compile-time meta-programming however necessitates a different compiler structure, with five major components and a non-linear data flow between its components. In this section I detail the structure of the Converge compiler, which hopefully serves as a practical example for compilers for other languages. Whether existing language compilers can be retro-fitted to conform to such a structure, or whether a new compiler would need to be written can only be determined on a case-by-case basis; however in either case this general structure serves as an example.

Figure 4.2 shows a (slightly non-standard) state-machine representing the most important states of the Converge compiler. Large arrows indicate a transition between compiler states; small arrows indicate a corresponding return transition from one state to another (in such cases, the compiler transitions to a state to perform a particular action and, when complete, returns to its previous state to carry on as before). Each of these states also corresponds to a distinct component within the compiler.

The stages of the Converge compiler can be described thus:

1. **Parsing.** The compiler parses an input file into a parse tree. Once complete, the compiler transitions to the next state.

2. **ITree Generation.** The compiler converts the parse tree into an ITree; this stage continues until the complete parse tree has been converted into an ITree. Since ITree's are exposed directly to the user, it is vital that the parse tree is converted into a format that the user can manipulate in a practical manner[7].

    a) **Splice mode / bytecode generation.** When it encounters a splice annotation in the parse tree, the compiler creates a temporary ITree representing a module. It then transitions temporarily to the bytecode generation state to compile. The compiled temporary module is injected into the running VM and executed; the result of the splice is used in place of the annotation itself when creating the ITree.

[7] An early, and naïve, prototype of the Converge compiler exposed parse trees directly to the user. This quickly lead to spaghetti code.

b) **Quasi-quotes mode / splice mode.** As the ITree generator encounters quasi-quotes in the parse tree, it transitions to the quasi-quote mode. Quasi-quote mode creates an ITree respecting the scoping rules and other features of section 4.2.5.

   If, whilst processing a quasi-quoted expression, a splice annotation is encountered, the compiler enters the splice mode state. In this state, the parse tree is converted to an ITree in a manner mostly similar to the normal ITree Generation state. If, whilst processing a splice annotation, a quasi-quoted expression is encountered, the compiler enters the quasi-quotes mode state again. If, whilst processing a quasi-quoted expression, a nested quasi-quoted expression is encountered the compiler enters a new quasi-quotes mode.

3. **Bytecode generation.** The complete ITree is converted into bytecode and written to disk.

4.3.3. Compiler interface

Converge provides the `CEI` module which user code can use to interact with the language compiler. Similar implementations will require a similar interface to allow two important activities:

1. The creation of fresh variable names (see section 4.2.6). This is vital to provide a mechanism for the user to generate unique names which will not clash with other names, and thus will prevent unintended variable capture. To ensure that all fresh names are unique, most practical implementations will probably choose to inspect and restrict the variable names that a user can use within ITree's via an analogue to Converge's `name` interface; this is purely to prevent the user inadvertently using a name which the compiler has guaranteed (or might in the future guarantee) to be unique.

2. The creation of arbitrary AST's. Since it is extremely difficult to make a quasi-quote mechanisms completely general without making it prohibitively complex to use, there are likely to be valid AST's which are not completely expressible via the quasi-quotes mechanism. Therefore the user will require a mechanism to allow them to create arbitrary AST fragments via a more-or-less traditional meta-programming interface [WC93].

Abstract syntax trees

One aspect of Converge's design that has proved to be more important than expected, is the issue of AST design. In typical languages, the particular AST used by the compiler is never exposed in any way to the user. Even in Converge, for many users the particulars of the ITree's they generate via the quasi-quotes mechanism are largely irrelevant. However those users who find themselves needing to

generate arbitrary ITree's via the `CEI` interface, and especially those (admittedly few) who perform computations based on ITree's, find themselves disproportionately affected by decisions surrounding the ITree's representation.

At the highest level, there are two main choices surrounding AST's. Firstly, should it be represented as an homogeneous, or heterogeneous tree? Secondly should the AST be mutable or immutable? The first question is relatively easy to answer: my experience suggests that homogeneous trees are not a practical representation of a rich AST. Whilst parse trees are naturally homogeneous, the conversion to an AST leads to a more structured and detailed tree that is naturally heterogeneous.

Let us then consider the issue of AST mutability. Initially Converge supported mutable AST's; whilst this feature has proved useful from time to time, it has also proved somewhat more dangerous than expected. This is because one often naturally creates references to a given AST fragment from more than one node. Changing a node which is referenced by more than one other node can then result in unexpected changes, which all too frequently manifest themselves in hard to debug ways. Since it is not possible to check for this problem in the general case, the user is ultimately responsible for ensuring it does not occur; in practise this has proved to be unrealistic, and gradually all ITree-mutating code has been banished from Converge code. Future versions of Converge will force ITree's to be immutable, and I would recommend other languages consider this point carefully.

4.4. Syntax extension for DSLs

Converge has a simple but powerful facility allowing users to embed arbitrary sequences of tokens within Converge source files. At compile-time these tokens are passed to a designated user function, which is expected to return an AST. This allows the user to extend the language's syntax in an arbitrary fashion, meaning that DSLs can be embedded within normal Converge code.

A DSL fragment is an indented block containing an arbitrary sequence of tokens. The DSL block is introduced by a variant on the splice syntax `$< expr >` where *`expr`* should evaluate to a function (the *DSL implementation function*). The DSL function will be called at compile-time with a list of tokens, and is expected to return an AST which will replace the DSL block in the same way as a normal splice. Compile-time meta-programming is thus the mechanism which facilitates embedding DSLs.

An example DSL fragment is as follows. Colloquially this block is referred to as 'a `TM.model_class`':

```
import TM.TM

$<TM.model_class>:
  abstract class ML1_Element {
```

```
    name : String;

    inv nonempty_name:
      name != null and name.len() > 0
  }
```

Note that the DSL fragment is written in an entirely different syntax than Converge itself.

Currently DSL blocks are automatically tokenized by the Converge compiler using its default tokenization rules — this is not a fundamental requirement of the technique, but a peculiarity of the current implementation. More sophisticated implementations might choose to defer tokenization to the DSL implementation function. However using the Converge tokenizer has the advantage that normal Converge code can be embedded inside the DSL itself assuming an appropriate link from the DSLs grammar to the Converge grammar.

4.4.1. DSL implementation functions

DSL implementation functions follow a largely similar sequence of steps in order to translate the input tokens into an ITree:

1. Alter the input tokens as necessary. Since DSLs often use keywords that are not part of the main Converge grammar, such alterations mostly take the form of replacing `ID` tokens with specific keyword tokens.

2. Parse the input tokens according to the DSL's grammar.

3. Traverse the parse tree, translating it into an ITree.

Section 4.5 explores these steps in greater detail via a concrete example.

4.4.2. Adding a `switch` statement

In this subsection, I detail a simple Converge DSL which allows `switch` statements to be embedded in Converge code. A simple example of the `switch` DSL in use is as follows:

```
$<switch>:
  switch x:
    case 2:
      Sys.println("2")
    case 4:
      Sys.println("4")
    default:
      Sys.println("default")
```

Pre-parsing and grammar

Before the `switch` DSL can parse its input, it first iterates through the input tokens searching for tokens which have type `ID` and value any of `switch`, `case`, or `default`. Such tokens are replaced

by a keyword token, whose type is the `ID`'s value. The grammar for this DSL is as follows:

```
switch  ::= "SWITCH" "ID" ":" "INDENT" clauses default "DEDENT"
clauses ::= { clause "NEWLINE" }* clause
clause  ::= "CASE" expr ":" "INDENT" expr_body "DEDENT"
default ::= "NEWLINE" "DEFAULT" ":" "INDENT" expr_body "DEDENT"
        ::=
```

References to the `expr_body` grammar rule reference the main Converge grammar.

Traversing the parse tree

Since the `switch` DSL references the main Converge grammar, the DSL extends the Converge compiler (via the `IModule_Generator` module) itself, needing only to add simple traversal functions for the four grammar rules added by the DSL. The main part of a DSL implementation function is concerned with traversing the parse tree, and translating it into an appropriate ITree. The CPK provides a simple traversal class (essentially a Converge equivalent of that found in the SPARK parser [AH02]) which provides the basis for most such translations. Users need only subclass the `Traverser` class and create a function prefixed by `_t_`*`name`* for each rule in the grammar. The `Traverser` class provides a `preorder` function will traverse an input parse tree in preorder fashion, calling the appropriate `_t_`*`name`* function for each node encountered in the tree. Note that each `_t_`*`name`* function can choose whether to invoke the `preorder` rule on sub-nodes, or whether it is capable of processing the sub-nodes itself. The `IModule_Generator` class is a sub-class of the CPK's `Traverser` class. Although subclassing of large and complex classes is often thought of as being dangerous, the `IModule_Generator` module has been specifically designed with sub-classing of this sort in mind.

The complete code for the `switch` translation is as follows:

```
1  class Switch_Translator(IModule_Generator._IModule_Generator)
2    func _t_switch(node):
3      // switch ::= "SWITCH" "ID" ":" "INDENT" clauses default "DEDENT"
4      self._var := CEI.ivar(CEI.name(node[2].value))
5      clauses := self.preorder(node[5])
6      default := self.preorder(node[6])
7      return return CEI.iif(clauses, default)
8
9    func _t_clauses(node):
10     // clauses ::= { clause "NEWLINE" }* clause
11     //         ::=
12     i := 1
13     clauses := []
14     while i < node.len():
15       clauses.append(self.preorder(node[i]))
16       i += 2
17     return clauses
18
19   func _t_clause(node):
20     // clause ::= "CASE" expr ":" "INDENT" expr_body "DEDENT"
21     return CEI.iclause([| $<<self._var>> == $<<self.preorder( \
22       node[2])>> |], self.preorder(node[5]))
23
24   func _t_default(node):
```

```
25      // default ::= "NEWLINE" "DEFAULT" ":" "INDENT" expr_body "DEDENT"
26      //         ::=
27      if node.len() == 1:
28        return [| pass |]
29      else:
30        return self.preorder(node[5])
```

The translation is straightforward. Line 4 records the variable which is being 'switched'. Lines 5 – 7 translate the `switch` statement into a single `if` statement; the switches `default` clause becomes the `elif` clause. Lines 21 – 22 translate each clause in the `switch` statement into an ITree clause which compares the 'switched' variable with the value of the expression in the clause. Lines 27 – 30 translate the `default` clause; if no such clause is specified, the translation returns the `pass` no-op.

4.4.3. Related work

Real-world implementations of a similar concept are surprisingly rare. The Camlp4 pre-processor [dR03] allows the normal OCaml grammar to be arbitrarily extended, and is an example of a heterogeneous syntax extension system in that the system doing the extension is distinct from the system being extended. The MetaBorg system [BV04] is a heterogeneous system that can be applied to any language; more sophisticated than the Camlp4 pre-processor, from an external point of view it more closely resembles Converge's functionality, although the implementations and underlying philosophies are still very different.

I am currently aware of only two homogeneous syntax extension systems apart from Converge. Nemerle [SMO04] allows limited syntax extension via its macro scheme. The commercial XMF tool [CESW04] presents only a small core grammar, with many normal language concepts being grammar extensions on top of the core grammar. Grammar extensions are compiled down into XMF's AST. XMF is thus much closer in spirit to Converge, although the example grammar extensions available suggest that XMF's compile-time facilities may be less powerful than Converge's, seemingly being based on a simplified version of TH's features. If true, this may limit the complexity of the grammar extensions.

4.5. Modelling language DSL

This section presents an example of a Converge DSL *TM* for expressing typed modelling languages; modelling languages can be instantiated create models. In its current simplistic form, TM operates with a fixed number of meta-levels in that it defines modelling languages that can create models, but those models are terminal instances (in ObjVLisp's terminology) — that is, they can not be used to create new objects.

This section serves two purposes. Firstly it is an example of Converge's syntax extension system,

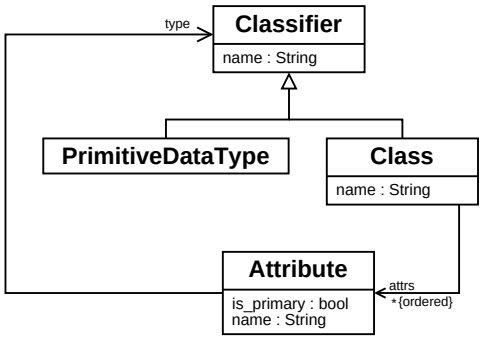


Figure 4.3.: ‘Simple UML’ model.

and fleshes out the method of section 4.4.1. Secondly the DSL in question is used in the remainder of this thesis.

4.5.1. Example of use

The TM DSL is housed within the package `TM`; the DSL implementation function `model_class` is contained within the `TM` module within the package. The following fragment uses the DSL to express a model of a simplified UML modelling language as shown in figure 4.3:

```
import TM.TM

$<TM.model_class>:
  abstract class Classifier {
    name : String;
  }

  class PrimitiveDataType extends Classifier { }

  class Class extends Classifier {
    attrs : Seq(Attribute);

    inv unique_names:
      attrs->forAll(a1 a2 |
        a1 != a2 implies a1.name != a2.name)
  }

  class Attribute {
    name : String;
    type : Classifier;
    is_primary : bool;
  }
```

Note that although this particular example shows a model of a modelling language, the DSL is capable of expressing any type of model — the example here is taken from section 5.3.5.

The `TM.model_class` DSL implementation function translates each class in the model into a function in Converge which creates model objects. As a useful convenience, each constructor function takes arguments which correspond to the order in which attributes are specified in the model class. If a model class has parents, their attributes come first, and so on recursively. Model objects can have their slots accessed by name. Note that since the modelling language is typed, setting attributes either

via the constructor function or through assigning to a slot forces the value to be of the correct type. Types can be of any model class (the DSL allows forward references), `int`, `String`, `bool` (where true and false are represented by 1 and 0 respectively), or sequences or sets of the preceding types. Note that sequences and sets can be nested arbitrarily. Model classes can contain invariants which are written in OCL; invariants are checked after an object has been initialized with values, and on every subsequent slot update. Currently only a subset of OCL 1.x is implemented, but the subset covers several different areas of OCL; implementing full OCL 1.x would be a relatively simple extension.

Assuming the above is held in a file `Simple_UML.cv`, one can then use the Simple UML modelling language to create models. The following example creates model classes `Dog` and `Person`, with `Dog` having an attribute `owner` of type `Person`:

```
person := Simple_UML.Class("Person")
dog := Simple_UML.Class("Dog")
dog.attrs.append(Simple_UML.Attribute("owner", person, 0))
```

One can arbitrarily manipulate models in the same way as standard objects:

```
dog.name := "Doggy"
```

Attempting to update a model in a way that would conflict with its type information results in an exception being raised. For example, attempting to assign an integer to the `Dog` model class' name raises the following exception:

```
Traceback (most recent call last):
  File "Ex1.cv", line 42, column 4, in main
  File "TM/TM.cv", line 162, column 5, in set_slot
Exception: Instance of 'Class' expected object of type 'String' for slot 'name'.
```

In similar fashion, if one violates the `unique_names` constraint by adding two attributes called `owner` to the `Dog` model class, the following exception is raised:

```
Traceback (most recent call last):
  File "Ex1.cv", line 45, column 17, in main
  File "TM/TM.cv", line 327, column 31, in append
  File "TM/TM.cv", line 407, column 3, in _class_class_check_invs
Exception: Invariant 'unique_names' violated.
```

As can be seen, the result of using the `TM.model_class` DSL is a natural embedding of an arbitrary modelling language within Converge. Furthermore the recording of type information using the modelling language DSL, allows the enforcement of such type information providing guarantees about models that would not have been the case if they were implemented as normal Converge classes.

In the following sections I outline how this DSL is implemented.

4.5.2. Data model

TM provides its own ObjVLisp style data model which is similar to, but distinct from, the Converge data model of section 4.1.4. TM needs to provide a new data model since the default Converge data model is inherently untyped; whilst figure 4.1 showed the core data model with types, such type

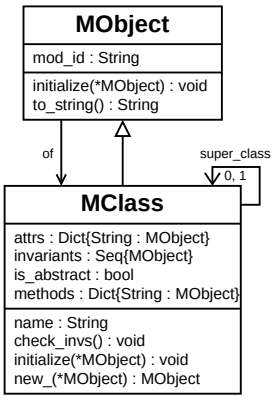


Figure 4.4.: `TM` data model.

information is purely for the benefit of the reader. In contrast, the TM data model is inherently typed, and the type information is used to enforce the correctness of models. The only exception to this is that functions are currently untyped; it would be relatively simple to extend the implementation to record and enforce functions' type information.

Figure 4.4 shows the TM data model. As in Converge, a bootstrapping phase is needed to set up the meta-circular data model. `MObject` and `MClass` are so named to avoid clashing with the built-in classes `Object` and `Class`. Similarly, method and attribute names which might conflict, or be confused with, those found in normal Converge classes are named differently. For example `init` becomes `initialize`, `to_str` becomes `to_string` and `instance_of` becomes `of`. For brevity, and for easy interaction with external code, TM does not directly replicate all built-in Converge types such as strings; built-in Converge types are treated internally as instances of `MObject`.

Once cosmetic differences between the two are ignored, some important differences in the TM and Converge data models become apparent. Most importantly the TM data model has the standard statically typed OO languages notions of separate methods and attributes. TM mclasses are also different in that they can be abstract (i.e. can not be instantiated) and have at most one super class. `MObject` classes possess a `mod_id` slot which is a unique identifier, and which is typed as `String` to allow flexibility over the format of identifiers. The `mod_id` slot is the sole factor in determining whether two model elements are equal or not; because this identifier is immutable, it is used as the objects hash, allowing model elements to be placed within Converge sets.

As all of this might suggest, the TM data model is intended to match the data model found at the core of modelling methodologies e.g. MOF [OMG00]. Since methods and attributes are housed separately within classes, model instances require only a slot per attribute; invoking a method on an object searches the objects `of` class (and its superclasses) for an appropriate method. This is achieved by making use of the Converge MOP (see section 4.1.7). Although the actual implementation is relatively complex, a simplified version demonstrates the salient points. All model objects are instances

of the Converge class `_Raw_Object` which is initialized with a blank slot per attribute of a model class. The Converge MOP is overridden via a custom `get_slot` function in the `_Raw_Object` class. If a slot name matches an attribute slot, that value is returned. Otherwise the model objects `of` class and, if necessary its superclasses, are searched for a method of the appropriate name. Finally, if a method is not found then if the slot name matches that of a normal Converge slot in the `_Raw_Object` instance, the value is returned; otherwise an exception is raised. The following, much simplified, version of the code shows the skeleton of the `_Raw_Object` class and part of its MOP:

```
1  class _Raw_Object:
2    func init(attr_names):
3      self._attr_slots := Dict{}
4      for attr_name := attr_names.iterate():
5        self._attr_slots[attr_name] := null
6
7    func get_slot(name):
8      if self._attr_slots.contains(name):
9        return self._attr_slots[name]
10     else:
11       class_ := self._attr_slots["of"]
12       while 1:
13         if class_.methods.contains(name):
14           return Func_Binding(self, class_.methods[name])
15         if (class_ = class_.super_class) == null:
16           break
17       if exbi Object.has_slot(name):
18         return exbi Object.get_slot(name)
19       else:
20         raise Exceptions.Slot_Exception(Strings.format( \
21           "No such model / Converge slot '%s'", name))
```

A few notes are in order. Firstly the `class_` variable on line 11 is so named since `class` is a reserved keyword in Converge; by convention variable names are suffixed by '`_`' if they would otherwise clash with a reserved word. Note that definitions prefixed by '`_`' are conventionally considered to be private to the module or class they are contained within. On line 14, the `Func_Binding` class creates a binding which, when invoked, will call the Converge function `class_.methods[name]` with its `self` variable bound to `self` (i.e. the `_Raw_Object` instance; see section 4.1.2 for more details about functions and function bindings). The ability to create function bindings in this fashion is an important feature of Converge, allowing a large deal of control over the behaviour of objects.

The TM data model can be considered to be a suitable template for suggesting how more advanced typed modelling languages – perhaps including packages and package inheritance [ACE+02], or allowing classes to inherit from more than one superclass – might be represented in a Converge DSL.

4.5.3. Pre-parsing and grammar

At a high-level, the translation of TM is fairly simple: each model class is converted into an object capable of creating model instances. Before the `TM.model_class` DSL can parse its input, it first it-

erates through the input tokens searching for tokens which have type `ID` and value any of `abstract`, `and`, `at`, `collect`, `extends`, `forAll`, `implies`, `inv`, `Seq`, `Set`. Such tokens are replaced by a keyword token, whose type is the `ID`'s value. Furthermore since the `TM.model_class` DSL is intended to emulate typed languages such as C and Java, it implements a white space insensitive grammar; thus all `INDENT`, `DEDENT`, and `NEWLINE` tokens are removed from the input. The modified token list is then parsed according to the following grammar:

```
top_level      ::= { class }*

class          ::= class_abstract "CLASS" "ID" class_super "{"  { class_field }*
                   "}"
class_abstract ::= "ABSTRACT"
               ::=
class_super    ::= "EXTENDS" "ID"
               ::=

class_field    ::= field_type
               ::= invariant

field_type     ::= "ID" ":" type ";"

type           ::= "ID"
               ::= "SEQ" "(" type ")"
               ::= "SET" "(" type ")"

invariant      ::= "INV" "ID" ":" expr

expr           ::= int
               ::= string
               ::= slot_lookup          %precedence 20
               ::= application          %precedence 15
               ::= binary               %precedence 10
               ::= seq
               ::= set
               ::= "ID"

int            ::= "INT"

string         ::= "STRING"

slot_lookup    ::= expr "." "ID"
               ::= expr "-" ">" forall
               ::= expr "-" ">" at
               ::= expr "-" ">" collect
forall         ::= "FORALL" "(" "ID" "|" expr ")"
               ::= "FORALL" "(" "ID" "ID" "|" expr ")"
at             ::= "AT" "(" expr ")"
collect        ::= "COLLECT" "(" "ID" "ID" "=" expr "|" expr ")"

application    ::= expr "(" expr { "," expr }* ")"
               ::= expr "(" ")"

binary         ::= expr "+" expr        %precedence 30
               ::= expr "-" expr        %precedence 30
               ::= expr ">" expr        %precedence 20
               ::= expr "<" expr        %precedence 20
               ::= expr "==" expr       %precedence 20
               ::= expr "!=" expr       %precedence 20
               ::= expr "IMPLIES" expr  %precedence 10
               ::= expr "AND" expr      %precedence 10

seq            ::= "SEQ" "{" expr ".." expr "}"
               ::= "SEQ" "{" expr { "," expr }* "}"
               ::= "SEQ" "{" "}"
```

```
set             ::= "SET{" expr { "," expr }* "}"
                ::= "SET{" "}"
```

Most of this grammar is straightforward, although it is worth noting a few peculiarities that result from the fact that tokenization is performed by the Converge tokenizer. For example, '`Set{`' is a single token (since `Set{...}` builds up a set in normal Converge). The equivalent notation for sequences is represented by two tokens: '`Seq`' (a new keyword introduced by the DSL) followed by '{'. Fortunately in practise, such idiosyncrasies are largely hidden from, and irrelevant to, the DSL's users.

4.5.4. Traversing the parse tree

For example, the `TM.model_class` function defines a traversal class `Model_Class_Creator` which translates the DSLs parse tree. An idealized version of the beginning of this class looks as follows:

```
import CPK.Traverser

class Model_Class_Creator(Traverser.Traverser):
  func translate():
    return self.preorder()

  func _t_top_level(node):
    // top_level ::= { class }*
    classes := []
    for class_node := node[1 : ].iterate()
      classes.extend(self.preorder(class_node))
    return classes

  func _t_class(node):
    // class ::= class_abstract "CLASS" "ID" class_super "{"  { class_field }*
    //           "}"
    ...
    return [|
      class $<<CEI.name(node[3].value)>>:
        ...
    |]
```

4.5.5. Translating

The actual translation of the parse tree to a ITree involves much repetition, and contains implementation details which are irrelevant to this thesis. The first point to note about the translation is that the resulting ITree largely follows the structure of the parse tree. Having the translation follow the structure of the parse tree is desirable because it significantly lowers the conceptual burden involved in creating and comprehending the translation.

In this subsection I highlight some interesting aspects of the translation; interested readers can use this as a step to exploring the full translation in the `TM` package.

OCL expressions

Translating the OCL subset into Converge is a simple place to start in the translation because it is mostly simple and repetitive. For example, converting binary expressions from OCL into Converge is mostly a direct translation as the elided `_t_binary` traversal function shows:

```
func _t_binary(node):
  // binary ::= expr "+" expr
  //        ::= expr "<" expr
  //        ::= expr "==" expr
  lhs := self.preorder(node[1])
  rhs := self.preorder(node[3])
  if node[2].type == "+":
    return [| $<<lhs>> + $<<rhs>> |]
  elif node[2].type == ">":
    return [| $<<lhs>> > $<<rhs>> |]
  elif node[2].type == "==":
    return [|
      func ocl_equals() {
        lhs := $<<lhs>>
        if lhs.conforms_to(Int) | lhs.conforms_to(String):
          return lhs == $<<rhs>>
        else:
          return lhs is $<<rhs>>
      }()
    |]
```

Note that there is a slight complexity in translating the `==` operator, since OCL defines equality between objects to be based on their value if they are a primitive type, and on their identity if they are a model element. Whilst this is simple to encode as a sequence of expressions, it slightly complicates the `_t_class` traversal function, which is expected to return only a single quasi-quoted expression. In order to work around this limitation the required sequence of instructions are grouped together with a function; the quasi-quotes returns the invocation of this function which is thus a single expression. This idiom occurs frequently in such translations.

Forward references

Forward references between model classes might appear to slightly muddy the structure of the translation. Consider the following example:

```
$<TM.model_class>:
  class Dog {
    owner : Person;
  }

  class Person {
    name : String;
    age : int;
  }
```

With a naïve translation, the result might look similar to the following Converge code:

```
class Dog:
  attributes := Dict{"owner" : Person}
  name := "Dog"

class Person:
```

```
attributes := Dict{"name" : String, "age" : Integer}
name := "Person"
```

Such code would compile correctly, but lead to an exception being raised at run-time since the `Person` class will not have assigned a value to the `Person` variable when it is accessed in the `Dog` class. A standard approach to this problem would be to make the `attributes` field a function; by placing the reference to `Person` a function, the variable access would be deferred until after the `Person` variable contained a value.

The `TM.model_class` DSL takes an alternative approach which is simplistic and, whilst not generally applicable, effective. The `TM` module keeps a record of all model classes encountered. When a new model class is created (i.e. when importing a model containing a `TM.model_class` block), it registers itself with the `TM` module. Rather than directly referring to model classes, type references are strings of the target model class name – when a model class needs to be retrieved, its name is looked up in the `TM` modules' registry, and the appropriate object returned. Thus forward references are a non-issue, since references are only resolved when necessary.

The reason the `TM.model_class` DSL takes this approach is that all model classes live in the same namespace; when we come to transforming model elements it aids brevity that model classes do not need to be prefixed by a package or module name.

Model class translation

The suggestion up to this point has largely been that model classes have been directly translated to normal Converge classes. In fact, model classes are instances of the `MClass` class. Whilst this would suggest that we can use the `metaclass` keyword shown in section 4.1.4, this is not possible since `MClass` requires more information than a normal Converge class. Normal Converge classes take only three parameters `name, supers, fields` whereas a model class requires information about whether it is abstract, its invariants and so on. Thus the class object must be created manually. Using `bound_func` keyword allows an appropriate bound functions to be expressed outside a class.

A much elided, and slightly simplified, version of the `_t_class` traversal function is as follows:

```
is_abstract := bool
class_name := String
super_ := String or null
attrs := Dict{name : type}
invariants := List of tuples [name, function]
operations := List of tuples [name, function]

init_func_var := CEI.ivar(CEI.fresh_name())

return [d|
  bound_func initialize(*args):
    super_attrs := _all_attrs($<<super_>>, 1)
    if args.len() > (super_attrs.len() + $<<CEI.lift(attrs.len())>>):
      raise Exceptions.Parameters_Exception("Too many args")
    super_args_pos := Maths.min(super_attrs.len(), args.len())
```

```
    Func_Binding(self, $<<super_>>.methods["initialize"]).apply( \
      args[ : super_args_pos])
    $<<init_func_body>>


  $<<CEI.ivar(CEI.name(class_name))>> := MClass($<<is_abstract>>, \
    $<<CEI.lift(class_name)>>, $<<super_>>, $<<CEI.lift(attrs)>>, \
    $<<CEI.idict(operations + [[CEI.lift("initialize"), \
    init_func_var]])>>, $<<CEI.ilist(invariants)>>)
|]
```

The `_t_class` traversal function first evaluates and transforms the details of the model class, placing information such as whether the class is abstract into appropriately named variables. Finally it returns quasi-quoted code which contains two things: a function to initialize model class instances, and finally the instantiation of `MClass` itself. The arguments that `MClass` itself requires are hopefully obvious due to the names of the variables passed to it in this example.

Summary of translation

Whilst this section has tersely presented the translation of a `TM.model_class` block, I hope that it shows enough detail to suggest that the bulk of the translation is simple work, with only one or two areas requiring the use of more esoteric Converge features. Appendix D.1 shows the pretty printed ITree resulting from the translation of the example in section 4.5.1.

4.5.6. Diagrammatic visualization

A useful additional feature of TM is its ability to visualize modelling languages and model languages as diagrams. Visualization makes use of the fact that the TM data model is fully reflective, making the traversal and querying of objects trivial. The `Visualizer` module defines several visualization functions, all of which use the GraphViz package [GN00] to create diagrams. For example, the `visualize_modelling_language` function takes a list of model classes, and visualizes them as a standard class diagram. Figure 4.5 shows the automatic visualization of the Simple UML modelling language originally shown in figure 4.3. Note that the modelling language visualization function explicitly shows that all model classes are subclasses of `MObject`.

The Visualizer is also able to visualize arbitrary models as UML object diagrams. Figure 4.6 shows the resulting visualization of the model in the following code:

```
person := Simple_UML.Class("Person")
dog := Simple_UML.Class("Dog")
dog.attrs.append(Simple_UML.Attribute("owner", person, 0))

Visualizer.visualize_model([Person, Dog])
```

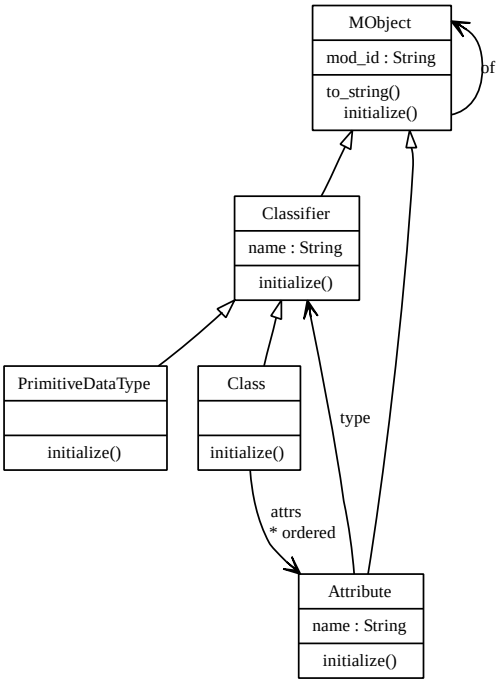


Figure 4.5.: ‘Simple UML’ modelling language visualized.

4.6. Future work

Because the core of Converge is a mix of established languages, the core language is largely stable syntactically and semantically. The implementation of the compile-time meta-programming facilities is currently less than satisfactory in one or two areas, but is eminently usable. One feature in particular that appears to confuse new users to the language relates to the very different effects of the splice annotation. The syntax, inherited from Template Haskell, means that $ behaves very differently when inside (a simple replacement of the splice annotation) and outside (cause compile-time evaluation) quasi-quotes. A simple change of syntax may suffice to solve this problem, or it may be considered

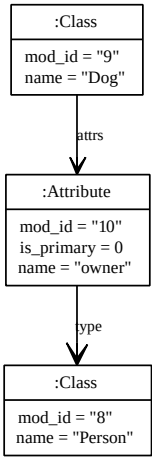


Figure 4.6.: A ‘Simple UML’ model language visualized.

to be an inevitable part of the learning curve that compile-time meta-programming presents.

Although Converge's error reporting facilities are at least as good as any comparable language, there is still room for considerable improvement from the users point of view. Although section 4.2.13 used nested quasi-quotes to customise error reports, it may be necessary to find a lighter weight technique if DSL authors are to be encouraged to provide high quality error reporting.

The syntax extension feature presents the greatest opportunity for future work. The most obvious improvement would be to allow the user to provide their own tokenization facility. This may result in simply passing a single string to the DSL implementation function, or it may involve a more sophisticated interaction between the Converge tokenizer and parser and the DSL tokenizer and parser. For example parsing algorithms such as Pack Rat parsing [For02] allow the conflation of tokenizing and parsing in a way that might lend themselves to syntax extension.

Section 5.6 shows how – currently slightly inelegantly – DSLs can incorporate normal Converge code within them. Improving the mechanism for weaving the core language and DSL extensions in arbitrary ways will be important if this feature is to be exploited to its full potential.

4.7. Summary

In this chapter, I first presented the core of the Converge programming language. The core language is a largely standard dynamically typed OO language. I then detailed Converge's compile-time meta-programming facility, which allows users to extend the language in an arbitrary fashion; I also explained how similar functionality may be added to other dynamically typed OO languages. I then used Converge's compile-time meta-programming facility to add a syntax extension facility to Converge. The syntax extension feature allows users who extend the language to present their extensions to other users in a syntactically seamless fashion. Finally I demonstrated the syntax extension feature by showing how a powerful typed modelling language can be embedded within Converge.

Chapter 5.

A rule based model transformation system

This chapter presents a unidirectional stateless model transformation language MT, implemented as a DSL within Converge. MT serves as an example of a non-trivial example of a DSL implemented as Converge, exercising many parts of Converge and showing many of the idioms common when developing DSLs in Converge.

MT shares several aspects in common with model transformation languages such as the QVT-Partners approach [QVT03b]. That is, it is a rule-based system, utilising patterns. However there are a number of advances over, and significant differences from, previous approaches. Some of these are a side-effect of implementing MT as a DSL within Converge; some are the result of experimentation with a concrete, but malleable, implementation. For example, MT allows normal Converge imperative expressions to be embedded within it. MT's implementation is interesting for other reasons, particularly its exploitation of a number Converge's other unique features, such as goal-directed evaluation.

This chapter comes in three main parts. Firstly the chapters running example is introduced, followed by an introduction to the QVT-Partners model transformation approach. The QVT-Partners approach is then used as the basis for the MT language, which is introduced partly through example. Finally the implementation of the MT language as a DSL in Converge is discussed in more detail. In the wider context of this thesis, MT is a necessary step towards the change propagating transformation language presented in chapter 6.

5.1. Running example

This chapter makes use of a simple running example of a transformation from a UML like modelling language to a model of relational databases. The chief reason for using this example is the ability to compare the result with its implementation in other model transformation approaches (e.g. [DIC03, OQV03, QVT03a]. The example also has the virtue that it can be easily considered in 'simple' and

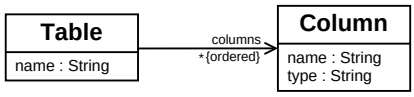


Figure 5.1.: Simplified relational database meta-model.

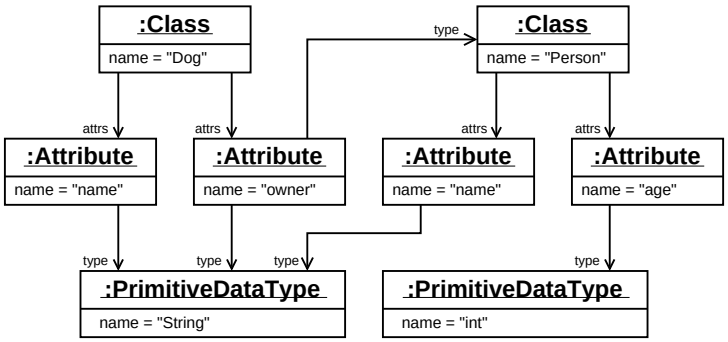


Figure 5.2.: An example of a source simple UML model.

'advanced' variants. Whilst the advanced variant does exercise a reasonable number of features of any given model transformation approach, it is still of an appropriate size for this thesis.

The original example is defined in [QVT03a]. The simple variant is as follows. The meta-model for a simple UML modelling language was previously given in figure 4.3. A corresponding meta-model for simplified relational databases is given in figure 5.1. In essence, the transformation takes in a `Class` and transforms it to a `Table` of the same name. `Attribute`'s whose type is a `PrimitiveDataType` (e.g. string, integer) are transformed to a column of the same name and the same primitive data type. `Attribute`'s whose type is a `Class` (i.e. the type refers to another user class) are transformed to a number of columns: the transformation recursively 'drills down' through a class's non-primitively typed attributes until it reaches attributes with primitive datatypes. At each point in the recursion the name of the current class along with a '_' character is appended to the column name. The net result of this is that non-primitive data types are flattened.

Consider the source model of figure 5.2 (trivially adapted from section 4.5.5). Assuming that the `Dog` class is used as the input class, an implementation of this transformation should produce the relational database model as in figure 5.3.

This is the core of the example that will be used throughout this chapter. As the chapter progresses,

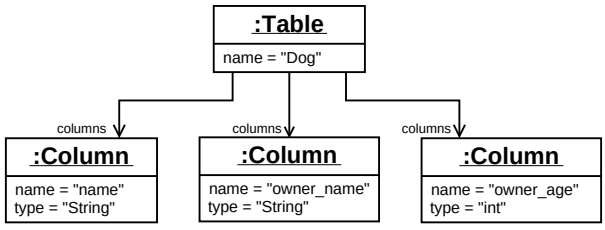


Figure 5.3.: An example of a target relational database model

I will progressively add complexity to the example.

5.2. The QVT-Partners model transformations approach

The QVT-Partners approach was outlined in section 3.3.10. In this section, I explain some relevant aspects of the approach in more detail, since the MT language shares several factors in common with the QVT-Partners approach. Whilst the QVT-Partners approach has the concept of 'specification' and 'implementation' transformations (known as *relations* and *mappings* respectively[1]), for the purposes of this chapter, transformation specifications are largely irrelevant and are consequently ignored. The QVT-Partners approach also defines a diagrammatic syntax for transformations which is similarly ignored.

5.2.1. Overview

A transformation in the QVT-Partners approach consists of a number of mappings. A mapping consists of one or more source *domains* (analogous to a function parameter) and a target imperative body. Each domain consists of one or more *patterns* to match against. Patterns are written in a language designed to make expressing constraints over models succinct; they are analogous to textual regular expressions as found in e.g. Perl [WCO00]. Imperative bodies consist of a single expression in an extended OCL variant that is capable of side effects. Using the meta-models presented in the previous section, a simple mapping for transforming a class to a table would look as follows:

```
mapping Class_To_Table {
  domain {
    (Class)[name = n, attrs = A]
  }

  body {
    let
      columns = A->collect(attr columns = Set{} |
        columns + Attr_To_Column(attr))
    in
      (Table)[name = n, columns = columns]
    end
  }
}
```

The intuitive meaning of this is hopefully fairly straightforward. The `Class_To_Table` rule will match against a `Class` model element, with the pattern binding whatever name the class has to the variable `n` and whatever attributes it has to the variable `A`. The imperative body then creates a corresponding `Table` whose name matches the source class. It should be noted that although the `Table` expression in the body appears to be a pattern, this is something of a syntactic illusion: the pattern-esque syntax is simply syntactic sugar for object creation and slot updating. `Attr_To_Column`

[1] The author of this thesis takes full responsibility for his decision to use these names — given their multiple overloaded meanings in the wider field, in retrospect they were not the best possible choices.

refers to another mapping, which is used to transform each attribute in the source class into one or more database columns which are then placed within the target table.

5.2.2. Pattern language

In the context of this thesis, the most important novel aspect of the QVT-Partners approach is its pattern language. Its aim is to provide a concise textual notation for expressing constraints over models, thereby reducing the time needed to write and to comprehend a transformation. In this subsection, I provide a brief background of patterns before informally explaining the QVT-Partners pattern language.

Many computer users are familiar with textual pattern languages e.g. via operating system commands such as `ls *.txt`. One can obtain a crude gauge of the popularity of textual regular expressions by the fact that suitable libraries are found as standard in many modern programming languages such as Perl, Python and Ruby. However whilst pattern languages are commonly thought of as being suitable only for matching against text, they can be used to match against other, much richer, datatypes. For example program transformation patterns match against complex AST's [Big98]. Intuitively, designing a pattern language involves a compromise between providing a concise notation for capturing common constraints, and providing a completely general mechanism — the more cases a pattern language can express, the less concise it is likely to be. Many pattern languages are thus tailored for the common case as opposed to the general case. Textual regular expressions, for example, are typically defined as finite-state automata which can not express a seemingly simple constraint such as ensuring that a string contains balanced open and close brackets[2]. It is therefore desirable that when a pattern languages' expressive limit is reached, a suitable escape mechanism into a more powerful, if verbose, system is available.

The QVT-Partners approach provides a specific pattern language for expressing constraints over models. The QVT-Partners approach is important in this respect because, as noted in chapter 3, most current model transformation approaches do not provide pattern languages. Although it could be argued that graph transformation approaches utilise pattern languages, their matching facilities are relatively simplistic, particularly when compared to textual regular expressions; many approaches allow nodes to only match fixed numbers of nodes and edges. For the purposes of this thesis, I therefore do not consider graph transformation approaches to have pattern languages.

The QVT-Partners approach provides a small pattern language for expressing constraints over models. A slightly simplified version of the grammar for the pattern language is as follows:

[2]Note that some implementations of regular expressions are no longer 'regular' in the formal sense of that word. For example, modern Perl contains an experimental feature which can express the balanced brackets constraint.

⟨*pattern*⟩ ::= '_'
| ⟨*set_pattern*⟩
| ⟨*seq_pattern*⟩
| ⟨*obj_pattern*⟩
| ⟨*expression*⟩

⟨*set_pattern*⟩ ::= '{' [⟨*pattern*⟩* ('|' ⟨*pattern*⟩)*] '}'

⟨*set_pattern*⟩ ::= '[' [⟨*pattern*⟩* ('|' ⟨*pattern*⟩)*] ']'

⟨*obj_pattern*⟩ ::= '(' ⟨*var*⟩ [',' ⟨*var*⟩] ')' '[' [⟨*field_pattern*⟩ (',' ⟨*field_pattern*⟩)*] ']'

⟨*field_pattern*⟩ ::= ⟨*var*⟩ '=' ⟨*pattern*⟩

⟨*var*⟩ ::= 'ID'
| '_'

The reference to the rule *expression* is a reference to a rule which contains the extended OCL variants' grammar.

The QVT-Partners approach identifies three main types of patterns: set, sequence, and object patterns. Although not explicitly noted as such, variables in patterns are essentially patterns themselves. To ensure consistency with the rest of this chapter, I refer to object patterns as *model element patterns*. All types of pattern share in common one thing: given a particular model element, they will either succeed or fail to match against it.

Set and sequence patterns are similar to those used in function parameters in functional and logic programming languages such as Haskell and Prolog. For example a set pattern `Set{1, 6 | R}` will match successfully against a set that contains at least two items `1` and `6`; a new set containing all of the original sets items other than `1` and `6` will be bound to `R`. Intuitively, variable names mean 'match anything and bind'; henceforth these will be referred to as *variable bindings*. If the same variable name appears more than once in the same scope, all instances of that variable name must match against equivalent objects (the definition of object equality in the QVT-Partners approach is inherited from the MOF [OMG00]). The special variable '`_`' matches against anything and immediately discards the result; multiple instances of '`_`' do not need to match against equivalent objects.

Although relatively simple, model element patterns are the backbone of the pattern language. Model element patterns specify the type that matching model elements must conform to, and an optional 'self' variable which will be bound to the element matched against. The model element then specifies a number of slots and a pattern against which each slot in the model element must match against. The terse power of model element patterns is best demonstrated by example. Consider first

the following model element pattern:

```
(Dog, d)[name = n, owner = (Person)[name = "Fred"]]
```

This pattern will match successfully against a model element which is of type `Dog` and whose owner is Fred. After the match the variable `d` will point to the particular `Dog` element matched, and `n` will contain the dog's name. This pattern is approximately equivalent to the following Converge-esque pseudo-code function which returns a dictionary of bindings if the source element is matched successfully, failing otherwise:

```
func match(element):
  if not element.conforms_to(Dog):
    return fail
  d := element
  n := element.name
  if not element.owner.conforms_to(Person):
    return fail
  if element.owner.name != "Fred":
    return fail

  return Dict{"d" : d, "n" : n}
```

Although it may seem more logical to have used OCL to express this, it should be noted that expressing the creation and update of bindings in OCL would require complex encodings. Partly due to this difficulty, the QVT-Partners approach defines a new calculus in order to have a suitable semantic domain. The calculus is given an operational semantics, and directly implements several pattern matching primitives; it can be seen to be similar to the imperative object calculus of Abadi and Cardelli [AC96] extended with pattern matching. Using Converge pseudo-code as the target translation of the example pattern avoids the need to define and explain the calculus.

As this example translation clearly shows, the model element pattern is not only considerably terser than its equivalent pseudo-code, but is arguably easier to comprehend. Since the pseudo-code has to explicitly embed certain aspects of the model transformation (e.g. the `return fail` statements) the important aspects of the pattern are obscured. This is simply a recasting of the problem of expressing model transformations in GPLs. Even though the pattern language is simple, it neatly solves many such problems.

5.2.3. Complete example

The running example expressed in the QVT-Partners approach is as follows:

```
mapping Class_To_Table {
  domain {
    (Class)[name = n, attrs = A]
  }

  body {
    let
      columns = A->collect(attr columns = Set{} |
        columns + or(Primitive_Type_Attr_To_Column("", attr),
          User_Type_Attr_To_Column("", attr))
    in
```

```
      (Table)[name = n, columns = columns]
    end
  }
}

mapping User_Type_Attr_To_Column {
  domain {
    (String, prefix)[]
  }

  domain {
    (Attribute)[name = n, type = (Class)[name = ct, attributes = A]]
  }

  body {
    let
      new_prefix =
        if prefix == "" then
          n
        else
          prefix + "_" + n
        end
    in
      for A->collect(attr attrs = Set{} |
        attrs + or(Primitive_Type_Attr_To_Column(new_prefix, attr),
          User_Type_Attr_To_Column(new_prefix, attr)))
    end
  }
}

mapping Primitive_Type_Attr_To_Column {
  domain {
    (String, prefix)[]
  }

  domain {
    (Attribute)[name = n, type = (PrimitiveDataType)[name = pt]]
  }

  body {
    Set{(Column)[name = if prefix == ""
        name = n
      else
        name = prefix + "_" + n
      end,
      type = pt]}
  }
}
```

One feature in particular that requires explanation is the `or` function used in the `Class_To_Table` and `User_Type_Attr_To_Column` mappings. `or` is not a normal function call, but is a built-in *combinator* which lazily executes the mappings passed as arguments to it in order until one succeeds and produces a value. Note that unlike most rule-based systems, the QVT-Partners does not provide a function which takes an element and attempts to find a rule which will transform it. Although the `or` combinator can provide the same functionality, its repeated use becomes tiresome to the programmer due to the continuous hard-coding of mapping names required.

The overall structure of this transformation is fairly simple. The `Class_To_Table` is the top-level mapping which takes in a class and iterates through its attributes, invoking other mappings to produce columns. Attributes are transformed in one of two ways. Both the `User_Type_Attr_To-`

`_Column` and `Primitive_Type_Attr_To_Column` mappings take two arguments: a string and an attribute. The string represents the current column name prefix being built up as the transformation drills into user data types. Attributes which have a primitive data type are transformed by the `Primitive_Type_Attr_To_Column` mapping into a single column. Attributes which have a user data type are transformed by the `User_Type_Attr_To_Column` into one or more columns; the `User_Type_Attr_To_Column` mapping is the recursive mapping which drills into user data types.

Although the example from section 5.1 has been successfully expressed in the QVT-Partners approach, the result is perhaps more verbose than one may have expected. Indeed, somewhat surprisingly, a simple GPL equivalent of this example is smaller. One might thus reasonably expect that expressing such a transformation in the QVT-Partners approach has other benefits. Since mappings only allow the expression of unidirectional stateless transformations, the only potential gain over a GPL approach is the possibility of automatically created tracing information. Unfortunately the QVT-Partners approach does not explain how rules can create such tracing information in practise. Since the GPL equivalent is likely to be more readily understood by a far wider range of people, the overall benefits of this approach are not clear cut. In the following subsection I outline three issues which are indicative of the problems of the QVT-Partners approach.

5.2.4. Issues with the approach

As the verbose example in section 5.2.3 may suggest, the QVT-Partners approach has a number of minor flaws and limitations which hamper practical use. In this subsection I outline, in approximately descending order, three areas which are indicative of where the QVT-Partners falls short of its intended goals. These points are instructive in understanding several of the design decisions made in the MT language of section 5.3.

Inappropriate imperative language

The imperative bodies of mappings are written in a so-called ‘extended OCL’, which is intended to allow users familiar with OCL the chance to reuse that knowledge in an imperative setting. This has an immediate negative effect: extending OCL with imperative constructs means that the often desirable properties OCL had as a purely side-effect free language are lost[3]. Conversely when it comes to acting as a normal GPL, the resulting language is decidedly unwieldy since it lacks appropriate constructs for common operations. For example, there is no explicit sequencing mechanism: the imperative

[3]OCL 2.0 is not in fact entirely side-effect free; however the situations in which this property is violated are largely irrelevant in the context of this thesis.

body consists of exactly one OCL constraint, and sequencing can be achieved clumsily via the `let` expression.

Underpowered patterns

The pattern language defined in the QVT-Partners approach is novel in the context of model transformations, and potentially very useful. However as the relatively simple definition in section 5.2.2 may suggest the pattern language is lacking in significant expressive power.

The pattern language itself is limited in two main ways:

1. Within model element patterns it is only possible to check for the equality of slots. For example, it is not possible to use a model element pattern to express that a match against an object should succeed provided a given slot does not match a particular value.

   In order to sidestep this problem, users must add additional OCL in a `when` clause.

2. Model element patterns can only match against a fixed number of elements. A model element pattern, for example, can only match successfully against one, and only one, model element. Note that whilst set and sequence patterns can match against sets and sequences of arbitrary lengths, only a fixed number of elements can be explicitly identified within any given set or sequence.

   There is no general solution to this problem. Typically a new mapping needs to be added so that iteration in a `when` clause can control the number of times another mapping is successfully matched.

Scoping rules

Since a bare variable name in a slot constitutes a variable binding, the QVT-Partners approach has fragile scoping rules, since it is difficult to distinguish a variable binding from a variable reference. Consider the simple example of section 5.2.2 `(Dog, d)[...]`. `Dog` is a reference to the `Dog` model class, whereas `d` is a variable binding which will be set to the self value of the object which matches the model element pattern. This means that it is impossible to express that a model element pattern should match against a particular element. For example, in a meta-circular system where a class `M` is an instance of the `Singleton` meta-class, the model element pattern `(Singleton, M)[...]` will create a local variable `M` rather than ensuring that it matches against the element pointed to by `M`.

The QVT-Partners approach allows a `when` clause to be scoped over all domains establishing a constraint across domains since it does not create any new variable bindings. However it is not possi-

ble to introduce a similar clause (often called `where` in similar approaches) scoped over all domains which introduces new variable binding without introducing ambiguities. Consider the following example: should the `x` in the pattern bind the value of `slot` to the variable, or should it ensure that `slot` contains the value introduced in the `where` clause?

```
mapping X {
  domain {
    (E)[slot = x]
  }

  where {
    x := 5
  }
}
```

In the QVT-Partners approach, scoping ambiguities are avoided by disallowing several potentially useful features that may introduce new variable bindings. However the end result is that whilst expressions such as in `(Dog, d)[...]` are statically resolvable, they are confusing for users. The overall effect of the scoping rules are to severely limit the possibilities for extending or embedding the language.

5.2.5. Summary

The QVT-Partners approach provides a number of innovations compared to other model transformation approaches, most notably the use of patterns. However in practise the simplistic nature of the approach means that it falls somewhat short in its aim to allow users to express model transformations more easily than in GPLs.

In the next section of this chapter, I define a new model transformation language MT which takes elements of the QVT-Partners approach, adding extra features and addressing some of the approaches flaws.

5.3. The MT Language

The MT language is a new unidirectional stateless model transformation language, implemented as a DSL within Converge. MT transforms instances of the typed modelling language TM (section 4.5) into new instances of TM. In essence, MT defines a natural embedding of model transformations within Converge, using declarative patterns to match against model elements in a terse but powerful way, whilst allowing normal imperative Converge code to be embedded within rules.

Because MT is implemented as a DSL within Converge, it has existed as a concrete implementation from shortly after its original design was sketched out. This has proved to be significant, since practical experience with the approach has been rapidly fed back into the implementation. Rapid

development has been facilitated by the flexible environment provided by Converge. The ability to experiment with the implementation has ultimately led MT to contain a number of insights and distinct differences from other approaches. Such insights range from a more sophisticated pattern language to suitable ways to visualize model transformations. In the wider context of this thesis, MT is also important as the basis of the change propagating language presented in chapter 6.

In this section I first highlight the main features of MT, then present an MT version of the running example, before showing how MT transformations are run in practise. This section is intended to present the basic features of MT, before more advanced features are described in section 5.5.

5.3.1. Basic details

An MT transformation has a name and consists of one or more rules, the ordering of which is significant. Rules are effectively functions which define a fixed number of parameters and which either succeed or fail depending on whether the rule matches against given arguments. Rules and functions in MT are essentially synonymous as in approaches such as TXL (see section 3.3.6). If a rule matches successfully, one or more target elements are produced and it is said to have executed; if it fails to match successfully, nothing is produced. Rules are comprised of: a source matching clause containing one or more source patterns; an optional when clause on the source matching clause; a target producing clause consisting of one or more expressions; and an optional where clause for the target production clause.

An MT transformation takes in one or more source elements, which are referred to as the *root set* of source elements. The transformation then attempts to transform each element in the root set of source elements using one of the transformations rules, which are tried in the order they are defined. If no rule matches a given element, an exception is raised and the transformation is aborted.

The general form of an MT transformation is as follows:

```
import MT.MT

$<MT.mt>:
  transformation transformation name

  rule rule name:
    srcp:
      pattern_1
      ...
      pattern_n

    src_when:
      expr

    tgtp:
      expr_1
      ...
      expr_n

    tgt_where:
```

expr$_1$
...
expr$_n$

The `srcp` and `srcp_when` clauses are collectively said to form the `source model clauses`; similarly the `tgtp` and `tgtp_when` clauses are collectively said to form the `target model clauses`.

Transformations are translated by MT into a Converge class with the name of the transformation; rules are translated to functions of the same name within the class. In order to run a transformation, the transformation class is instantiated; each class can be instantiated multiple times. Transformation classes have additional functions for e.g. extracting tracing information (see section 5.3.6).

Transformation rules contain normal Converge code in expressions; such expressions can reference variables outside of the model transformation DSL fragment. This is an important aspect of MT since it allows users to use normal Converge functions arbitrarily, and without penalty. In other words, when the model transformation language itself is inadequate in a particular respect, a normal reusable Converge function can be defined outside of the model transformation, but which can be called from within any model transformation.

MT transformations hold a record of tracing information, which is automatically created as transformation rules are executed. Each rule executed adds a new trace. Each trace is a tuple, encoded as a Converge list, of the form `[[`*`source elements`*`], [`*`target elements`*`]]`. The source elements that are stored in the tracing information do not necessarily constitute the entire universe of elements passed via parameters to the transformation. By default, only elements matched by non-nested model element patterns are recorded in the tracing information. Section 5.4.2 details the default tracing creation mechanism, and explains how it can be augmented or overridden.

A simple example of a transformation and a rule is as follows:

```
$<MT.mt>:
  transformation Classes_To_Tables

  rule Class_To_Table:
    srcp:
      (Class, <c>)[name == <n>, attrs == <A>]

    tgtp:
      (Table)[name := n, cols := columns]

    tgt_where:
      columns := []
      for attr := A.iterate():
        columns.extend(self.transform([""], [attr]).flatten())
```

This rule is the MT analogue of the rule in section 5.2.1. Note how normal Converge code is interspersed amongst the MT DSL (see section 5.6.12 for implementation information). Since transformations are translated to Converge classes, to access functions 'internal' to the transformation, one must use the `self.` prefix. The `transform` function, for example, takes source elements and tries each

rule in the transformation in succession until it finds one which successfully matches the elements and produces values. If the `transform` function does not find a suitable rule then, by default, an exception is raised. The `transform` function is also used internally by the transformation as the mechanism used to transform the root set of source model elements. Section 5.5.3 shows an example of a rule that can guarantee that the `transform` rule can be made to succeed on any given input.

In the following subsections I explain in more detail how rules match and produce elements, including a detailed examination of the pattern language and pattern multiplicities.

5.3.2. Matching source elements with patterns

Each pattern in the `srcp` clause of a rule corresponds to a domain in the QVT-Partners approach. Arguments must be passed as lists rather than sets; whilst TM elements can be placed into Converge sets (section 4.5.2), users may wish to transform non-hashable elements such as lists. Each list contains the top-level model elements which each pattern can match against. Elements can exist, directly or indirectly – that is, as top-level elements, or by being reachable via the graph that constitutes a model – in one or more arguments. In order to avoid the problems noted in section 5.2.4, variable bindings are surrounded by angled brackets '`<`' and '`>`' to distinguish them from normal Converge variable references.

The matching algorithm used by MT is intentionally simple. Each pattern in turn attempts to match against the top-level source elements passed in the appropriate argument. Each time a pattern matches it produces variable bindings which are available to all subsequent patterns. If a pattern fails to match, control backtracks to previous patterns (in the order of most recently called), which attempt to generate another match given the variable bindings and arguments available to them. The generation of an alternative match causes new variable bindings to produce, which allows the rule to attempt another match of later patterns. The `src_when` clause, if it exists, is tried once all patterns have been matched successfully; it is essentially a guard over patterns. If it fails, patterns are requested to generate new matches exactly as in the failure of a pattern to match. The implementation details of such behaviour are largely hidden from the user by the use of patterns.

The order that patterns are defined in the `srcp` clause is significant, for two separate reasons. Most obviously it is necessary to ensure that users' sequence variable bindings and references to the bound variables correctly. However there is a second reason that, whilst less obvious, is critical to the performance of larger transformations. Making the order of patterns significant allows users to make use of their domain knowledge to order them in an efficient way. Consider a rule which has two independent patterns *x* and *y* where *x* tends to match against many source elements, but *y* against few. Placing *x* first in the `srcp` clause means that when *y* fails *x* will try to produce more values; if

x can produce multiple matches, *y* may be executed many times unnecessarily. If *y* is placed first in the `srcp` clause then if it fails to match against its input the rule fails without ever trying to match *x*. Sensible ordering of patterns in this way can lead to a significant boost in performance as unnecessary matches are not evaluated.

Each pattern is translated to a Converge generator, which provides a natural mechanism for lazily generating all possible matches. Translated patterns are conjoined to make use of Converge's backtracking abilities. Note that the `src_when` clause, if it exists, must be a single Converge expression which either succeeds or fails given variable bindings generated by patterns in the `srcp` clause.

5.3.3. Pattern language

MT's pattern language is a super-set of that found in the QVT-Partners approach. MT defines a number of *pattern expressions*: model element patterns, set patterns, variable bindings, and normal Converge expressions. Patterns written in the latter language can be directly translated into MT with only minor syntactic changes.

There are two significant differences between the two pattern languages. Firstly – as noted in the section 5.3.2 – variable bindings in MT must be surrounded by angled brackets to ensure harmony between MT and Converge's scoping rules. Secondly, model element patterns in MT can contain comparisons other than equality between slots; henceforth these are known as *slot comparisons*. Any of the standard Converge comparison operators can be used in slot comparisons (see section 4.1.5 for a list of comparison operators). A model element pattern in MT is said to consist of zero or more slot comparisons.

As a trivial example of slot comparisons, one can take the model element pattern example from section 5.2.2 (making the necessary minor syntactic modifications), and change it to find dogs whose owner is not Fred:

```
(Dog, <d>)[name == <n>, owner == (Person)[name != "Fred"]]
```

This same example would necessitate an OCL constraint in a `when` clause in the QVT-Partners approach.

Allowing different types of slot comparison in model element patterns opens up new possibilities. Since MT allows the same slot name to appear in more than one slot comparison, one can test a slot for multiple conditions as in the following model element pattern:

```
(Person)[age >= 18, age <= 25]
```

There is one other additional feature in the MT pattern language. Since model elements are Converge objects, slot comparison is not entirely synonymous with attribute comparison, since slots may contain functions. MT's model element patterns therefore provides support for functions as shown in

the following example:

```
(Person)[calc_wage() > 18000]
```

Functions in slot comparisons can be passed an arbitrary number of arguments passed to them; all arguments are normal Converge expressions.

Pattern multiplicities are not considered to be a part of the core pattern language, but are a significant enhancement in MT over the QVT-Partners approach; they are detailed in section 5.5.2.

5.3.4. Producing target elements

When an MT rule executes it produces one or more target elements. An exception is raised if a rule executes but fails to produce any elements. The number of elements produced is determined by the number of expressions in the `tgtp` clause. If the `tgtp` clause has a single expression, then the rule produces a single element; if it contains more than one expression, then the rule produces a list whose length is the same as the number of expressions in the `tgtp` clause. Each expression is a normal Converge expression, but with an important addition. The MT DSL admits *model element expressions* by extending the Converge grammar rule `expr` (see section 5.6.13 for implementation details). Model element expressions differ from model element patterns both conceptually and syntactically. Conceptually a model element expression is an imperative, creational action. There is therefore no concept of a 'self' variable in a model expression. Furthermore, to reinforce the notion that model expressions are imperative actions, slot assignments use the normal Converge assignment operator `:=`.

Expressions in `tgtp` have an optional `for` suffix which allows a single expression to generate multiple values. If one ignores the obvious syntactic difference of the relative location of the keyword, the `for` suffix works largely as its normal Converge counterpart, taking a single expression and continuously pumping it for values until it fails. Variables defined in the `for` suffix are scoped only over the single expression in the `tgtp` clause that it suffixes. Assuming `COLS` is a list, a typical usage of this feature is as follows:

```
(Column)[name := col.name] for col := COLS.iterate()
```

Note that if the above example was the only expression in a `tgtp`, the result of the rule would be a list of length `COLS.len()`. However, if the expression was the first of two in a `tgtp`, the rule would produce a list of length two, with the first element being a list of length `COLS.len()`. Section 5.8 suggests a possible extension to MT which would allow a rule to produce a number of elements not solely determined by the number of expressions in its `tgtp`.

The `tgt_where` clause, if it exists, is a sequence of Converge expressions which are executed before the `tgtp` clause. Variables in the `tgt_where` clause are automatically scoped over the

`tgtp` clause. Unlike the `src_when` clause, there is no notion of success or failure with the `tgt_where` clause, which is simply a helper function for the `tgtp` clause. Note that expressions in the `tgt_where` clause can contain model element expressions.

5.3.5. Example

The following is a complete Converge module which implements the running example:

```
1  import Sys
2  import Relational, Simple_UML
3  import MT.MT
4
5  func concat_name(prefix, name):
6    if prefix == "":
7      return name
8    else:
9      return prefix + "_" + name
10
11 $<MT.mt>:
12   transformation Classes_To_Tables
13
14   rule Class_To_Table:
15     srcp:
16       (Class, <c>)[name == <n>, attrs == <A>]
17
18     tgtp:
19       (Table)[name := n, cols := columns]
20
21     tgt_where:
22       columns := []
23       for attr := A.iterate():
24         columns.extend(self.transform([""], [attr]).flatten())
25
26   rule User_Type_Attr_To_Column:
27     srcp:
28       (String, <prefix>)[]
29       (Attribute)[name == <n>, type == (Class)[name == <cn>, attrs == <CA>]]
30
31     tgtp:
32       self.transform([concat_name(prefix, n)], [ca]) for ca := CA.iterate()
33
34   rule Primitive_Type_Attr_To_Column:
35     srcp:
36       (String, <prefix>)[]
37       (Attribute)[name == <n>, type == (PrimitiveDataType)[name == <pn>]]
38
39     tgtp:
40       [(Column)[name := concat_name(prefix, n), type := pn]]
```

The overall structure of this transformation is deliberately similar to the version in the QVT-Partners approach of section 5.2.3. One important difference is that the repetitive code which builds up the column prefix is factored out into a normal top-level function `concat_name`.

A slight difference between the MT transformation and the QVT-Partners approach equivalent is that the `User_Type_Attr_To_Column` in the former rule produces a nested list, the innermost list containing a list of columns. The outer list will be of length `CA.len()`, with each entry in the list being of arbitrary length. Consequently the `flatten` function call in line 24 is necessary to remove the nesting that will be present if the `User_Type_Attr_To_Column` rule is called.

5.3.6. Running a transformation

Details of how to run a model transformation, including details such as the format of its inputs and outputs and so on, are surprisingly absent from descriptions of the majority of model transformation approaches. Since this has implications for the design of the model transformation languages presented in this thesis, it is important to be explicit about how transformations are run. In this subsection I detail the process of running a MT transformation.

Running a transformation in MT involves instantiating a transformation class and passing it model elements. The transformation then executes, attempting to find a rule to transform each element in the root set of source elements. If the transformation is successful in transforming the root set of elements, a transformation object will be returned. The transformation object can then be queried to find the target model elements produced and the corresponding tracing information. The format of MT's inputs and outputs is simple. Source elements must be instances of elements defined in a `TM.model_class` block (see section 4.5.2, and note that built-in Converge types such as strings and ints are defined to be valid TM model elements). Similarly target elements will be TM model elements.

The following example creates a simple input model and then executes the `Classes_To_Tables` transformation:

```
dog := Simple_UML.Class("Dog")
person := Simple_UML.Class("Person")

dog.attrs.append(Simple_UML.Attribute("name", Simple_UML.String))
dog.attrs.append(Simple_UML.Attribute("owner", person))

person.attrs.append(Simple_UML.Attribute("name", Simple_UML.String))
person.attrs.append(Simple_UML.Attribute("age", Simple_UML.Integer))

transformation := Classes_To_Tables(dog)
```

The target elements produced by the transformation can be accessed via the `get_target` function. Since both the source and target elements are TM model elements, one can apply the standard TM visualization to our example. The source model is shown in figure 5.4, with the target model in figure 5.5. Note that the colours given to the source and target models will be used in the remainder of this thesis: source elements are shown in blue, target elements in green. To emphasise that all such visualization's are the result of a real, running system, figure 5.6 shows an MT transformation executing on an OpenBSD machine.

If an element passed to the `transform` function can not be transformed by any of the available rules, an exception is raised showing the offending element(s) and the transformation is aborted. Users may catch such an exception if desired, however one may reasonably ask why the transformation does not attempt to recover gracefully in such instances. Unfortunately this seems to be unrealistic in the

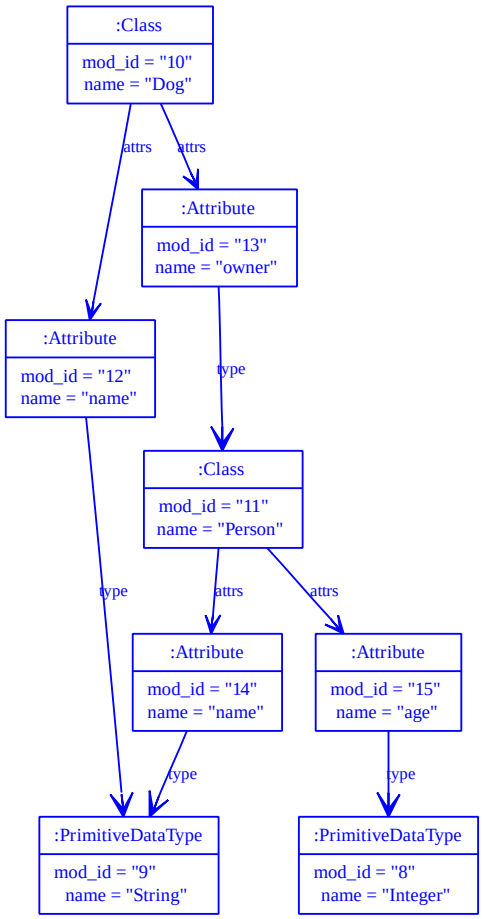


Figure 5.4.: Source model.

general case for the following reason. Since the `transform` function is called with the expectation that it will return a result, when it fails to find a suitable rule to transform a given element it is unable to fulfil the callers expectation that an element will be returned. In order to maintain this expectation, `transform` could conceivably return a 'dummy' target element as a placeholder. However such a dummy element would be unlikely to satisfy the constraints on the target meta-model, and would thus generally cause an exception to be raised. In the, probably small, number of situations where the dummy element did not cause an error, it is then less than clear that the resulting target model will be of significant use to the user. Section 5.5.3 shows an example of a 'default' rule which guarantees that the `transform` function can not fail.

5.4. Tracing information

In this section I describe how MT deals with tracing information. First I show how tracing information is visualized, then describe the standard mechanism for creating it, before showing how the user can augment or override the default tracing information created.

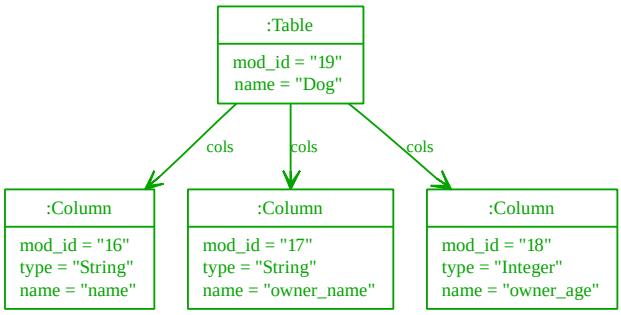


Figure 5.5.: Target model.

5.4.1. Visualizing tracing information

Section 5.3.6 showed how a MT transformation can be run, and used the default visualization capabilities of TM to visualize the source and target models of a transformation. However, MT transformation instances also store tracing information (see section 5.3.1) relating source and target elements. Visualizing tracing information is an interesting challenge, and one that has hitherto received scant attention in the context of model transformations. Work on trace visualization in areas such as object orientated systems (e.g. [BH90]) is of little use in the context of model transformations due to the different nature of what is being visualized. Egyed motivates the use of tracing information in the context of modelling, but explains neither how to generate or visualize such information [Egy01]. MT and TM cooperate together to present a simple visualization of tracing information that also allows users to build up a detailed picture of how the transformation executed.

In order to visualize tracing information, one needs to understand how this information is stored. Transformation instances contain two separate lists of equal length related to tracing information. The first list contains tuples (encoded as lists) relating source and target elements. The second list contains the name of the transformation rule which created the corresponding entry in the first list. The fact that they are stored separately is a simple implementation detail — conceptually these two lists can be considered to constitute one single piece of information.

The `TM.Visualizer` module defines a function `visualize_trace(transformation)` which takes a transformation instance and visualizes it complete with tracing information. The result of visualizing the tracing information for the example model of section 5.3.6 can be seen in figure 5.7. The original source model is on the left in blue, with the target model on the right in green. The black lines between source and target elements are the traces between source and target elements. Individual traces always run from a single source element to a single target element. Each trace has a name of the form t*n* where *n* is an integer starting from 1. The integer values reflect the traces position in the execution sequence; trace numbers can be compared to one another to determine whether a rule execution happened earlier or later in the execution sequence. Trace names can be looked up in the

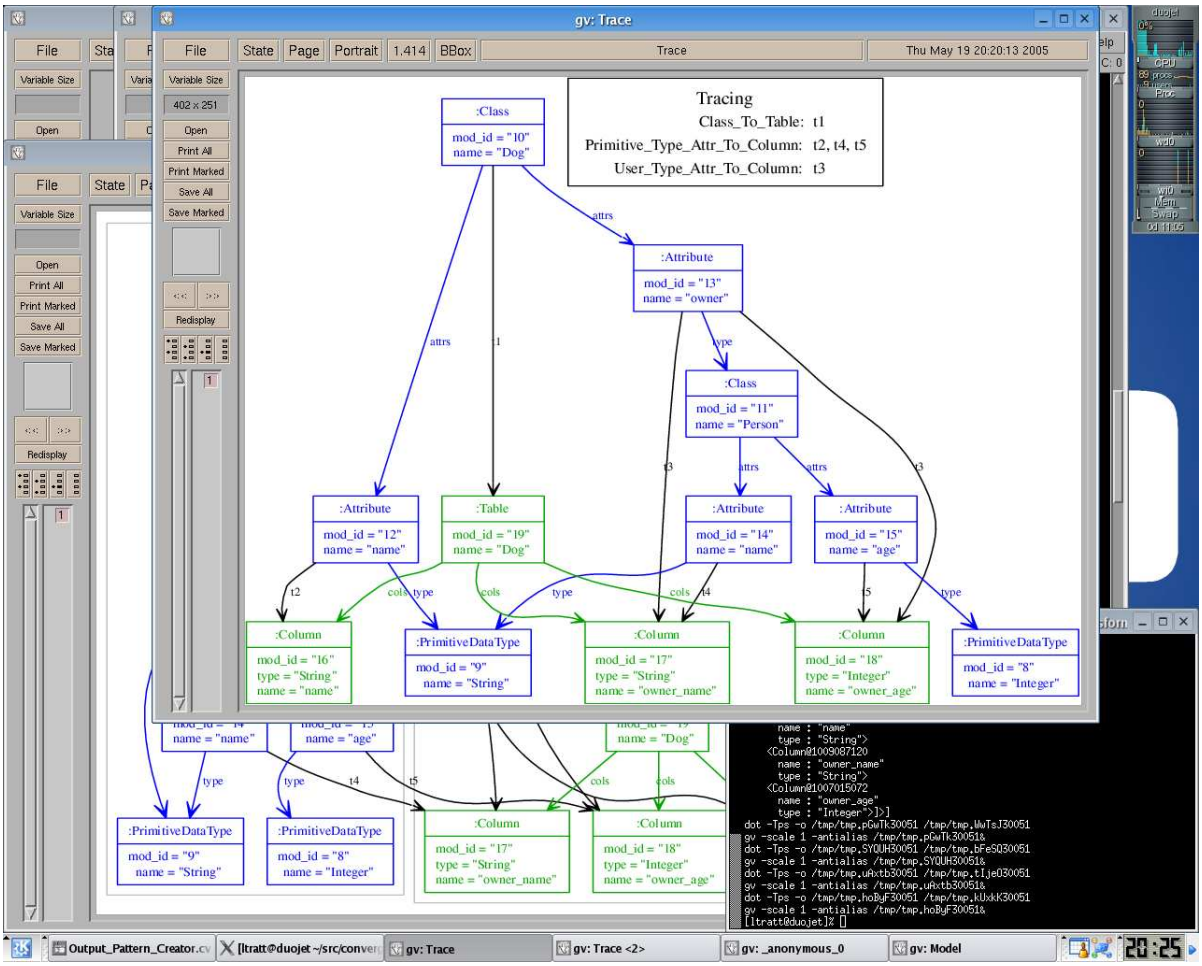


Figure 5.6.: MT running.

'Tracing' table at the top right of figure. The tracing table contains the name of each rule which was executed at least once during the transformation. Against each rule name are the names of traces; each trace name represents an execution of that rule. Note that a single rule execution can create more than one trace; however each trace created in a single execution will share the same name.

Although the visualization of tracing information may seem simple, it allows one to infer a great deal of useful information about the execution of a transformation. This information is useful both for analysis and debugging of a transformation. At a simple level, one can use the names of tracing information to determine which rule consumed which source elements and produced which target elements. For example the 't1' trace from the source class to the target table is a result of the `Class_To_Table` rule. One can also deduce from this traces name that it was the result of the first rule execution in the system. Similarly since two traces share the name 't4', one can determine that a rule – in this case `User_Type_Attr_To_Column` – created more than one target element in a single execution.

Although this subsection has talked about how tracing information is stored and visualized, and

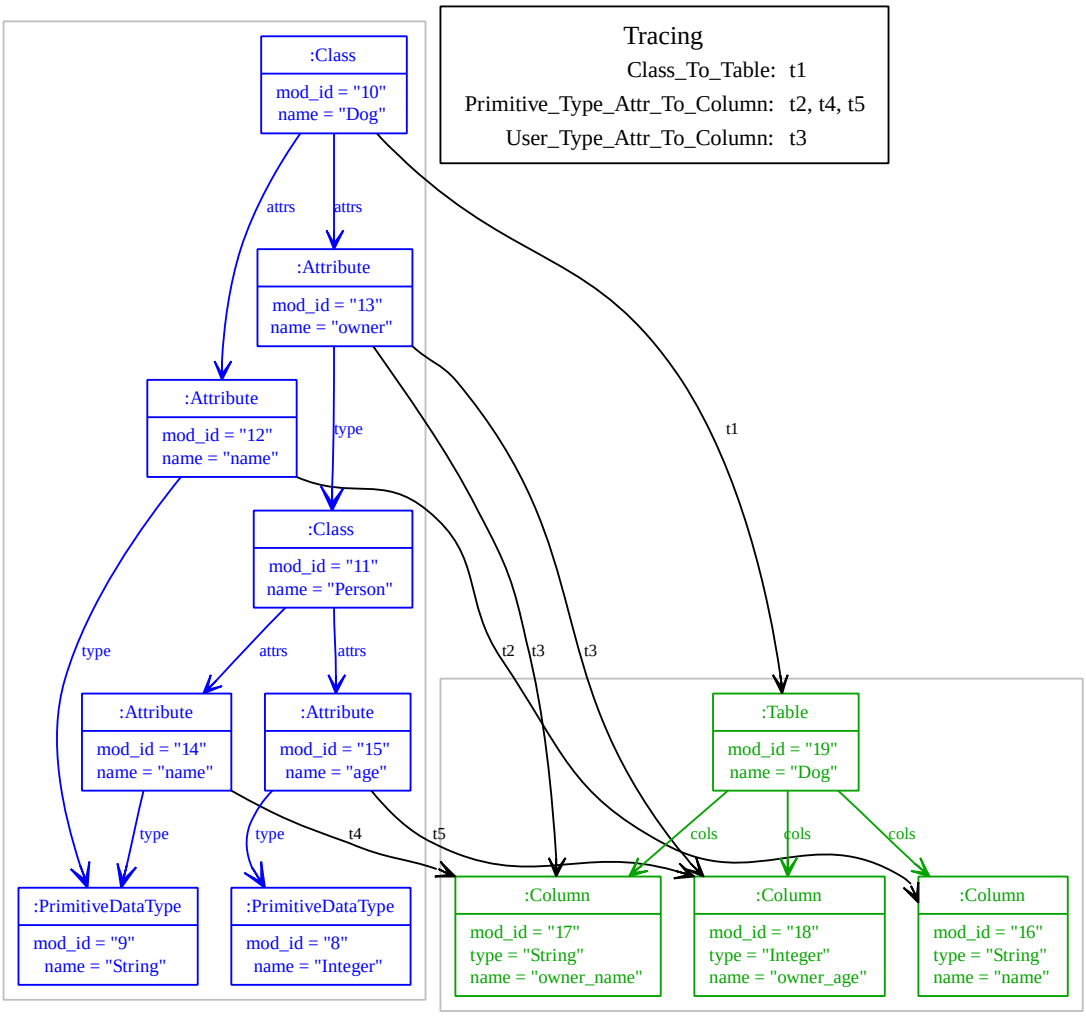


Figure 5.7.: Visualizing tracing information.

what the visualization can be used for, it has not discussed how the tracing information is created. Section 5.4.2 explains how tracing information is created, and how users can control its creation.

Alternative visualizations

The tracing information in figure 5.7 is visualized with the source and target models formatted exactly as they were when presented individually in figures 5.4 and 5.5. Whilst this visualization works well for small transformations, larger transformations with greater volumes of tracing information tend to become unreadable as the strict formatting of the source and target models forces traces to overlap with each other. There is thus an alternative form of visualization available via the visualizers `visualize_trace_clustered`[4] function where the source and target elements can be formatted directly alongside one another. Figure 5.8 shows this alternative visualization. Note that the diagram colouring now becomes critical to distinguish source and target model elements from one

[4]The 'clustered' part of this function name reflects the mechanism used in GraphViz to enable this layout.

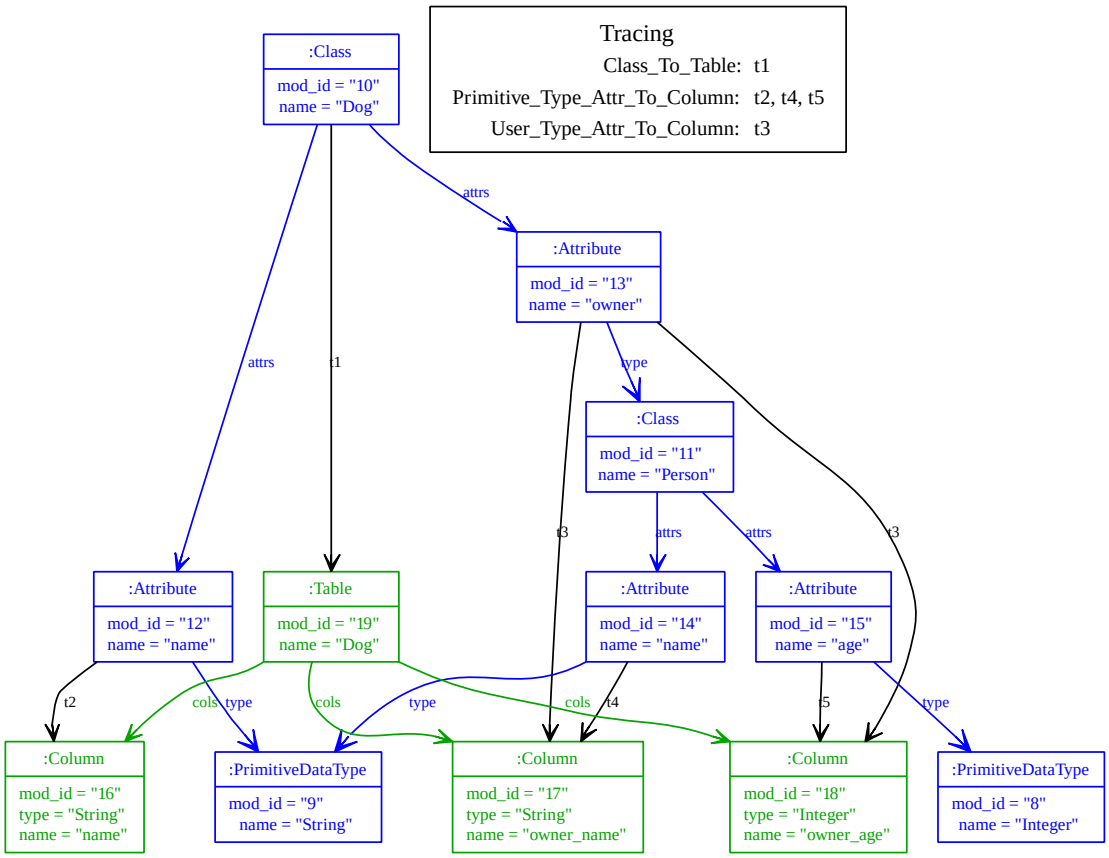


Figure 5.8.: Visualizing tracing information with the free layout of source and target model elements.

another. Due to its general lack of cluttering, this is generally the preferred visualization when tracing information is involved, and is used in the remainder of this thesis.

5.4.2. Standard tracing information creation mechanism

Section 5.4.1 showed how MT and TM can visualize the tracing information automatically created by MT transformations. In this subsection I outline how the default tracing information is created by MT. Most, if not all, model transformation approaches are currently somewhat vague on this subject. There is therefore little prior art to use as a basis, or point of comparison, for any such mechanism. MT takes a simple approach to the problem to ensure that its behaviour is predictable from a users perspective – this is vital to ensure that users can make informed choices about when and where to add or override tracing information (see section 5.4.3).

As standard, tracing information in a rule is created between all source elements matched by non-nested model element patterns, and all target elements produced by model element expressions, nested or otherwise. Non-nested model element patterns are defined to be those which are not nested within another model element pattern. For example in the following model element pattern, tracing information will be created only from instances of the `Dog` model class:

```
(Dog, <d>)[name == <n>, owner == (Person)[name != "Fred"]]
```

It may initially seem somewhat arbitrary to try to minimise the source elements used in tracing information whilst maximising the target elements used. The reason for minimising the source elements used is due to a simple observation: individual source elements are often matched in more than one rule execution. This then causes some source elements to be the source for large numbers of traces which can obscure the result of the transformation. Empirical observations of MT transformations suggest that when model elements are matched via nested model element patterns, they are also matched as a non-nested model element pattern during a separate rule execution. In the case of target elements, a different challenge emerges. Rather than trying to create an 'optimum' amount of traces one wishes to ensure that, as far as is practical, every target element has at least one trace associated with it. Since target element expressions are inherently localised to individual rule executions it is highly unusual for an element created by such an expression to be the target of more than one trace. Thus it is important to ensure that nested target element expressions have traces associated with them. Section 5.6.15 shows how nested model element patterns can be made to contribute towards tracing information if desired (that section also shows the large number of extra, largely uninteresting, traces created).

The standard tracing information mechanism can be seen in practice by comparing the visualized trace information of figure 5.8 with the transformation that created it in section 5.3.5.

5.4.3. Augmenting or overriding the standard mechanism

Whilst the standard tracing creation mechanism performs well in many cases, users may wish to augment, or override, the default tracing information created. Users may wish to add extra tracing information to emphasise certain relationships within a transformation, or to remove certain tracing information that unnecessarily clutters the transformation visualization. MT provides a simple capability for augmenting, or overriding, the default tracing information created by the standard mechanism.

For example, using the MT example presented in section 5.3.5 as a base, imagine that one wishes to add extra traces between the source class and all target columns. In order to achieve this one makes use of the optional `tracing_add` clause on MT rules. This clause must contain a single Converge expression which evaluates to a tuple relating source and target model elements. The tuple is then added to the tracing information created automatically by the rule. The new `Class_To_Table` rule looks as follows:

```
rule Class_To_Table:
  srcp:
    (Class, <c>)[name == <n>, attrs == <A>]
```

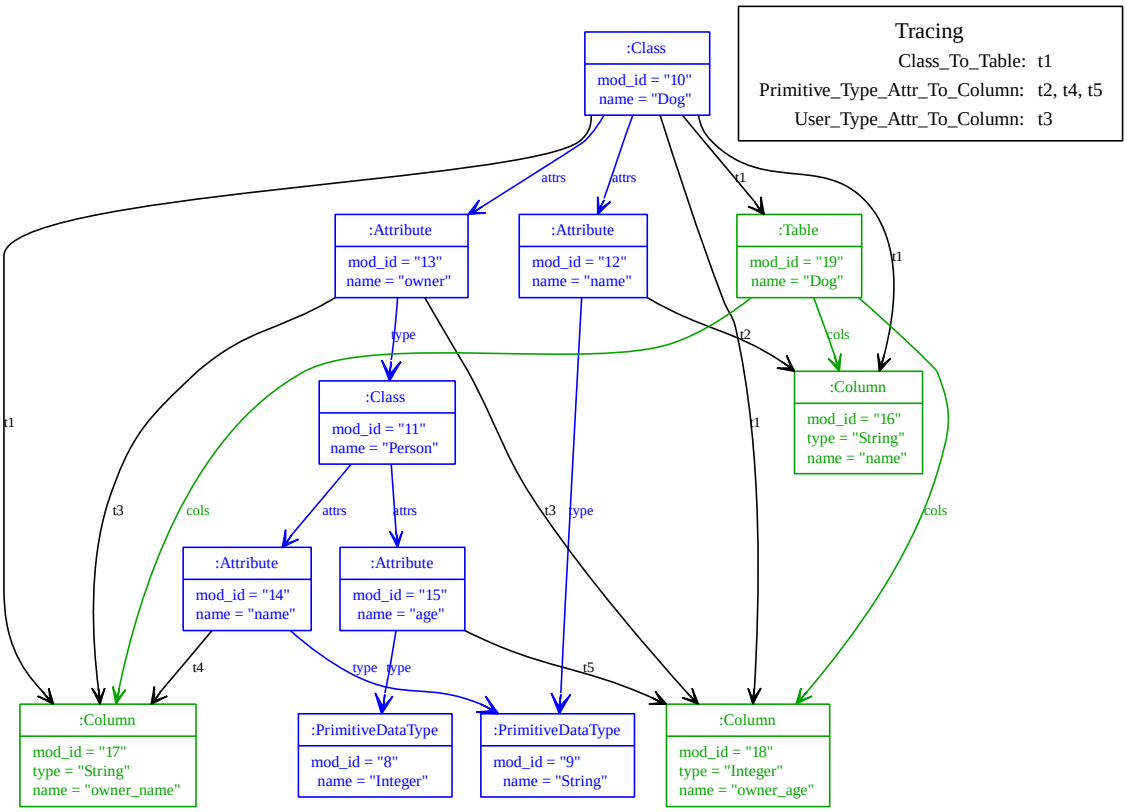


Figure 5.9.: Augmenting the default tracing information.

```
tgtp:
  (Table)[name := n, cols := columns]

tgt_where:
  columns := []
  for attr := A.iterate():
    columns.extend(self.transform([""], [attr]).flatten())

tracing_add:
  [[c], columns]
```

Note that since `c` is a single element it needs to be placed within a list to create a valid trace tuple. The tuple in the `tracing_add` clause is then added to the tracing information automatically created by the rule – hence the new traces have the same tracing number (in this case 't1') as the default traces for the rule execution. Figure 5.9 shows the resulting visualization of the transformation with the extra tracing information added in.

In some circumstances, users may wish to entirely override the default tracing information, rather than simply augmenting it. The `tracing_override` clause in a rule turns off the rules default tracing generation, replacing it with the tuple returned by the single Converge expression in the clause. `tracing_add` and `tracing_override` are thus mutually exclusive clauses within a rule. Whilst maintaining the additional tracing information created by the modified `Class_To_Table` rule, assume one now wishes the other two rules in the transformation to be prevented from generating any tracing information at all. In order to achieve this, `tracing_override` clauses

which contain tuples relating the empty set of source elements to the empty list of target elements are defined. The modified rules are as follows:

```
rule User_Type_Attr_To_Column:
  srcp:
    (String, <prefix>)[]
    (Attribute)[name == <n>, type == (Class)[name == <cn>, attrs == <CA>]]

  tgtp:
    self.transform([concat_name(prefix, n)], [ca]) for ca := CA.iterate()

  tracing_override:
    [[], []]

rule Primitive_Type_Attr_To_Column:
  srcp:
    (String, <prefix>)[]
    (Attribute)[name == <n>, type == (PrimitiveDataType)[name == <pn>]]

  tgtp:
    [(Column)[name := concat_name(prefix, n), type := pn]]

  tracing_override:
    [[], []]
```

The result of running the transformation with its three rules altered can be seen in figure 5.10. As this example shows, users can completely customise the tracing information created by MT to their own needs.

5.5. Towards more sophisticated transformations

The previous section introduced the basics of the MT language, via a simple version of the running example. In this section, I delve into some of the more advanced aspects of the MT language which allow more complex and sophisticated transformations to be expressed. In order to explore these aspects fully, I first present a more complex version of the running example.

5.5.1. Extending the running example

In this subsection I define the ‘advanced’ variant of the running example. The overall idea is, as before, to translate UML class models into relational database models. In order to make the example more challenging, the ‘Simple UML’ meta-model is extended in several ways as can be seen in figure 5.11. These extensions extend the required transformation as follows:

- Associations are added to the meta-model. Associations add a significant degree of complexity to the meta-model because a class’s ‘real’ attributes are determined by the union of the attributes it directly links to, and the associations for which it is a source.
- Attributes can be marked as being part of a class’s primary key by having the `is_primary` attribute set to true. Note that associations play no part in determining a class’s primary key.

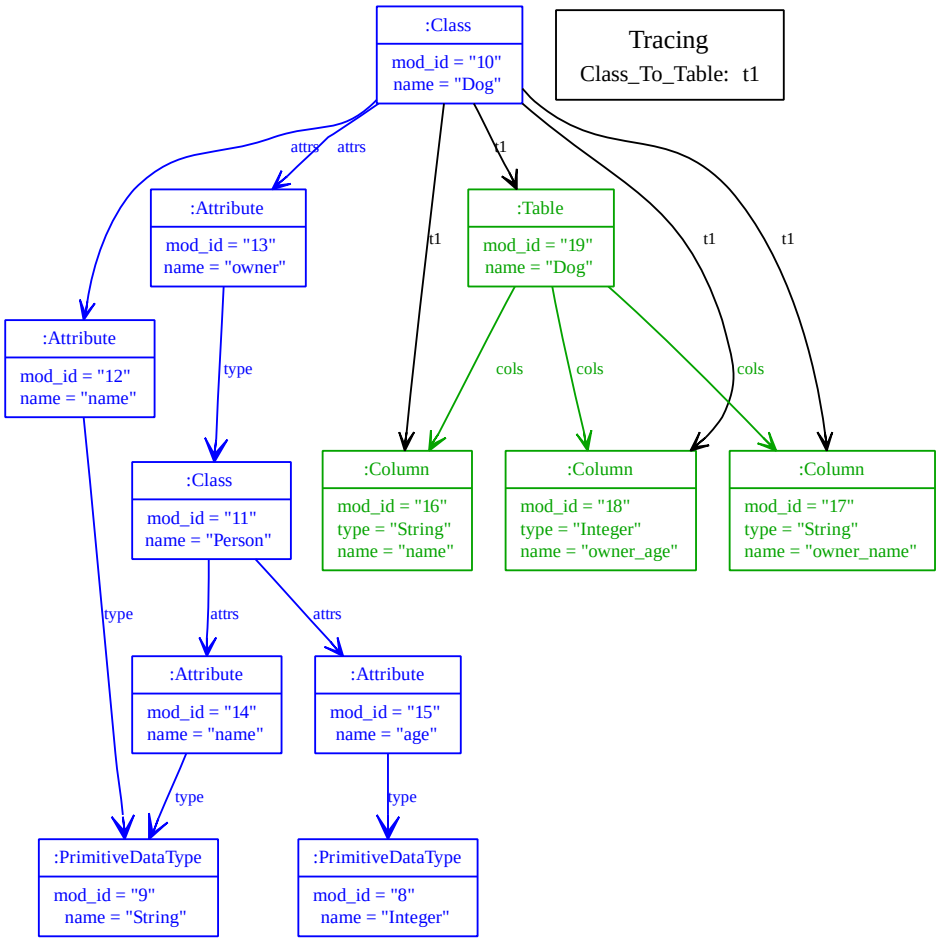


Figure 5.10.: Augmenting and overriding the default tracing information.

- Classes which have the `is_persistent` attribute set to true will be converted to tables; references to such classes (via attribute types or associations) will result in the classes primary key attributes being to converted to columns used as a foreign key. Class's which do not have the `is_persistent` attribute set to true will not be transformed into tables, and will have their attributes drilled into, as in the simple transformation.

The relational database meta-model is also extended, as shown in figure 5.12. The extended meta-model allows tables to define primary keys and foreign keys. Note that, since the TM data model allows nested data types to be expressed, foreign keys are defined as a sequence of sequences of columns.

5.5.2. Pattern multiplicities

One of the problems noted in section 5.2.4 with the QVT-Partners approach is that model element patterns can only match against a fixed number of elements. Some very simple transformations naturally consist only of rules which match against a fixed number of elements in the source model.

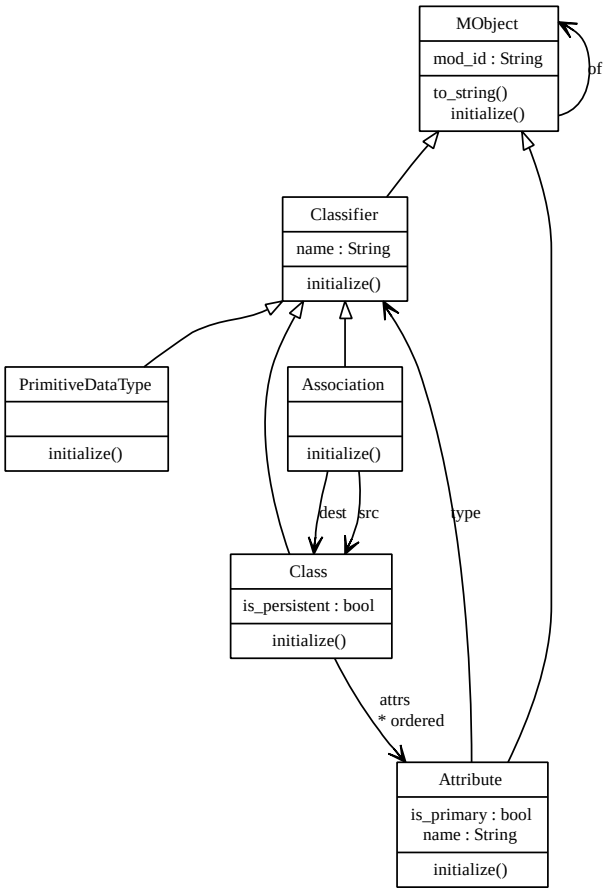


Figure 5.11.: Extended ‘Simple UML’ meta-model.

However, many, if not most, non-trivial transformations contain rules which need to match against an arbitrary number of source elements. Expressing such transformations in the QVT-Partners approach can be cumbersome.

To solve this problem, MT adapts the concept of multiplicities found in many textual regular expression languages. Each source pattern in MT can optionally be given a *multiplicity*. Multiplicities specify how often a given source pattern can, or must, match against its source elements. Multiplicities are therefore a constraint on the universe of model elements passed in the parameter corresponding to the patterns position in the `srcp` clause. Each pattern in a `srcp` clause can optionally be suffixed with a multiplicity and an associated variable binding. The following example of a pattern multiplicity will match zero or more associations, assigning the result of the match to the `assocs` variable:

```
(Association, <assoc>)[name == n] : * <assocs>
```

The syntax for multiplicities is inspired by Perl’s regular expression languages. The following multiplicities, and possible qualifiers, are defined in MT:

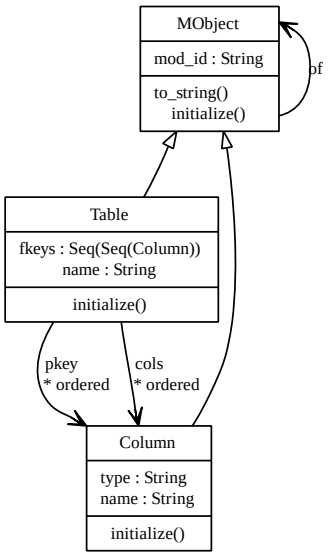


Figure 5.12.: Extended relational database meta-model.

m	Must match exactly *m* source elements.
*	Will match against zero or more source elements.
* !	Must match against every source element.
* ?	Will match against the minimum possible number of source elements.
m .. *n*	Must match no less than *m*, and no more than *n* source elements.
m .. *n* ?	Will match against the minimum number of source elements once m elements have been matched, but will not exceed *n* matches.
m .. *	Must match no less than *m* elements.
m .. * ?	Will match against the minimum number of source elements once m elements have been matched.

As with Perl textual regular expressions, multiplicities default to ‘greedy’ matching — that is, they will match their pattern against the maximum number of elements that causes the multiplicity to be satisfied. When backtracking in a `srcp` clause calls upon a multiplicity to provide alternative matches, it then returns matches of lesser lengths. The concept of greedy and non-greedy matching is however much simpler in the case of textual regular expressions since text is an inherently ordered data type. Thus the length of matches is calculated by determining how many characters past a fixed starting point a match extends. In contrast to this, model elements have no order with respect to one another, and thus MT has to take a very different approach to the concepts of greedy and non-greedy matches. MT defines the length of a multiplicities match as the number of times the multiplicity matched; however since model elements are not ordered, this does not present an obvious way of returning successively smaller matches. In order to resolve this problem in the case of greedy

matching, MT creates the powerset of matches, and iterates over it, successively returning sets with smaller number of elements when called upon to do so. Note that whilst MT guarantees that with greedy matching $|match_n| \geq |match_{n+1}|$, it makes no guarantees about the order that sets of equal size in the powerset will be returned.

The `?` qualifier reverses the default greedy matching behaviour, attempting to match the minimum number of elements that causes the multiplicity to be satisfied, successively returning sets of greater size from the powerset when called upon to do so. The `!` qualifier is the 'complete' qualifier which ensures that the pattern matches successfully against every model element passed in the patterns appropriate argument. Whilst the `?` qualifier, in a slightly different form, is standard in most textual regular expression languages, the `!` qualifier is specific to MT.

Variable bindings in the presence of multiplicities

Variable bindings in patterns suffixed by multiplicities need to be treated differently from variables in bare patterns. When a multiplicity is satisfied, its associated variable binding is assigned a list of dictionaries. Each dictionary contains the variable bindings from a particular match of the pattern. The need for different treatment of variable bindings inside and outside multiplicities is most easily shown by examining what would happen if they were treated identically. Consider the following incorrect MT code:

```
(Association)[src == (Class)[name = n]] : * <assocs>
(Class, <c>)[name == n]
```

A first glance may suggest that when the rule these patterns are a part of runs, `c` will be set to the class which has the same name as the associations source class. However, the example is nonsensical since `n` has no single value. Indeed `n` may have no value at all, since it will be bound to zero or more class names as the multiplicity attempts to match the model pattern as many times as possible. As this example shows, `n` has no meaning outside of the multiplicity it is bound in; however it clearly has a meaning in the context of the multiplicity.

In order to resolve this quandary, MT takes a two stage approach. Within multiplicities, local variable bindings are accessed as normal. At the end of each successful match, MT creates a dictionary relating variable binding names to their bound values. The list of these values is then assigned to the variable binding associated with the multiplicity. Thus the variable bindings for each individual match can be accessed. To illustrate this, I reuse the original multiplicities example:

```
(Association)[src == (Class)[name = <n>]] : * <assocs>
```

Printing the `assocs` variable would lead to output along the following lines:

```
[Dict{"n" : "orders"}, Dict{"n" : "parts"}]
```

The `MT` module provides a convenience function `mult_extract(bindings, name)` which iterates over a list of dictionaries, as generated by a multiplicity, and extracts the particular binding `name` from each dictionary, returning a list. A standard idiom in MT is to use this function with a self variable binding in a model element pattern, which allows a user to determine all the model elements matched by a particular pattern multiplicity.

5.5.3. Extended example

In this subsection I show the MT version of the extended example. The added complexity in this version of the transformation over the original simpler version is due to three considerations:

1. Classes can not be transformed in isolation – all associations for which a class is the source must be considered in order that the table that results from a class contains all necessary columns.

2. Classes which are marked as persistent must be transformed substantially different from those not marked as persistent.

3. Foreign keys and primary keys reference columns. It is important that the column model elements pointed to by a table are the appropriate model elements, and not duplicates.

The MT example is as follows:

```
$<MT.mt>:
  transformation Classes_To_Tables

  rule Persistent_Class_To_Table:
    srcp:
      (Class, <c>)[name == <n>, attrs == <attrs>, is_persistent == 1]
      (Association, <assoc>)[src == c] : * <assocs>

    tgtp:
      (Table)[name := n, cols := cols, pkey := pkeys, fkeys := fkeys]

    tgt_where:
      cols := []
      pkeys := []
      fkeys := []
      for aa := (attrs + MT.mult_extract(assocs, "assoc")).iterate():
        a_cols, a_pkeys, a_fkeys := self.transform([""], [aa])
        cols.extend(a_cols)
        pkeys.extend(a_pkeys)
        fkeys.extend(a_fkeys)

  rule Primary_Primitive_Type_Attribute_To_Columns:
    srcp:
      (String, <prefix>)[]
      (Attribute)[name == <attr_name>, type == (PrimitiveDataType)[name == \
        <type_name>], is_primary == 1]

    tgtp:
      [col]
      [col]
      []

    tgt_where:
```

```
    col := (Column)[name := concat_name(prefix, attr_name), type := \
      type_name]

rule Non_Primary_Primitive_Type_Attribute_To_Columns:
  srcp:
    (String, <prefix>)[]
    (Attribute)[name == <attr_name>, type == (PrimitiveDataType)[name == \
      <type_name>], is_primary == 0]

  tgtp:
    [(Column)[name := concat_name(prefix, attr_name), type := type_name]]
    []
    []

rule Persistent_User_Type_Attribute_To_Columns:
  srcp:
    (String, <prefix>)[]
    (Attribute, <attr>)[name == <attr_name>, type == (Class, <class_>) \
      [name == <class_name>, attrs == <attrs>, is_persistent == 1]]

  tgtp:
    cols
    []
    [cols]

  tgt_where:
    cols := []
    for attr := attrs.iterate():
      a_cols, a_pkeys, a_fkeys := self.transform([concat_name(prefix, \
        attr_name)], [attr])
      cols.extend(a_pkeys)

rule Non_Persistent_User_Type_Attribute_To_Columns:
  srcp:
    (String, <prefix>)[]
    (Attribute, <attr>)[name == <attr_name>, type == (Class, <class_>) \
      [name == <class_name>, attrs == <attrs>, is_persistent == 0]]

  tgtp:
    cols
    []
    []

  tgt_where:
    cols := []
    for attr := attrs.iterate():
      a_cols, a_pkeys, a_fkeys := self.transform([concat_name(prefix, \
        attr_name)], [attr])
      cols.extend(a_cols)

rule Persistent_Association_To_Columns:
  srcp:
    (String, <prefix>)[]
    (Association)[name == <attr_name>, dest == (Class, <class_>)[name == \
      <class_name>, attrs == <attrs>, is_persistent == 1]]

  tgtp:
    cols
    []
    [cols]

  tgt_where:
    cols := []
    for attr := attrs.iterate():
      a_cols, a_pkeys, a_fkeys := self.transform([concat_name(prefix, \
        attr_name)], [attr])
      cols.extend(a_pkeys)

rule Association_Non_Persistent_Class_To_Columns:
```

```
    srcp:
      (String, <prefix>)[]
      (Association)[name == <attr_name>, dest == (Class, <class_>)[name == \
        <class_name>, attrs == <attrs>, is_persistent == 0]]
      (Association, <assoc>)[src == class_] : * <assocs>

    tgtp:
      cols
      []
      fkeys

    tgt_where:
      cols := []
      fkeys := []
      for aa := (attrs + MT.mult_extract(assocs, "assoc")).iterate():
        a_cols, a_pkeys, a_fkeys := self.transform([concat_name(prefix, \
          attr_name)], [aa])
        cols.extend(a_cols)

rule Default:
  srcp:
    (MObject)[]

  tgtp:
    null
```

In order to run this transformation, a list of top-level elements (classes and associations) should be passed to it. Unlike the simple version of the example, there is no need to designate one particular class as being the 'start' class for the transformation. The output of the transformation will consist of a number of tables.

One feature in particular requires explanation to make sense of this transformation. Many of the rules have more patterns than there are arguments passed to the `transform` function. For example, the `Association_Non_Persistent_Class_To_Columns` rule defines three patterns but the `transform` function is never called with more than two arguments – it would thus seem impossible for this rule to ever execute. However, MT defines that when a rule is passed fewer arguments than it has parameters, the root set of source elements is substituted for each missing argument. This is effectively an escape mechanism allowing rules access to the complete source graph. Although this might seem an arbitrary design choice, without such a mechanism transformations such as this would be complicated by the need to pass the root set of source elements to every rule execution.

The overall structure of this transformation is hopefully relatively straightforward. The `Persistent_Class_To_Table` rule ensures that each class marked as being persistent in the source model is transformed into a table in the target model. It takes a persistent class, and finds all of the associations for which the class is a source; it then iterates over the union of the classes' attributes and associations for which it is a source, transforming them into columns. All of the other rules take in a string prefix (representing the column prefix being constructed as the transformation drills into user types), and an attribute or association (and, in the case of the `Association_Non_Persistent_Class_To_Columns` rule, an additional set of associations) and produces three things: a

list of normal table columns; a list of primary key columns; a list of foreign key columns. The final rule in the transformation `Default` is a 'catch all' rule that takes in model elements from the root set which not matched by other rules – non-persistent classes and associations – and transforms them into the `null` object; this causes MT to discard the result of the transformation rule, and not create any tracing information. The `Default` rule is necessary to ensure that such elements in the root set of source elements do not cause the transformation to raise a `Can not transform` exception.

Figure 5.13 shows a visualization of a particular execution of the transformation. The size of the source model has been increased to the maximum that can be sensibly visualized on paper, to provide some reassurance that MT can cope with transformations beyond a small handful of elements. Note that when freed of paper-based space constraints, and the visualization technique can easily cope with much larger source models.

5.5.4. Pruning the target model

One thing not immediately obvious from viewing figure 5.13 is that the final target model is not a union of the model elements produced in every rule execution. In fact, if one were to take the union of model elements produced by every rule execution, the target model would contain many superfluous model elements. The reason for this can be seen by examining a rule such as `Persistent_Association_To_Columns`. This rule calls the `transform` function but then effectively discards some of the model elements produced by this call (the rule in question cares only about primary key columns, and ignores non-primary key columns). Knowing that, as an implementation detail, TM assigns each new model element a unique and monotonically increasing identifier, one can see from figure 5.13, that some elements have been discarded, due to the non-contiguous model identifiers in target model elements. For example, the lowest identifier for a target model element is 29 and the highest 47, but identifiers such as 42 are missing in the figure – these are elements that were produced by a rule execution, but discarded by other rules.

MT's approach to achieving the final target model involves firstly taking the model elements produced by transforming each element in the root source set. It then then uses these elements as the root nodes in a simple graph walking scheme. Only target model elements which are reachable from these elements are considered to be in the eventual target model. Note that scheme does allow the eventual target model to consist of unconnected subgraphs.

5.5.5. Combinators

One of the most interesting features in the QVT-Partners approach are combinators. Section 5.2.3 showed the `or` combinator; the QVT-Partners approach also defines `and` and `not` combinators. The

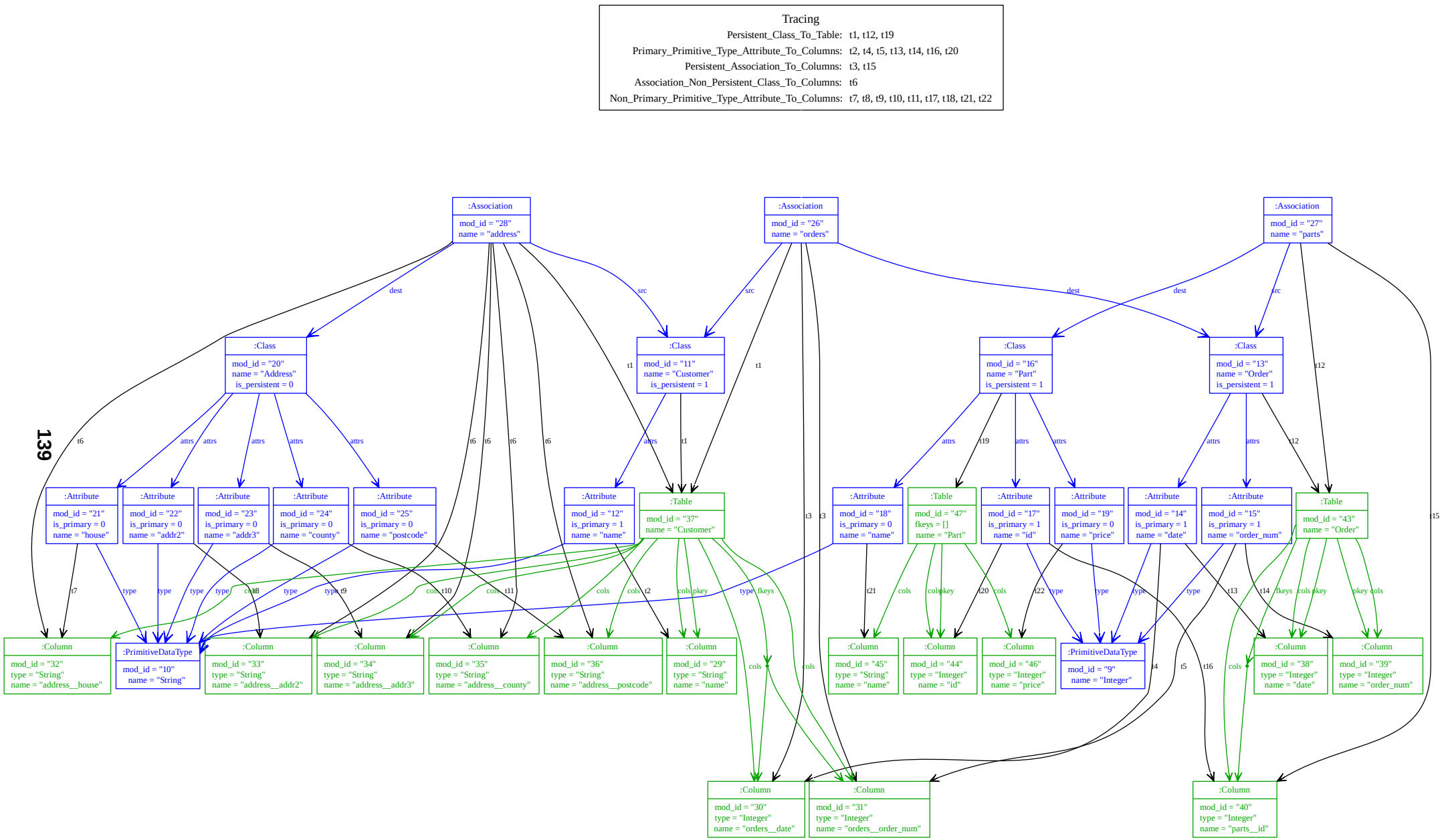


Figure 5.13.: An example execution of the extended transformation.

combinators work largely as one might expect given their names. For example the `and` combinator takes two or more rule invocations, and succeeds only if each invocation succeeds.

Since MT rules are able to utilise the standard Converge notions of success and failure, the base combinators from the QVT-Partners approach can be encoded directly in MT using the `not`, disjunction and conjunction operators for `not`, `or`, and `and` respectively. The following contrived transformation rule will match against a class iff one of its attributes can be transformed by one or the other of the `R1` or `R2` rules:

```
rule X:
  srcp:
    (Class)[attributes = {a | 0}]

  src_when:
    self.R1(a) | self.R2(a)
```

The QVT-Partners approach defines extra semantics for the `and` combinator which automatically merges together the outputs of different rules. In the general case, I believe that such functionality is undesirable since the merging of outputs can only sensibly be determined at the fine-grained level by transformation writers themselves. However building a 'merging' combinator on top of the existing functionality is relatively simple, since it merely involves storing and then merging the result of each expression in a conjunction.

Although the treatment of combinators in MT is currently simplistic, the direct encoding of these features in terms of primitive Converge features is interesting. Whereas the QVT-Partners combinators are new primitives in the language, MT is able to directly utilise Converge features. I believe a fruitful area of future research will be to investigate more powerful combinators, with a view to including the most useful in a standard library. Work such as that of Bézivin on model unification [BÓ5], and Chivers and Paige [CP05] may have relevance to the investigation of combinators.

5.6. Implementation

In this section I discuss some of the most interesting aspects of the MT implementation. Since the implementation follows the typical structure of DSL implementation functions as outlined in section 4.4.1 – and as seen concretely in TM (section 4.5) – many aspects of the implementation have already been presented elsewhere in this thesis. In this section I outline the novel aspects of the implementation, relative to what has already been presented. Appendix B.1 contains the MT grammar which is referenced throughout this section.

5.6.1. Outline of the implementation

The translation of an `MT.MT` block into Converge is relatively straight forward at a high-level. An MT transformation is translated to a single class with a number of standard functions (e.g. `transform` and `get_target`, as seen earlier), fields for holding tracing information and so on, and a function for each rule in the transformation. The translation records the names of all transformation rules as a list in the `_rule_names` field within the transformation class; the list retains the rules' order in the source file.

The following subsections show how rules are translated and the definitions of the standard functions.

5.6.2. Translating rules

The translation of a rule into a function conceptually follows the path outlined by example in section 5.3.1. The translated function takes a variable number of arguments, each of which must be a list containing the 'universe' of elements which each pattern in the rule can match against. If the translated source model clauses fail to match against the arguments passed to the translated function, the rule fails. If these clauses succeed, they return the set of model elements matched by model element patterns, and a dictionary of bindings. The dictionary of bindings is passed to the translated `tgtp` and `tgtp_where` clauses which produce and return a list of target elements. The matched model element patterns and target elements are then used to create a suitable tuple for the transformations tracing execution, before the rule returns the target elements produced.

A simplified version of the outer translation of a rule is as follows:

```
1  func _t_mt_rule(node):
2    // mt_rule ::= "RULE" "ID" ":" "INDENT" mt_src "NEWLINE" mt_out "DEDENT"
3    return [|
4      bound_func $<<CEI.name(node[2].value)>>(*objs):
5        if not mep_objects, bindings := $<<self.preorder(node[5])>>.apply(objs):
6          return fail
7
8        if not target_elements := $<<self.preorder(node[7])>>(bindings):
9          raise Exceptions.Exception(Strings.format(\${}<<CEI.lift( \
10           "Failed to generate anything for '%s'.")>>, objs.to_str()))
11
12       self._tracing.append([mep_objects, target_elements])
13
14       return target_elements
15   |]
```

Line 5 translates the rules source model clauses; note that if the source model clauses fail, the entire rule fails. If the source model clauses succeed, then the translation of the target model clauses in line 8 is executed. Failure of the target model clauses is deemed a fatal error, and an exception is raised. Note that there is no concept of backtracking between the target and source model clauses – once the source model clause has successfully matched, the target model clauses are executed. The final part

of the rule in line 12 creates the necessary tracing information.

The translation of a rules source model clauses contains subtleties to ensure that backtracking amongst patterns works correctly; this is explored in the upcoming subsections. Since the target model clauses are already fairly standard imperative code (with the addition of model element expressions), their translation is largely uninteresting and consequently elided. However, the translation of all clauses is complicated by the need to embed normal Converge code, and to ensure that there are no unintended interactions between translated and embedded code (see sections 5.6.12 and 5.6.14).

5.6.3. Translating a rules source model clauses

A rule potentially contains two source model clauses: the `srcp` and `srcp_when` clauses, the latter of which is optional. Each pattern in the `srcp` clause is translated into a generator function which takes in a list of model elements, and a Converge dictionary of bindings. Each time the pattern matches successfully it returns a list containing three items: a list of the model elements matched by model element patterns, a dictionary of new bindings, and the object the pattern evaluated to. The last of these is largely an internal detail needed to support the nesting of patterns. Section 5.6.4 contains more detail on the translation of patterns.

Since each pattern is a generator, it needs to be placed within a `for` construct to ensure that possible matches can be generated. When more than one pattern is present in a `srcp` clause, patterns must be translated 'inside out' into nested `for` constructs; that is, the first pattern will be the outermost `for` construct, and the last pattern the innermost. This slightly complicates the translation, which iterates over the patterns in the order they are presented in the parse tree. In order to achieve the desired effect, a standard idiom is used. Patterns are first translated to a temporary list. The translated `srcp_when` clause, if it exists, is used as the innermost construct. The temporary list is then iterated over in reverse order with each iteration placing the result of the previous iteration inside a `for` construct. This idiom is highly useful, and also shows a simple example of a DSL translation where the translated code does not directly reflect the order of its source. Noting that a lists `riterate` function iterates in reverse order over the list, the simplified version of this translation is as follows:

```
translated_patterns := [ordered translated patterns]
patterns_expr := [| $<<self.preorder(src_when clause)>> |]
for translated_pattern := translated_patterns.riterate():
  patterns_expr := [|
    for $<<translated_pattern>>:
      $<<patterns_expr>>
  |]
```

This simple translation only deals with part of the problem caused when the failure of a pattern leads to backtracking to an earlier clause. As each pattern matches, it returns a list containing three items: a list of the model elements matched by model element patterns, a dictionary of new bindings, and

the object the pattern evaluated to. As each pattern in the `srcp` clause is matched, the rules records of matched model element patterns and variable bindings grow. When a pattern fails, the list of elements it matched by model element patterns and variable bindings it created need to be 'undone' from the rules' records. Since the failure of one pattern may cause the failure of an arbitrary number of preceding patterns, the 'undo' mechanism also needs to work to an arbitrary depth.

MT makes use of Converge's variable capturing and scoping rules to implement a simple and efficient undoing mechanism. Each rule defines two variables `matched_mp_elems` and `bindings` which store the rules evolving list of matched model pattern elements and variable bindings. These variables are available to each translated pattern. Each translated pattern then defines two variables private to that pattern (hidden via Converge's scoping rules; see section 4.2.5), named `matched_mp_elems_backup` and `bindings_backup`. As the names of these variables may suggest, they are used to store the values of the `matched_mp_elems` and `bindings` variables as they were before the pattern is matched; if the pattern did not match successfully or is required to generate new matches, they are used to restore their value. A slightly elided version of the translation function is as follows:

```
func _t_mt_src(node):
  // mt_src  ::= mt_srcp mt_srcc
  // mt_srcp ::= "SRCP" ":" "INDENT" pt_spattern { "NEWLINE" pt_spattern }*
  //             "DEDENT"

  translated_patterns := [ordered translated patterns]
  if node[2].len() > 1:
    // mt_srcc ::= "NEWLINE" "SRC_WHEN" ":" "INDENT" expr "DEDENT"
    patterns_expr := [|
      if $<<self.preorder(node[2][5])>>:
        return [&matched_mp_elems, &bindings]
     |]
  else:
    // mt_srcc ::=
    patterns_expr := [| return [&matched_mp_elems, &bindings] |]

  for i := (translated_patterns.len() - 1).to(-1, -1):
    patterns_expr := [|
      if $<<CEI.lift(i)>> < &args.len():
        elements := &args[$<<CEI.lift(i)>>]
      else:
        elements := &self._root_set

      matched_mp_elems_backup := &matched_mp_elems
      bindings_backup := &bindings
      for new_matched_mp_elems, new_bindings, matched_elem := \
        $<<translated_patterns[i]>>(&bindings, elements):
        &matched_mp_elems := &matched_mp_elems + new_matched_mp_elems
        &bindings := &bindings + new_bindings
        $<<patterns_expr>>
        &matched_mp_elems := matched_mp_elems_backup
        &bindings := bindings_backup
    |]

  return [|
    func (*args):
      if args.len() > $<<CEI.lift(translated_patterns.len())>>:
        return fail
```

```
        matched_mp_elems := Set{}
        bindings := Dict{}

        $<<patterns_expr>>

        return fail
    |]
```

5.6.4. Translating patterns

In this subsection I show how patterns are translated in MT (the translation of pattern multiplicities is detailed in section 5.6.10). Each pattern is translated into a generator function which takes in a list of model elements, and a Converge dictionary of bindings. Each time the pattern matches successfully it returns a list containing three items: a list of the model elements matched by model element patterns, a dictionary of new bindings, and the object the pattern evaluated to.

Patterns can be any one of a number of pattern expressions: model element patterns, set patterns, variable bindings, and normal Converge expressions. Pattern expressions may arbitrarily nest other pattern expressions. Each pattern expression is translated to a generator which can generate zero or matches against given model elements. The translation is complicated by the fact that nested pattern expressions are also generators; therefore when a pattern is asked by backtracking to generate new matches, the backtracking may need to resume several levels deep in a nested pattern expression.

All translated pattern expressions contain a wrapper which iterates over the objects passed to the pattern and passes them one at a time to the pattern expression. Since the pattern expression is a generator, its result is immediately yielded to the translated patterns caller. Noting that `yield` in Converge is an expression, the outer translation is as follows:

```
func _t_pt_spattern(node):
  // pt_spattern ::= pt_spattern_expr
  return [|
    func (bindings, elements):
      for element := elements.iterate() & yield $<<self.preorder(node[1])>> \
        (bindings, element)

      return fail
  |]
```

In the following subsections I detail how each type of pattern expression is translated by MT.

5.6.5. Translating variable bindings

As befits the simplest type of pattern expression, MT's translation of variable bindings is simple:

```
1  func _t_pt_svar(node):
2    // pt_svar ::= "<" "ID" ">"
3
4    self._pattern_vars.add(node[2].value)
5    var_str := CEI.lift(node[2].value)
6    return [|
7      func (bindings, element):
8        if bindings.contains($<<var_str>>) & not bindings[$<<var_str>>] == \
```

```
9           element:
10          return fail
11        return [Set{}, Dict{$<<var_str>> : element}, element]
12    |]
```

Lines 8 – 10 check to see whether the variable in question has already been bound; if it has, the value of the existing binding is compared against the element being matched to ensure equality. If the original and new binding values are not equal, the variable binding fails. As such, this behaviour is largely redundant in MT since exactly the same effect can be achieved by having an initial variable binding followed by references to that variable. This behaviour is maintained for the sake of ensuring backwards compatibility with the QVT-Partners approach.

As the MT translation encounters variable bindings, it adds them to the set of known variable bindings in the translations' `_pattern_vars` field (line 4). This information is used in two different ways. Firstly the set of variable bindings is used to determine the valid variable references for subsequent patterns and clauses in a rule; this is necessary since variable bindings in patterns with multiplicities are dealt with differently (see sections 5.5.2 and 5.6.10). Secondly variables in a Converge expression which reference a variable binding need to be translated into a dictionary lookup on the current set of known bindings (see section 5.6.12).

5.6.6. Translating model element patterns

The translation of model patterns is the largest individual part of MT's translation, but can be split into two parts: matching the model element type and dealing with the self variable; matching against model element slots. The former part is relatively simple. The latter part is complicated by the need to deal with nested pattern expressions. In this subsection I first present a simplified translation of model element patterns which does not deal with nested pattern expressions, before presenting the complete translation.

A simplified translation

In order to understand the translation of model element patterns, I first consider a simplified variant. The model element patterns in this simple variant can not contain any reference to the self variable, and slot comparisons with nested pattern expressions will only be evaluated once. If a slot comparison fails, the entire pattern fails immediately. In the interests of brevity, only equality and inequality slot comparisons are translated. This subset still encompasses a number of interesting and useful model element patterns such as the following:

```
(Dog)[name == <n>, owner == (Person)[name != "Fred"]]
```

The simplified translation is as follows:

```
1  func _t_pt_smodel_pattern(node):
2    // pt_smodel_pattern ::= "(" pt_smodel_pattern_self ")" "[" pt_sobj_slot
3    //                       pt_smodel_pattern_comparison pt_spattern_expr { ","
4    //                       pt_sobj_slot pt_smodel_pattern_comparison
5    //                       pt_spattern_expr }* "]"
6    //                   ::= "(" pt_smodel_pattern_self ")" "[" "]"
7
8    // pt_sobj_pattern_self ::= "ID"
9    type_match := [|
10     if not TM.type_match($<<CEI.lift(node[2][1].value)>>, &element):
11       return fail
12   |]
13
14   slot_comparisons := []
15   while i < node.len() & node[i].conforms_to(List) & node[i][0] == \
16     "pt_sobj_slot":
17     slot_name := node[i]
18     slot_comparison := node[i + 1]
19     slot_pattern := node[i + 2]
20
21     if slot_comparison[1].type == "==":
22       slot_condition := CEI.ieq_comparison
23     elif slot_comparison[1].type == "!=":
24       slot_condition := CEI.ineq_comparison
25
26     slot_comparisons.append([|
27       slot_element := &element.$<<CEI.name(slot_name[1].value)>>
28       if not new_matched_mep_elems, new_bindings, matched_elem := \
29         $<<self.preorder(slot_pattern)>>(bindings, &slot_element):
30         return fail
31       if not $<<slot_condition([| &slot_element |], [| matched_elem |])>>:
32         return fail
33       &local_bindings += new_bindings
34     |])
35
36     i += 4
37
38   return [|
39     func (bindings, element):
40
41       local_bindings := Dict{}
42
43       $<<type_match>>
44
45       $<<slot_comparisons>>
46
47       return [Set{element}, local_bindings, element]
48   |]
```

Lines 9 to 12 deal with ensuring the model element to be matched is of the correct type. Lines 26 to 34 shown the heart of the translation of slot comparisons. Line 27 extracts the value of the model elements slot. Line 28 evaluates the slots pattern; if the slots pattern fails for any reason, the entire model pattern fails. The hitherto unused third element, hitherto referred to as ‘the object the pattern evaluated to’, returned by the pattern expression is then compared to the value of the model elements slot obtained in line 27, using the slot comparison operator. If the comparison operator fails, then the entire model pattern fails. As can be seen in line 48, a model element pattern ignores any model elements matched by nested model element patterns.

The clumsy term ‘the object the pattern evaluated to’ is used because conceptually there are two distinct ways in which a pattern expression evaluates. Converge expressions used as patterns

(e.g. `"Fred"`; see section 5.6.8) simply evaluate to an object which must be checked against the model elements slot. However other types of pattern expressions, such as the model element pattern `(Person)[name != "Fred"]`, are passed the value of the model elements slot and asked to match against it; they then return the value of the model elements slot unchanged. In other words, some types of pattern expressions (Converge expressions) evaluate to a new object whilst some return the object passed to them (e.g. model element patterns). Note that even in the latter case it is necessary to check the slot comparison after the evaluation, so that element patterns such as the following (which is functionally equivalent to the previous example) evaluate correctly:

```
(Dog)[name == <n>, owner != (Person)[name == "Fred"]]
```

Ensuring the complete evaluation of nested pattern expressions

The preceding translation of model element patterns contains one major flaw: it does not correctly deal with nested pattern expressions that may generate more than one match. Consider the following pattern:

```
(Dog)[name == <n>, allowable_foods == Set{<x> | <Y>}]
```

The set pattern `Set{<x> | <Y>}` will potentially generate a match for every element in a set matched against it. If, for example, the rule this pattern is part of contains a `src_when` clause along the lines of `x == "Biscuit"` then it is vital that the set pattern generates all possible matches to ensure that a correct match can be found (if one exists). In the previous translation of model elements, unless the nested set pattern happened to stumble across the correct combination during its first iteration, then the entire rule this is a part of would fail.

In the general case, pattern expressions may be nested to an arbitrary depth within one another. MT thus needs to ensure that all pattern expressions, no matter how deep they are nested, can generate all their possible matches. For model element patterns, it is also desirable that the pattern expressions in slot comparisons generate multiple matches in a predictable fashion. Pattern expressions are thus evaluated in a deliberately similar fashion to patterns in a `srcp` clause in the order that they were defined, from left to right. If a model element pattern is requested to generate more matches, the right most pattern expression will generate a further match if possible. When a pattern expression within a model element pattern generates all its possible matches, the pattern expression to its left generates a new match, which causes the control flow to return to its right, causing that pattern to generate a new match. When all pattern expressions within a model element pattern have generated all their matches, then the model element pattern itself fails. In common with patterns in a `srcp` clause, MT ensures that each time a pattern expression fails, the appropriate variable bindings are ‘undone’.

In order to cope with arbitrary levels of nested pattern expressions, one might reasonably expect a significant degree of complexity to be needed in the translation – indexes within lists needing to be passed around and stored, and so on. However by careful use of Converge generators and the conjunction operator, the desired effect can be achieved with a relatively small amount of code. Considering the same marginally simplified variant of model element patterns as previously (references to the self variable not allowed; only a subset of slot comparison operators dealt with), the translation is as follows:

```
func _t_pt_smodel_pattern(node):
  type_match := [|
    if not TM.type_match(\$<<CEI.lift(node[2][1].value)>>, &element):
      return fail
  |]

  returns_vars := []
  current_bindings_var := CEI.ivar(CEI.fresh_name())
  conjunction := [[| $<<current_bindings_var>> := &bindings |]]

  while i < node.len() & node[i].conforms_to(List) & node[i][0] == \
    "pt_sobj_slot":
    slot_name := node[i]
    slot_comparison := node[i + 1]
    slot_pattern := node[i + 2]

    if slot_comparison[1].type == "==":
      slot_condition := CEI.ieq_comparison
    elif slot_comparison[1].type == "!=":
      slot_condition := CEI.ineq_comparison

    next_bindings_var := CEI.ivar(CEI.fresh_name())
    return_var := CEI.ivar(CEI.fresh_name())
    returns_vars.append(return_var)
    conjunction.append([| $<<return_var>> := $<<func_>>( \
      $<<current_bindings_var>>) |])
    conjunction.append([| $<<next_bindings_var>> := \
      $<<current_bindings_var>> + $<<return_var>>[1] |])
    current_bindings_var := next_bindings_var
    i += 4

  conjunction.append([| [Set{&element}, Functional.foldl(_adder, \
    Functional.map(_element1, $<<CEI.ilist(returns_vars)>>)), &element] |])

  return [|
    func (bindings, element):

      $<<type_match>>

      for yield $<<CEI.iconjunction(conjunction)>>

      return fail
  |]

func _adder(x, y):
  return x + y

func _element0(x):
  return x[0]

func _element1(x):
  return x[1]
```

Note that in this code `_adder`, `_element0` and `_element1` are module level functions. These functions are used in later translations in this chapter.

The underlying theme in this translation is that pattern expressions in slot comparisons may be generators. Pattern expressions are placed into a single conjunction expression (chiefly built up in lines 25 – 28). The translation places the conjunction containing translated pattern expressions within a `for` construct (line 40), which yields a value each time a successful match of all slot comparisons is found. Thus the translation utilizes Converge's built-in goal-directed evaluation (section 4.1.3) to ensure that all possible values – including those from nested pattern expressions – for all translated slot comparisons are evaluated.

There are two further (somewhat related) aspects of the translation which require explanation. The first of these relates to MT's treatment of variable bindings, particularly the need to 'undo' variable bindings when a slot comparison fails and Converge backtracks. After each slot comparison has been added to the conjunction, MT creates a new uniquely named variable (line 22) which has assigned to it the union of the existing variable bindings and those created by the pattern expression (lines 27 and 28). The pattern expression in the next slot comparison then uses this union of variable bindings as its set of currently valid bindings (line 26). When Converge backtracks, the currently valid bindings are implicitly undone since the union of existing and new bindings is performed after each translated slot comparisons.

The second aspect relates to the value returned by a model element pattern, which is created in lines 32 to 33. Since model element patterns ignore elements matched by nested model element patterns (section 5.4.2), it is not surprising that the first element of the returned list is a set containing only the element matched by the current model element pattern. The second element of the returned list which is initially rather foreboding using as it does the `foldl` and `map` functions which operate as their LISP counterparts. Before tackling it directly, we first need to investigate the `return_vars` variable in the translation, which is a list containing quasi-quoted variables. Each time a nested pattern expression is evaluated, a new `return_var` variable is created (line 23) to which the return value of the pattern expression is assigned (lines 25 and 26). The `return_var` variable is then added to the `return_vars` list (line 24). Each `return_var` variable thus holds a standard three element list. The `foldl` call in line 32 is then passed a list of lists at run-time; each sub-list will be a list containing three elements, as returned by a pattern expression. The `_element1` function then selects the variable bindings generated from each slot comparison; the `_adder` function then creates a union of these variable bindings. This union is a non-strict subset of the final value of, `current_bindings_var` which will include all the bindings passed to the model element pattern in its `bindings` argument. Note that whilst it may initially appear simpler to make `return_vars`

a run-time variable to which each pattern expressions return list is appended, this would then lead to complications when back-tracking would require items in the list to be removed. However since the variables in `return_vars` are known at compile-time, it would be possible to achieve a small optimization by moving the `foldl` call to compile-time; this is left as an exercise for the reader.

It is interesting to compare this translation to that used for patterns in a `srcp` clause, as presented in section 5.6.3. While the two transformations are essentially functionally equivalent, the earlier translation is perhaps more initially appealing since it reuses a familiar concept (nested `for` constructs). Although the translation in this section uses the less familiar conjunction operator, it results in a much shorter, more idiomatic – and marginally more efficient – translation. As this may suggest, making use of some of the less common features present in Converge can be of significant advantage when translating DSLs.

5.6.7. Translating set patterns

In this subsection I outline the translation of set patterns, but do not delve into the code of the translation which uses the same techniques and idioms outlined in the translation of model element patterns.

Set patterns match against single element patterns (those to the left of the '`|`' character) and subset patterns (those to the right of the '`|`' character) simultaneously. For each single element pattern, MT iterates over the set being matched; no two single element patterns will be matched against the same element simultaneously. For each subset pattern, MT iterates over the powerset of the set to match against. The intersection of all subsets (including the set comprised of all single element patterns matched) must be $\emptyset$. The union of all subsets (including the set comprised of all single element patterns matched) must equal the set being matched. Set patterns then generate an appropriate return value whenever each single element pattern and each subset pattern match successfully.

5.6.8. Translating Converge expressions when used as patterns

As outlined in section 5.6.6, when Converge expressions are used as pattern expressions, they act in a different fashion to other pattern expressions. Whereas all other types of pattern expressions are a declarative match against model elements, Converge expressions are simply expected to evaluate to constants (in this situation meaning integers, strings, model elements and so on). MT therefore defines that Converge expressions in this situation are only evaluated once – even if the particular Converge expression is a generator, it will only ever be required to generate a single value. Converge expressions used as pattern expressions return a list consisting of the empty set to represent the model elements matched by model element patterns, an empty dictionary of bindings, and the constant object the expression evaluated to.

The translation of Converge expressions in this instance thus requires only a very thin wrapping around the actual expression itself:

```
func _t_pt_spattern_expr(node):
  // pt_spattern_expr ::= pt_sobj_pattern
  //                  ::= pt_sset_pattern
  //                  ::= pt_svar
  //                  ::= expr

  if node[1][0] == "expr":
    return [|
      func (bindings, elements):
        return [Set{}, Dict{}, $<<self.preorder(node[1])>>]
    |]
  else:
    return self.preorder(node[1])
```

Section 5.6.12 details how the Converge grammar rule `expr` is embedded in the MT grammar.

5.6.9. An example translated pattern

Having now seen the translations of variable bindings, model element patterns, set patterns and Converge expressions used as pattern expressions, we are now in a position to see the result of translating a particular pattern. I use the following pattern, which incorporates all three types of pattern expressions:

```
(Dog)[name == <n>, allowable_foods == Set{"pork" | <Y>}]
```

The result of translating this pattern is the following ITree:

```
unbound_func (bindings, element){
  if not Input_Pattern_Creator.TM.type_match("Dog", element):
    return Input_Pattern_Creator.fail
  for yield $$76$$ := bindings & $$78$$ := unbound_func (bindings){
    slot_element := element.name
    for matched_mp_elems, new_bindings, matched_elem := unbound_func \
      (bindings, element){
      if bindings.contains("n") & not bindings["n"] == element:
        return Input_Pattern_Creator.fail
      return [Set{}, Dict{"n" : element}, element]
    }(bindings, slot_element):
      if slot_element == matched_elem:
        yield [matched_mp_elems, new_bindings, matched_elem]
    return Input_Pattern_Creator.fail
  }($$76$$) & $$77$$ := $$76$$ + $$78$$[1] & $$84$$ := unbound_func (bindings){
    slot_element := element.allowable_foods
    for matched_mp_elems, new_bindings, matched_elem := unbound_func \
      (bindings, element){
      if not element.conforms_to(Input_Pattern_Creator.Set):
        return Input_Pattern_Creator.fail
      if element.len() < 1:
        return Input_Pattern_Creator.fail
      for $$79$$ := element.iterate() & $$80$$ := unbound_func (bindings, \
        elements){
        return [Set{}, Dict{}, "pork"]
      }(bindings, $$79$$) & $$81$$ := Input_Pattern_Creator.Functional. \
        powerset_generator(element) & $$81$$.union(Set{$$79$$}) == element & \
        not $$81$$.contains($$79$$) & $$82$$ := unbound_func (bindings, \
        element){
        if bindings.contains("Y") & not bindings["Y"] == element:
          return Input_Pattern_Creator.fail
        return [Set{}, Dict{"Y" : element}, element]
      }(bindings, $$81$$) & $$82$$[2] == $$81$$:
```

```
34          yield [$$80$$[0] + $$82$$[0], $$80$$[1] + $$82$$[1], element]
35        return Input_Pattern_Creator.fail
36      }(bindings, slot_element):
37        if slot_element == matched_elem:
38          yield [matched_mp_elems, new_bindings, matched_elem]
39      return Input_Pattern_Creator.fail
40    }($$77$$) & $$83$$ := $$77$$ + $$84$$[1] & [Set{element}, \
41      Input_Pattern_Creator.Functional.foldl(Input_Pattern_Creator._adder, \
42      Input_Pattern_Creator.Functional.map(Input_Pattern_Creator._element1, \
43      [$$78$$, $$84$$])), element]
44    return Input_Pattern_Creator.fail
45  }(bindings, element)
```

Despite its initial appearance as an impenetrable jumble of bizarrely named identifiers, through careful examination of the input pattern, and the translations presented in this section, it is possible to identify which parts of this ITree relate to specific parts of the input pattern. The first step in this is to recursively break the input pattern down into its constituent pattern expressions. One can then determine which line numbers each pattern expression relates to. A simple table showing this is as follows (note that due to the recursive breakdown, outer pattern expressions line numbers overlap with those of nested pattern expressions):

Pattern	Lines
`(Dog)[name == <n>, allowable_foods == Set{"pork" | <Y>}]`	1 – 45
`<n>`	6 – 13
`Set{"pork" | <Y>}`	20 – 34
`"pork"`	23 – 26
`<Y>`	28 – 33

5.6.10. Translating pattern multiplicities

In order to deal with patterns with multiplicities (see section 5.5.2), some additions need to be made to the outer translation of patterns from section 5.6.4. Although each of the several forms of pattern multiplicities requires a specific translation, they all follow the same general form, which can be split into two distinct phases. In order to demonstrate this, I present the translation for the `*` multiplicity in an elided view of the `_t_pt_spattern` function:

```
1  func _t_pt_spattern(node):
2    // pt_spattern ::= pt_spattern_expr pt_spattern_qualifier
3    // pt_spattern_qualifier ::= ":" pt_multiplicity "<" "ID" ">"
4    // pt_multiplicity ::= pt_multiplicity_upper_bound
5    // pt_multiplicity_upper_bound ::= "*"
6
7    self._inside_multiplicity_pattern += 1
8    pattern := [|
9      func (bindings, elements):
10       matches := []
11       for element := elements.iterate() & matches.append($<<self.preorder( \
12         node[1])>>(bindings, element))
13
14       powerset := Functional.powerset(matches)
```

```
15        powerset := Sort.sort(powerset, func (x, y) {
16          return x.len() < y.len()
17        })
18        for matches := powerset.riterate():
19          if matches.len() == 0:
20            continue
21          yield [Functional.foldl(_adder, Functional.map(_element0, matches)), \
22            Dict{$<<CEI.lift(node[2][4].value)>> : Functional.map(_element1, \
23            matches)}, Functional.foldl(adder, Functional.map(func (x) {
24            return [x[2]]
25          }, matches))]
26
27        return [Set{}, Dict{$<<CEI.lift(node[2][4].value)>> : []}, elements]
28    |]
29    self._inside_multiplicity_pattern -= 1
30
31    return pattern
```

The first phase of a multiplicities execution involves matching elements. In the case of the `*` multiplicity, this occurs in lines 10 to 12 which evaluates every successful match of the pattern (note that when the pattern does not match successfully, backtracking ensures that the `append` call will not be executed). The second phase of execution then successively returns permutations of the matches. Note that, although not the case for the `*` multiplicity, in some multiplicities these two phases will be partially intertwined. Lines 14 to 17 evaluate the powerset[5] of matches, sorting the resulting permutations into ascending order based on the number of elements in each. The `for` construct in line 18 then iterates over the the powerset in reverse order, yielding permutations of lesser size as the multiplicity is called upon to generate new matches. The list yielded by the multiplicity in lines 21 to 25 is simpler than it may first appear. Line 21 unions the elements matched by model element patterns from each match in the permutation. Lines 22 and 23 create a single binding for the multiplicities variable, assigning it a list of variable bindings, with a bindings entry for each match in the permutation. Lines 23 to 25 union the objects each match in the permutation evaluated to.

The translation of pattern multiplicities requires a small but important change to the translation of variable bindings, to prevent variable bindings within multiplicities from being added to the `_pattern_vars` field. The `_inside_multiplicity_pattern` field within the translation tracks whether the translation is currently processing a pattern multiplicity or not. The updated `_t_pt_svar` function thus looks as follows:

```
func _t_pt_svar(node):
  // pt_svar ::= "<" "ID" ">"

  if _inside_multiplicity_pattern == 0:
    self._pattern_vars.add(node[2].value)
  var_str := CEI.lift(node[2].value)
  return [|
    func (bindings, element):
      if bindings.contains($<<var_str>>) & not bindings[$<<var_str>>] == \
        element:
        return fail
      return [Set{}, Dict{$<<var_str>> : element}, element]
  |]
```

[5] The Converge `powerset` function returns a list of lists if, as in this case, it is passed a list rather than a set.

5.6.11. Standard functions

Each transformation has two standard functions of particular importance: the `transform` and `transform_all` functions. In this subsection I show the definition of these two functions.

The `transform` function was outlined in section 5.3.1. It takes a variable number of arguments, each of which must be a list. Noting that function objects in Converge have a function `apply` which takes a list of values and applies them to the function as if they were passed as individual arguments, the `transform` function looks as follows:

```
1  func transform(*elems):
2    for elem := elems.iterate():
3      if not elem.conforms_to(List):
4        raise Exceptions.Type_Exception(List, elem.instance_of, elem.to_str())
5
6    for rule_name := self._rule_names.iterate():
7      if target := self.get_slot(rule_name).apply(elems):
8        return target
9
10   raise Exceptions.Exception(Strings.format("Unable to transform '%s'.", \
11     elems.to_str()))
```

The first action of the `transform` function After is to type-check its arguments in lines 2 to 4. It then makes use of the `_rule_names` field within a transformation which records the names of a transformations rule in the order they were defined. Iterating over the `_rule_names` field allows the translated rule function to be accessed via the `get_slot` function. Using this, the `transform` function calls rules in the order in which they were defined, succeeding as soon as it finds a rule which executes on the input. If no rules execute, an exception is raised.

The `transform_all` function is a simple, but highly useful, convenience function built on top of the `transform` function. Given a list of model elements, it transforms each using the `transform` function. The definition of `transform_all` is thus simple:

```
func transform_all(elems):
  if not elems.conforms_to(List):
    raise Exceptions.Type_Exception(List, elems.instance_of, elems.to_str())

  target_elems := []
  for elem := elems.iterate():
    target_elems.append(self.transform([elem]))

  return target_elems
```

5.6.12. Embedding Converge code within DSLs

When compared to other model transformation approaches, one of MT's most novel aspects is its ability to embed GPL code within it. This is possible due to Converge's DSL embedding features. The ability to embed Converge code in DSLs benefits both the DSLs users and implementers. Users can reuse their knowledge of standard Converge, whilst DSL implementers can reuse tried and trusted parts of the Converge compiler. In this subsection I explain how a DSLs can embed Converge within

itself.

The key to embedding normal Converge code can be seen in such as `pt_ipattern_expr` in the MT grammar (section B.1) which reference the `expr` rule from the Converge grammar (chapter A). The first point to note is that all rules in the MT grammar are prefixed by `mt_`; this allows the MT grammar to be merged with the Converge grammar with no conflicts. Since CPK grammars are currently defined in strings, merging two grammars together is simply a case of adding two strings. In a similar fashion to TXL (section 3.3.6) there is currently no notion of grammar namespaces nor are any checks for conflicts between the two grammars. As this may suggest, whilst the current implementation of this feature is workable, it is one of the less refined parts of DSL implementation in Converge.

MT extends `IModule_Generator` in much the same way as the `switch` DSL (see section 4.4.2). The MT subclass has only one non-trivial interaction with its superclass, needing to override the `_t_var` function. References to variable bindings are translated into dictionary lookups on the `bindings` variable (see section 5.6.5); see also section 5.6.14. An elided version of the `_t_var` translation function is as follows:

```
func _t_var(node):
  // var ::= "ID"

  if self._pattern_vars.contains(node[1].value):
    return [| &bindings[$<<CEI.lift(node[1].value)>>] |]
  else:
    return exbi IModule_Generator._IModule_Generator._t_var(node)
```

In summary – despite the need to add strings representing grammars together, and to subclass a complex class residing in the depths of the Converge compiler – the process of embedding Converge code in DSLs is surprisingly simple and relatively free of complications. However it is unclear whether this approach would scale satisfactorily to larger examples. I believe that in the future two things may need to be changed to improve the situation. Firstly grammars need to be properly modularised to ensure that naming problems between grammars do not arise, and that the relationship between grammars is clearly stated. Secondly it would be useful to loosen the coupling between DSLs and the `IModule_Generator` module, possibly by removing the requirement to subclass the `IModule_Generator` class.

5.6.13. Extending the Converge grammar

Although this section has thus far ignored the translation of a rules target clauses, the presence of model element expressions in such clauses is worthy of examination. As a brief recap, model element expressions such as `(Dog)[name := "Fido", owner := (Person)[name = "Fred" ]]` create new model elements; they are syntactically similar, although not identical, to model ele-

ment patterns. Section 5.3.4 contains more details on model element expressions. Model element expressions can be used anywhere in a rules target clauses that a normal Converge expression can be used. Although this may suggest that model element expressions can only be used at the top-level within target clauses, they can in fact be used within Converge expressions themselves. For example the following expression shows how a model element expression can be used within a Converge list:

```
[(Person)[name = "Fred"]]
```

As this example shows, in the context of target clauses, model element expressions effectively embed themselves in the the base Converge language itself.

The embedding of model element expressions is currently implemented by taking advantage of the fact that that CPK grammars are strings, and that production rules can have alternatives added at any point in the grammar. Thus in the MT grammar (section B.1) the `expr` rule from the Converge grammar is extended with a new alternative by MT pointing to the `pt_mep_pattern` rule. Since the `expr` translation function in the Converge compiler immediately hands computation over to the rule named in its alternatives, the MT translation class needs only to provide a simple translation function for `pt_mep_pattern`.

It should be noted that whilst extremely powerful, this technique is not generally applicable. It currently requires detailed knowledge of the Converge grammar and the Converge compilers internals in order to ensure that extending a rule in the Converge grammar has the desired effect. I hope that future versions of Converge will be able to provide safer support for extension of this sort.

5.6.14. Unintended interactions between translated and embedded code

One of the challenges not tackled in the TM DSL was preventing unintended variable capture from DSL input and the translated DSL code. This problem arises when an ITree derived from user input is placed inside an ITree containing dynamically scoped variables. As seen in the translations in this section, dynamically scoped variables occur frequently, chiefly via the `&var` syntax. Dynamically scoped variables are highly useful in allowing ITrees to be built piece meal. However whilst statically scoped variables are automatically safely renamed by Converge's scoping rules (section 4.2.5), dynamically scoped variables may cause variable capture with ITree's derived from user input. For example, the variable `bindings` is frequently dynamically scoped in the translations of this section; if an MT were to use the same variable name in, for example, a `tgt_where` clause, then unexpected results would almost certainly arise.

To prevent this problem occurring, MT performs its own α-renaming of variables in Converge expressions. MT takes advantage of the fact that each ITree can report its free and bound variables (via the `get_free_vars` and `get_bound_vars` functions respectively). For each rule, MT first

calculates the free and bound variables of all Converge expressions contained in that rules clauses. For each variable, a fresh name is then generated; a dictionary records the mapping between the original and fresh names. When MT encounters Converge variables during its translation, it translates them to variables with the corresponding fresh name. By renaming all variables from users input, MT thus ensures that there can be no unintended variable capture.

Once all variables have been safely renamed, a rules free variables then require extra treatment. For example the `concat_name` function in section 5.3.5 is a free variable in the context of the `Classes_To_Tables` transformation, since it is defined outside of the transformation. All instances of the `concat_name` within a given rule will be renamed to a variable along the lines of `$$5$concat_name$$`. At this point, there is no link between the value of `concat_name` outside the rule, and the value of `$$5$concat_name$$`. Thus MT adds to translated rules assignments from the original value of variables to their fresh name equivalent. In the case of the `concat_name` function, the result of the translation would look along the lines of the following:

```
$$5$concat_name$$ := concat_name

...
  $$5$concat_name$$("", bindings["n"])
...
```

Assigning to variables in outer blocks

Although the α-renaming mechanism of variables in user input prevents unintended variable capture, it introduces problems due to the disconnect between the original variable and its fresh-named clone. This leads to two related problems.

The first problem relates to assigning to free variables. Since a fresh-named clone is made of each free variable, assigning to free variables in an MT block does not affect the value of the original variable. MT thus mirrors the normal Converge expectation that variables assigned to in a block (where a block in MT is essentially a rule) are local to that block. However a problem arises if one wishes an assignment to a free variable. First, let us assume that MT allows some free variables to be declared as nonlocal (recall that in normal Converge `nonlocal` x is a declaration that assignment to the variable x does not create a local x, but instead binds to the the first outer block which contains an assignment of x). Assignment to a free variable then becomes problematic, since the user will be assigning to the variables fresh-named clone; furthermore there is no way to assign to the original variable without reintroducing the prospect of variable capture. A partial solution to this problem is for MT to mirror the assignment of free variables to their fresh-named clone at the beginning of the translated rule, with the assignment of the fresh-named clone to its free variable equivalent at the end of the transformation. Whilst this is possible, it means that during the execution of the rule the local

and global values of the variable may differ.

This then highlights the more general problem, which is that at any point during the execution of a rule the values of the original variable and its fresh-named clone may diverge either through assignment to the original variable or its fresh-named clone. There is no solution for this problem at the moment in Converge. Although one can devise increasingly sophisticated work arounds which reduce the potential for the problem to arise, fundamentally the cloning of variables is flawed since there is no mechanism for atomically synchronising the cloned and original variables.

The situations in which this deficiency are exposed are essentially confined to compile-time meta-programming, although one would not normally expect to encounter the problem in practise. However Converge DSLs such as MT, which embed Converge code within the DSL, greatly increases the chances of hitting this problem due to potential clashes in variable names between embedded Converge code and the DSL. A possible solution to this deficiency would be for Converge to acquire a 'variable alias' feature which would alias a variable `x` in an outer block to `y` in an inner block. Since the names would merely be aliases for the same underlying variable, there could then be no synchronization issues between the two. Such a feature would ideally work in much the same way as the `nonlocal` declaration; indeed, it is also implicit that aliased variables are nonlocal to the block in which they are renamed. Although the Converge VM provides sufficient support for such a feature, the compiler and language have yet to be sufficiently extended.

5.6.15. Generating tracing information from nested model patterns

In section 5.4.2, the standard MT tracing information creation mechanism was outlined. By default, only non-nested model element patterns contribute to the source part of trace tuples. I asserted that empirically this appeared to be a sensible compromise that created sufficient tracing information without overwhelming the user. However it is clear that this technique may not be suitable for all applications; one can easily imagine further research to determine the most practical tracing information creation techniques for different types of transformations. To this end, in this subsection I present a simple modification to the MT translation which changes the default tracing information created by allowing nested model element patterns to contribute to the source part of trace tuples. This serves two separate purposes. Firstly it provides evidence that the default tracing information creation mechanism achieves a useful balance in terms of the volume of information it creates. Second it shows that DSL implementations in Converge tend to be amenable to changes, and also that the MT implementation itself can serve as a testbed for further model transformation experimentation.

The modification to MT necessary to allow nested model expression patterns to contribute to the source part of trace tuples is in fact rather simple. All that is needed is for model element patterns

to return the union of all elements matched by nested model element patterns. By default, model element patterns only return the element they matched against, ignoring the elements matched by nested model element patterns. However the required information is present in the `return_var` associated with each slot comparison in a model element pattern. Thus all that is needed is to use the same technique used to union the bindings of each slot comparison. Replacing lines 31 to 32 of the complete translation from section 5.6.6 with the following achieves the desired effect:

```
conjunction.append([| [Set{&element} + Functional.foldl(_adder, \
  Functional.map(_element0, $<<CEI.ilist(returns_vars)>>)), \
  Functional.foldl(_adder, Functional.map(_element1, $<<CEI.ilist( \
    returns_vars)>>)), &element] |])
```

Taking exactly the same source model and transformation used in figure 5.13, the result of making this change to MT can be seen in figure 5.14. Note that in this new visualization, one can see that many target elements have tracing information from more than one source element; the end result is rather harder to read than figure 5.13, and does not add significantly to the users understanding of the transformation in this particular case.

5.6.16. Summary of the implementation

In this section I have presented an analysis of the major parts of the MT implementation. To demonstrate the result of MT's translation of a transformation, section D.2 shows the complete result of translating the simple MT transformation from section 5.3.5.

5.7. Related work

Chapter 3 gave an overview of many of the leading model transformation systems currently available. As with the majority of existing systems, MT is a unidirectional stateless model transformation system. MT's most obvious ancestor is the QVT-Partners approach [QVT03b] which pioneered the use of patterns in model transformations. MT takes the base QVT-Partners pattern language and enriches it with features such as pattern multiplicities, and variable slot comparisons. Furthermore, by providing a concrete implementation – and a detailed explanation of that implementation – much of the vagueness associated with other model transformations such as the QVT-Partners approach is avoided in MT.

A significant difference from the QVT-Partners approach is in MT's imperative aspects. Due to its implementation as a Converge DSL, MT can embed normal Converge code within it. This contrasts sharply with the QVT-Partners approach which is forced to define an OCL variant with imperative features in order to have a usable language. As explained in section 5.2.4, this variant language suffers from several conceptual and practical problems. I believe that MT is unique in being able

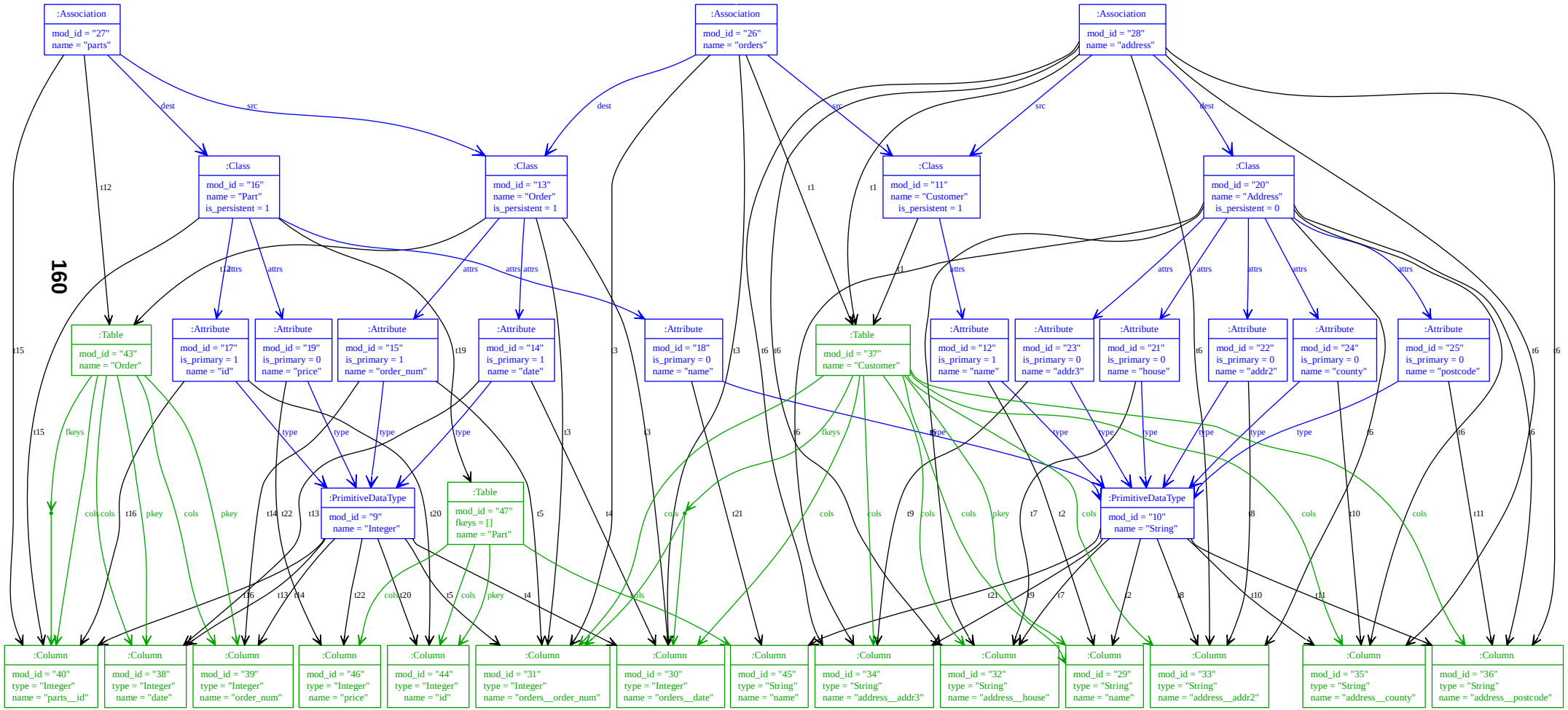


Figure 5.14.: Tracing information from nested model pattern expressions.

to embed a GPL within it. Perhaps more significant than the actual language embedded within MT transformations is the ability to call out naturally to normal Converge code, even if it is defined outside of the transformation. MT users are thus not constrained by any limitations of the particular model transformation approach. Although this may initially appear to be a mere implementation detail, it differentiates MT from virtually all existing model transformation approaches, which typically present a highly constrained execution environment.

Perhaps the closest model transformation approach is the commercial XMap language [CESW04], an approach essentially based on the QVT-Partners approach. This also means that the issues noted in both this section, and in section 5.2.4 with respect to the QVT-Partners approach, apply equally to XMap. XMap is however notable for its sister language XSync which allows changes to be propagated in the style outlined by Tratt and Clark [TC03]. Chapter 6 shows how MT can be evolved into a powerful change propagating language.

Perhaps surprisingly, given the seeming simplicity of the task, one of MT's most distinctive features is its automatic creation of tracing information. Most approaches neglect this problem; the few that tackle it, such as the DSTC approach [DIC03], require the user to manually specify the tracing information to be created. By using patterns defined by the user to automatically derive tracing information has not, to the best of my knowledge, been used by any other system. MT distinguishes itself further by its simple, but effective, technique for reducing superfluous tracing information.

It is perhaps telling that although MT contains several enhancements compared to existing approaches, it also shares many of the limitations of existing approaches, such as a lack of rule structuring mechanisms. Section 5.8 outlines the work that may resolve some of these limitations.

5.8. Future work

Although I believe that MT is currently one of the most advanced model transformation languages available, the relative immaturity of the area means that no new approach can claim to present a definitive solution.

Perhaps the most pressing question for every model transformation approach, including MT, is with regards to scalability. Although MT has been used to express transformations of the order of magnitude of the low tens of rules, it is clear that in order to make larger transformations feasible, new techniques for structuring and combining rules will be required. For example, currently all the rules in a MT transformation exist in a single namespace; there is no notion of 'transformation modules'. Similarly at the moment all rules exist at the same level; that is, given an element to transform, rules are tried in order. Complex transformations will require more selective mechanisms

to determine which rules can be executed, both for structuring efficiency reasons. At the moment, a transformations execution time has a worst case proportional to n^2 where n is the number of rules in the transformation; authors of larger transformations may require that their domain knowledge is used to narrow down the number of rules used to transform given elements. I believe that analysing work on combinators in functional languages may lead to new insights on how to better structure transformations[6].

The desirable for scalability is a concrete manifestation of a more nebulous problem surrounding model transformations: their usability. Whilst one can present advanced tools to users, it is vital that the tools be relatively easy to use. I believe there is significant work to be done in presenting model transformation languages to different users. MT could serve a useful purpose here in allowing model transformation languages to be easily tailored for different audiences.

In terms of 'nitty gritty' details, there are several aspects of MT that could usefully be improved. For example, one irritation encountered in this chapter relates to the `for` suffix of expressions in a rules `tgtp` clause. Currently rules can generally only produce as many top-level elements as they have expressions in the `tgtp` clause. This can occasionally lead to cumbersome or dangerous work arounds being employed. It would be useful to have a variant `for` suffix which would 'fold in' the elements produced by its expression as if they had been produced by top-level `tgtp`. As befits a new, small language similar examples can easily be found elsewhere in MT.

5.9. Summary

In this chapter I presented the MT model transformation language. I started by examining the QVT-Partners approach, from which MT is partly derived, in depth. Identifying the strengths and weaknesses of this approach explains some of the underlying design decisions taken with MT. I then explored MT's basic features, including its novel visualization abilities of transformations, including automatically generated tracing information. I then explored some of MT's more advanced features such as pattern multiplicities, which allowed a sophisticated model transformation to be concisely expressed. I then finished the chapter by examining in depth the translation of an MT transformation into MT.

I believe that MT is the first model transformation approach to present a detailed analysis of its implementation. In so doing, I hope that MT serves two purposes: a demonstration of implementing a non-trivial DSL in Converge; a demonstration of practical idioms for implementing model transformation engines. As the source code for MT is freely available, I hope that it will allow others to take

[6] This suggestion is partly a result of a conversation with Bernhard Rumpe, made during a visit to the Technische Universität Braunschweig in February 2005.

MT and alter it for their own purposes. In this way, I hope that MT aids further experimentation into differing model transformation techniques.

MT's implementation is also notable for its relative brevity. Through careful use of Converge idioms such as generators and the use of goal-directed evaluation, I assert that much of the tedious machinery that would be needed if MT were to be implemented in a standard GPL has been avoided. Although it is outside of the scope of this thesis to present hard numbers to back up this claim, I believe that MT provides compelling evidence that the seemingly disparate influences on Converge (such as Icon's goal-directed evaluation, ObjVLisp's data model, and Template Haskell's compile-time meta-programming) coalesce to form a natural and highly powerful development environment.

In the following chapter MT is used as the basis for a change propagating transformation language.

Chapter 6.

A change propagating model transformation system

This chapter builds upon the MT language defined in the previous chapter, creating a new unidirectional change propagating model transformation language PMT. Motivation for change propagating transformations was given in section 2.2.3. Alanen and Porres provide a useful overview of change propagating transformations, which also explains some of the categories of changes that can be propagated [AP04]. Change propagating transformations introduce considerable complexity compared to stateless transformations. It is my belief that no one approach to change propagation is likely to prove sufficient for all purposes. Furthermore due to the lack of focus on this particular area of model transformations, much exploration will be necessary to determine when different approaches are most applicable. The aim of this chapter is to outline some of the possibilities for change propagating approaches, and to present a particular unidirectional change propagating solution, PMT. PMT is intended to provide support for use cases similar to that outlined in section 2.2.3.

As noted in chapter 3, although several model transformation approaches mention change propagating transformations few actually provide such a mechanism. For the purposes of this thesis, only three approaches are potentially of interest: BOTL [BM03], Johann and Egyed's approach [JE04], and XMOF [CS03]. Both BOTL and XMOF are of limited interest, due to their differing aims compared to PMT. Since BOTL restricts itself to bijective transformations, I discount it, since I believe that bijective transformations constitute only a small proportion of useful transformations (see section 3.3.4). XMOF is also of limited interest since it is poorly documented, and aims to provide a solution for bidirectional change propagating model transformations, which introduces an extra set of challenges above and beyond those presented by unidirectional change propagating model transformations. Johann and Egyed's approach is the most interesting of the three, as it tackles unidirectional change propagating model transformations; however it explains only one aspect of its approach in detail, and furthermore is incapable of propagating some important types of changes.

It is an explicit aim of PMT to facilitate change propagation in any type of model transformation. However it is important to note that PMT is not as mature or stable as MT – by its nature PMT is much more of an experiment than MT. Nevertheless I hope that this chapter serves as a useful step on the path towards mature change propagating model transformation solutions.

This chapter begins with an overview of some of the high-level strategies and design decisions relevant to change propagation. PMT itself is then introduced, and via example it is shown how it allows change propagating transformations to be expressed. I show how PMT relates source and target models, and how it is capable of propagating changes that defeat other approaches. I also detail PMT's support for expressing change propagating transformation specifications. Finally I detail some of the relevant parts of PMT's implementation; since PMT is able to reuse much of MT's implementation, this chapter places less emphasis on the implementation than in the previous chapter.

6.1. Change propagation

Whilst section 2.2.3 motivated change propagating model transformations, it gave very little hint as to how such transformations might be realised. The intention of this section is to outline the background of change propagations, and some of the overall design decisions possible when implementing a change propagating model transformation approach. Note that I only consider these design decisions in the context of unidirectional change propagating transformations.

6.1.1. Change propagation compared to incremental transformation

Incremental transformation (sometimes known as incremental computation) is a well studied field (see [RR93] for an overview of some of the available literature). The most widespread, and one of the simplest, examples of incremental transformation are code compilation systems. For example the UNIX `make` command takes a list of source code files, and compiles only those which have been modified since the last execution of `make`.

Incremental transformation initially appears to be very similar to change propagation. Both approaches provide support for taking a source item and transforming it into an appropriate target item; subsequent changes made to the source item then cause appropriate updates on the target item. However incremental transformation approaches assume that the target item will be unmodified by the user when they update it. Incremental transformation need not therefore concern itself with many of the issues that affect change propagation in the context of this thesis, chiefly how to propagate changes non-destructively into the target model. This can be seen clearly in the code compilation system example; any modifications the user may make to the output of the compiler will be lost the

next time the code compilation system discovers it needs to recompile the associated source file.

There is thus a fundamental difference between the two approaches, since an incremental transformation approach is able to make assumptions about its environment that conflict with the use case outlined in section 2.2.3. For the purposes of this thesis, change propagation is therefore largely treated as a new subject with respect to incremental transformation systems.

6.1.2. Manual or automatic change propagation

Tratt and Clark outline a framework intended to allow unidirectional stateless transformations to be associated with one or more *delta transformations* which can propagate changes [TC03]. The execution sequence of such transformations is as follows. The unidirectional stateless transformation takes in a source model and produces a target model as normal. Subsequent changes made to the source model are extracted as change deltas to the source model. These deltas are then passed to an appropriate delta transformation which is expected to propagate the change represented by the delta to the target model. In general each different type of change will require a different delta transformation to be created. Note that the framework itself does not impose, or facilitate, a particular change propagation mechanism is left open in this framework. An example of this framework can be seen in the XMF tool which includes a change propagation framework with a dedicated delta transformation language XSync, to accompany a unidirectional stateless model transformation language XMap [CESW04].

The concept of delta transformations is an interesting one in that it provides a means of integrating legacy, or otherwise incapable, transformations into a change propagating transformation. However it has two inherent problems. Firstly there is an inevitable disconnect between the core unidirectional stateless transformation, and the delta transformations, all of which must be created by hand. Secondly there is, in general, no bound on the number of delta transformations needed to cope with change deltas. For this reason I classify this framework as manual change propagation, since the code to perform change propagation must be manually created.

Manual change propagation contrasts with automatic change propagation, where a transformation can propagate changes without additional code needing to be added. Some approaches choose a hybrid approach, being able to automatically propagate some changes whilst requiring manual assistance to propagate others. For example, OptimalJ is able to propagate changes between some of its simple models automatically, but can require assistance when propagating changes between a complex model and its textual representation [OJ04].

6.1.3. Propagating changes in batch or immediate mode

There are two potential modes of operation for running change propagating transformations: 'batch' and 'immediate' mode. These two modes refer to the number of changes that are propagated in each step.

Batch change propagation takes a number of changes from the source model and propagates them to the target model only when explicitly requested to do so by the user. The advantage of batch change propagation is that the user is in complete control of when changes are propagated. Batch change propagation can be considered to be similar to code compilation — users typically make multiple edits to a source code file before choosing to compile it. Since change propagation may be a relatively slow activity, it is beneficial to the user if they can schedule change propagation at a time convenient to them. On the other hand, the user may consider it inconvenient to have to manually force changes to be propagated.

The concept of immediate change propagating transformations is defined in [CS03]. An immediate change propagating transformation propagates changes to the target model as soon as the source model is changed. Unlike a batch mode change propagating transformation, which implicitly propagates multiple changes when run, an immediate mode change propagating transformation propagates small changes, which can be viewed as being semi-atomic. The advantage of immediate mode propagation is that the source and target models involved in the transformation are always synchronised with each other. However there are a number of potential disadvantages to immediate change propagating transformations.

From the users point of view, immediate change propagation may introduce a lag every time the user makes a change to the source model, whilst the system propagates the appropriate changes to the target model. During this lag, the system can choose to either lock the source model, thus preventing the user making changes to it, or to place changes to the source model into an ordered queue. In the former case, the user is likely to become highly frustrated; in the latter case, the advantage of synchronised source and target models is lost, albeit temporarily. Furthermore, the process of changing a model frequently involves passing through one or more intermediate stages. Each intermediate stage may see elements being temporarily deleted, renamed and so on. If the changes from these intermediate stages are propagated, it is possible that incorrect, and irreversible, changes may be made to the target model. Consider a tool which allows a user to 'cut' a model element to a clipboard, who then intends to paste the element to another part of the model later. If such a change is propagated immediately, it will lead to the deletion of target elements. Such elements may contain manual changes or additions in the target model; when the element is deleted, the manual changes will be lost and will not be replaced when the source element is 'pasted' back into the model. Since only

the user can know the intended end goal of their sequence of actions, immediate change propagating transformations pose an extra set of challenges for such scenarios.

6.1.4. Relating source and target elements by key, trace, or identifier

One of the chief challenges when propagating changes is to find a mechanism for relating, or distinguishing, the specific target elements created by a given rule relative to specific source elements. The distinguishing of elements is vital to ensure that target elements are modified, created or deleted correctly during change propagation. This problem is largely irrelevant during the initial run of a change propagating transformation, but is vital when subsequently propagating changes; this problem was outlined by example in section 2.2.3.

Johann and Egyed present a basic, high-level overview of this subject, describing the distinguishing of elements by key and by identifier [JE04]. For the purposes of this thesis I identify three chief ways of relating or distinguishing which target elements are related to specific source elements: by key, by trace, and by identifier. I now outline these three possibilities in more detail.

Distinguishing target elements by key

A simple mechanism for distinguishing elements is to do so on their key i.e. a collection of attributes which, collectively, uniquely identify any given element. Using this mechanism for change propagation is advocated by the DSTC QVT approach [DIC03]. By requiring elements to be defined in keys, this mechanism implicitly adds an extra burden on the user since all elements in a model must be augmented with a key definition. Although this is often trivial, it is an extra burden, and can be difficult when elements have no natural key.

The essential idea of propagating by key is that when changes from an element need to be propagated, the source element is transformed (possibly to a temporary location), and the key of the target element is extracted. This then allows the changed parts of the target element to be merged with an existing target element with the same key. However this means that modifying the values of attributes involved in a key confuses the propagation algorithm. Consider the transformation from and to a simple modelling language where the key of a `Class` is its `name` attribute. If a class named `x` is transformed to a class also named `x`, then many changes made to the source model (e.g. adding attributes) can be trivially propagated to the relevant target element by transforming the source models key and finding the target element with the appropriate key. However if the source element is renamed to `y` then the key relationship between the source element `y` and target element `x` is broken; the change propagation algorithm will assume that the relevant target element has been deleted, and will recreate it from scratch.

Although not mentioned in the DSTC QVT approach, one technique which may potentially improve the coverage of this technique is to use the previous generation of a source element to calculate the key of the appropriate target element. This allows changes to be propagated successfully even when source elements have had the values of attributes involved in their key altered. However it is unable to cope when manual changes are made to a target elements' key.

In the general case, propagation by key is insufficient. However it may be combined with other propagation techniques to increase coverage.

Relating target elements via tracing information

Using the tracing information created by a transformation (see section 5.4) to relate source and target elements seems a good candidate, particularly as the information already exists. However, as shown in MT, there are various different tracing information creation mechanisms. The success of a change propagation algorithm then depends on factors such as the coverage and granularity of the recorded tracing information. For example, while the default tracing information generated by MT records which target elements were created by a rule from specific source elements, it does not generate enough information to know which part of the rule created which target element. Such information may be vital for an accurate change propagation algorithm.

There is thus a potential tension between the different uses of tracing information. The type of tracing information desirable for change propagation may be very different from that required by a user to understand transformations on their model. However, assuming that it is suitably detailed, tracing information is sufficient as the sole means of distinguish elements for change propagation.

Distinguishing target elements by identifier

A technique that can ultimately be seen as a slight variation on distinguishing target elements by tracing information was detailed by this author in [Tra05], and independently by Johann and Egyed in [JE04]. When a target element is created it is given an identifier which contains, at a minimum, the concatenated identifiers of all the source elements which led to the creation of the target element. Henceforth I refer to this as the *target element identifier*. Note that the target element identifier may be in addition to an elements standard identifier, and that conceptually there is no requirement that this new identifier be a single field.

Conceptually this technique does not add any additional power over using tracing information to distinguish elements; it is an alternative way of storing tracing information. Indeed, a simple concatenation of the source elements identifiers means that the target element identifier is merely an alternative way of storing information that can in theory be directly derived from suitably fine-

grained tracing information. However extra information can be easily stored in the target element identifier, if required, to allow a transformation to encode information which may not be present in tracing information. This then allows tracing information to be used for other purposes. Furthermore this then means that tracing information need neither have complete coverage, nor be fine-grained; as such, tracing information can be recorded in a fashion which gives it the greatest utility to the user.

6.1.5. Correctness checking and conflict resolution

Some changes made to a source model may not be able to be propagated successfully to the target model. For example, when propagating an element newly added to the source model, a conflict may arise with an element already present in the target model. There are three main strategies that can be taken in such cases:

1. Propagate all changes regardless of correctness conditions, accepting that the resulting target model may not match expectations, and may even be ill-formed.
2. Check for the correctness of changes before propagating them; refuse to propagate changes which will violate correctness conditions.
3. Propagate all changes which do not violate correctness conditions; note those which violate such conditions and request manual intervention from the user.

Whilst the first strategy requires little extra support, in the cases of the second and third strategies change propagating model transformation approaches have to decide upon the form of correctness checking, its completeness, and its ability to be controlled by users.

6.2. PMT

PMT's implementation began as a fork of MT, and can be considered initially to be a superset of MT. Most valid MT transformations can be moved into PMT without syntactic change — when used as a stateless model transformation language, PMT performs largely as MT. When compared to the design decisions detailed in section 6.1, PMT can be said to be a fully automatic, batch change propagation approach, which distinguishes target elements by their identifiers, and which has user controllable correctness checking built in. The details of this broad overview will be filled in as this chapter progresses.

Despite many similarities, the sequence of running a PMT transformation is fundamentally different from MT. An MT transformation is initialized with one or more source elements which are immediately transformed into target elements. In contrast, a PMT transformation is initialized with

a source model, a (possibly empty) target model, and a (possibly empty) set of tracing information. Unlike MT, source elements are not immediately transformed after initialization, waiting for the transformation to be executed by the user. Since none, parts, or all, of the target model may be present after the initialization of the PMT transformation, the concept of rule execution in PMT is markedly different in MT. In MT, when a rules source clauses match its input, the execution of the rule implies the production of new target elements. In PMT, when a rules source clauses match its input, the execution of the rule implies that the target model is modified to make it conformant with respect to the transformation. Although from a naïve users perspective there is a difference between the initial execution of a PMT transformation – which appears to populate an empty target model – and subsequent executions which propagate changes, from PMT's perspective there is no difference between the initial and subsequent executions.

Put crudely, the difference between MT and PMT is that the former is an imperative model transformation language whilst the latter is declarative. Conceptually, the execution of a PMT rule is fundamentally different from MT. When a PMT rule is executed, it attempts to make the necessary changes to the target model to satisfy the rules declaration. This may require elements being added, altered and deleted from the target model. The way in which the relationship between source and target elements is specified, and the process by which the update of the target model occurs are the two defining aspects of PMT.

6.2.1. A PMT transformation's stages

The stages of a PMT transformation are as follows:

1. Take a source model, and an empty target model and transform the source model. This stage – if taken in isolation and viewed as a black box – is essentially identical to an MT transformation. After the transformation has executed, the source and target models, together with the tracing information created, are stored in some fashion.

2. The user may make arbitrary changes to both the source and target models, independent from one another.

3. The user then requests that the changes they have made to the source model are propagated non-destructively to the target model. The transformation is reinitialized with the updated source and target models, and the tracing information from the previous execution. The execution of the transformation then propagates changes from the source model to the target model. After the transformation has executed, the source and target models, together with the new tracing information created are once again stored.

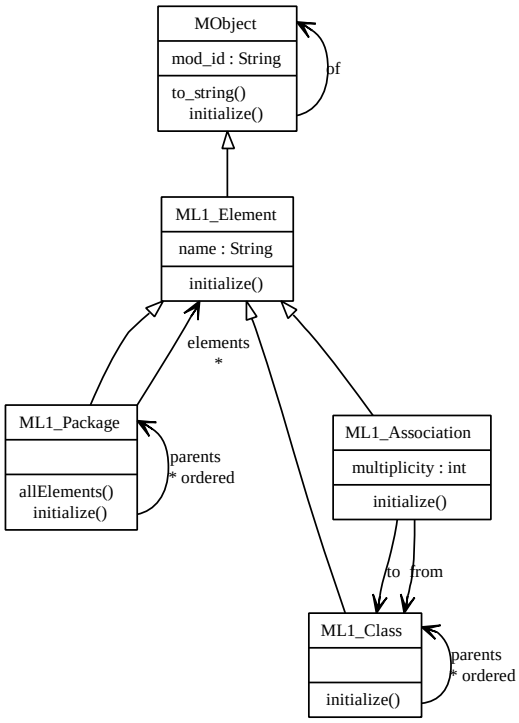


Figure 6.1.: The ML1 modelling language.

At this point, the sequence moves back to stage 2.

6.2.2. Example

This subsection presents a simple example of change propagation, which is based on the change propagation example from section 2.2.3. That example showed the conceptual problems of a change propagating transformation from the ML2 to the ML1 modelling language. The metamodels of the ML1 and ML2 modelling languages are shown in figures 6.1 and 6.2 respectively.

The transformation itself is as follows:

```
1  $<PMT.mt>:
2    transformation ML2_to_ML1
3
4    rule Package_To_Package:
5      srcp:
6        (ML2_Package)[name == <n>, elements == <elements>]
7
8      tgtp:
9        (ML1_Package)[name := n, elements :>= tgt_elements]
10
11     tgt_where:
12       tgt_elements := Set{}
13       for x := elements.iterate():
14         tgt_element := self.transform([x])
15         if tgt_element.conforms_to(List):
16           tgt_elements.extend(Set(tgt_element))
17         else:
18           tgt_elements.add(tgt_element)
19
20   rule Class_To_Class:
21     srcp:
```

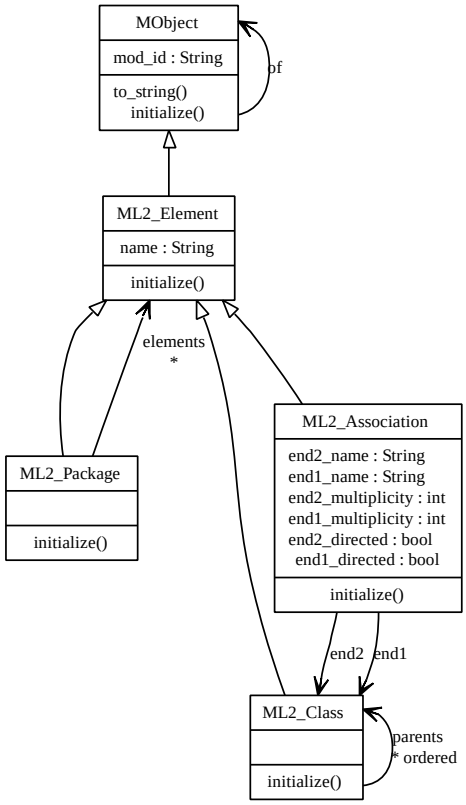


Figure 6.2.: The ML2 modelling language.

```
22        (ML2_Class)[name == <n>]
23
24      tgtp:
25        (ML1_Class)[name := n]
26
27    rule Association_To_Association:
28      srcp:
29        (ML2_Association)[name == <n>, end1 == <end1>, end2 == <end2>, \
30          end1_directed == 0, end2_directed == 0, \
31          end1_multiplicity == <end1_multiplicity>, \
32          end2_multiplicity == <end2_multiplicity>, end1_name == <end1_name>, \
33          end2_name == <end2_name>]
34
35      tgtp:
36        (ML1_Association)[name := end2_name, from := tgt_end1, to := tgt_end2, \
37          multiplicity := end2_multiplicity]
38        (ML1_Association)[name := end1_name, from := tgt_end2, to := tgt_end1, \
39          multiplicity := end1_multiplicity]
40
41      tgt_where:
42        tgt_end1 := self.transform([end1])
43        tgt_end2 := self.transform([end2])
```

This is an intentionally simple transformation which, in the interests of brevity, ignores parent packages and only handles associations which are navigable at both ends. Since converting ML2 classes and packages to ML1 classes and packages is exceedingly trivial, the `Package_To_Package` and `Class_To_Class` rules are simple (lines 12 - 18 are a largely inconsequential implementation detail that essentially normalizes the return value from other transformation rules). The `Association_To_Association` rule is slightly more complex, although it only deals with associations

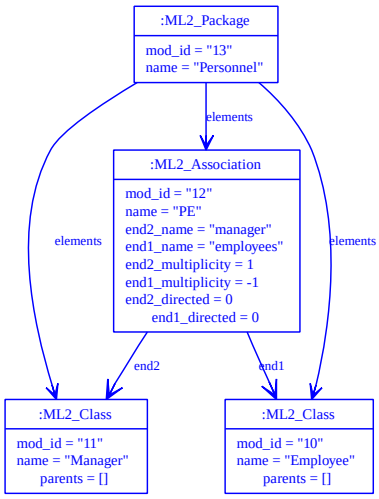


Figure 6.3.: Initial source model for the ML2 to ML1 transformation.

which are navigable at both ends; each such ML2 bidirectional association is transformed into two ML1 directed associations.

The initial source model I use for this transformation is shown in figure 6.3 (note that this is merely the TM version of figure 2.3(a)). The resulting visualization of the transformation is shown in figure 6.4. At this point, there are only two hints that we are dealing with a PMT, and not an MT, transformation definition and execution: the `:>=` operator in line 9 is invalid in MT; identifiers in the target model have a noticeably different format to those in MT transformations.

Let us now assume that the user has modified the target model as in figure 6.5, adding in a directed association from `Employee` to `Manager` denoting an employee's secondary manager. Let us then assume that the user returns to the original source model and updates it as in figure 6.6, adding in a `DepartmentHead` class and an associated transformation. If the ML2 to ML1 transformation was an MT transformation, the user would now have two choices. If they were to rerun such a transformation, the original target model would be overwritten and the `secondary_manager` association would not exist in the new target model. Alternatively the user could choose to manually port the changes from the source model to the target model. In the former scenario, changes to one or the other model are lost; in the latter, differences must be manually propagated between models.

It is at this point – corresponding to stage 3 as described in section 6.2.1 – in the transformation execution cycle that PMT fundamentally distinguishes itself from MT, by automatically propagating the changes made to the source model in figure 6.6 into the updated target model. The visualization of the target model after change propagation can be seen in figure 6.7. As this figure shows, not only have the changes to the source model been propagated into the target model, but the manual changes made to the target model by the user have been preserved. It is important to note that the changes made to the source and target models by the user in this example are entirely arbitrary.

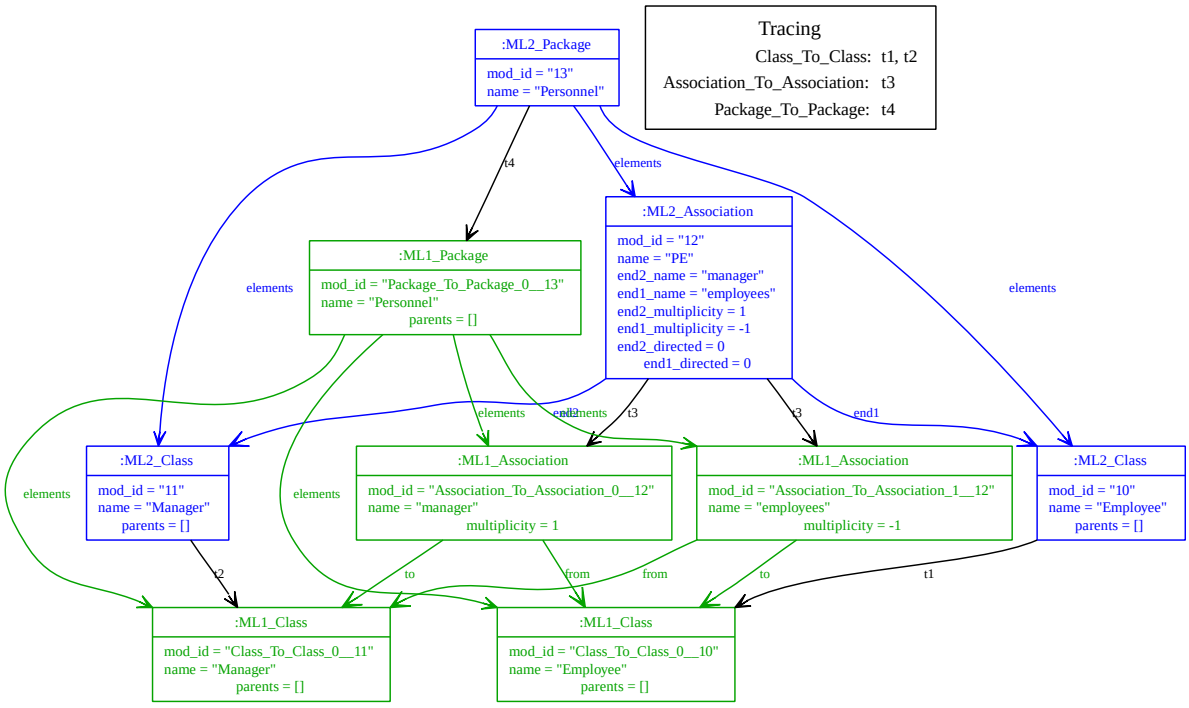


Figure 6.4.: Visualization of the initial execution of the ML2 to ML1 transformation.

The basics of PMT's change propagation approach are very simple. Both model element patterns and model element expressions play a key part in the process of propagation. PMT uses model element patterns as the primary means of calculating target element identifiers (see section 6.1.4). When a rule is executed, and its source clauses match successfully, a target element identifier is created, based on unioning the identifiers of the source elements matched by model element expressions. Target element expressions in the target clauses use the target element identifier created by the source clauses. When a model element expression is executed, it looks in the TM object repository to see if an element with the same identifier as the target element identifier already exists. If no such element exists, a new model element with that identifier is created and populated accordingly. If such

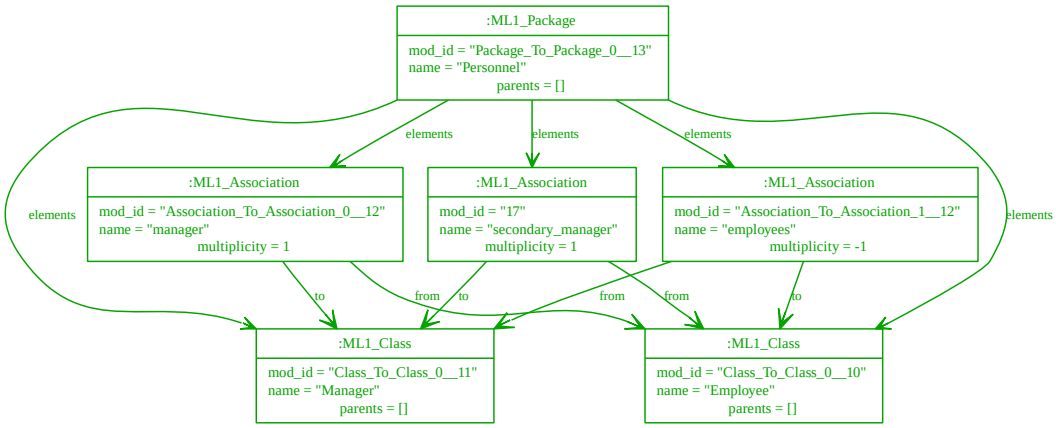


Figure 6.5.: The updated target model.

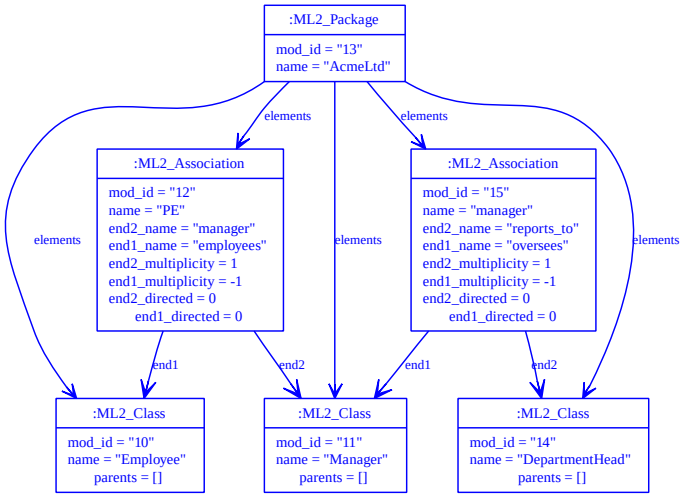


Figure 6.6.: The updated source model.

an element exists, it is taken from the object repository and its contents are adjusted as necessary to satisfy the transformation. Sections 6.2.3 and 6.2.4 explain the creation of identifiers and altering of elements in more depth.

6.2.3. Creating target element identifiers

The construction of target element identifiers is a vital part of PMT's change propagation approach. Target element identifiers should ideally satisfy two criteria: that they are unique with respect to particular source elements and a particular rule execution; that they can be created deterministically across multiple transformation executions. The need for the former criteria is self evident, the latter perhaps less so. However PMT's approach relies on the fact that the construction of target element identifiers can be replicated over multiple transformation executions. Since satisfying either, or both, of these two criteria is non-trivial, I consider it highly desirable that target element identifiers can be automatically created and used without burdening the user unnecessarily. In this subsection I outline in detail how PMT automatically creates target element identifiers; this process is somewhat more involved than its description in previous sections has suggested.

The way in which target element identifiers are created and stored makes use of two internal TM and PMT features. Firstly, as shown in figure 4.4, the identifier of a TM model element is a string. Unioning identifiers thus becomes a case of simple string concatenation which, whilst not an entirely robust technique, is adequate for the purposes of this thesis. Although TM supplies a default identifier, a user supplied identifier – such as a PMT target element identifier – can be specified when elements are created. Secondly, PMT uses the concept of model elements matched by model element patterns – exactly as used by the tracing information creation mechanism (see section 5.4) – to determine which source elements will have their identifiers unioned. Thus creating target element identifiers requires

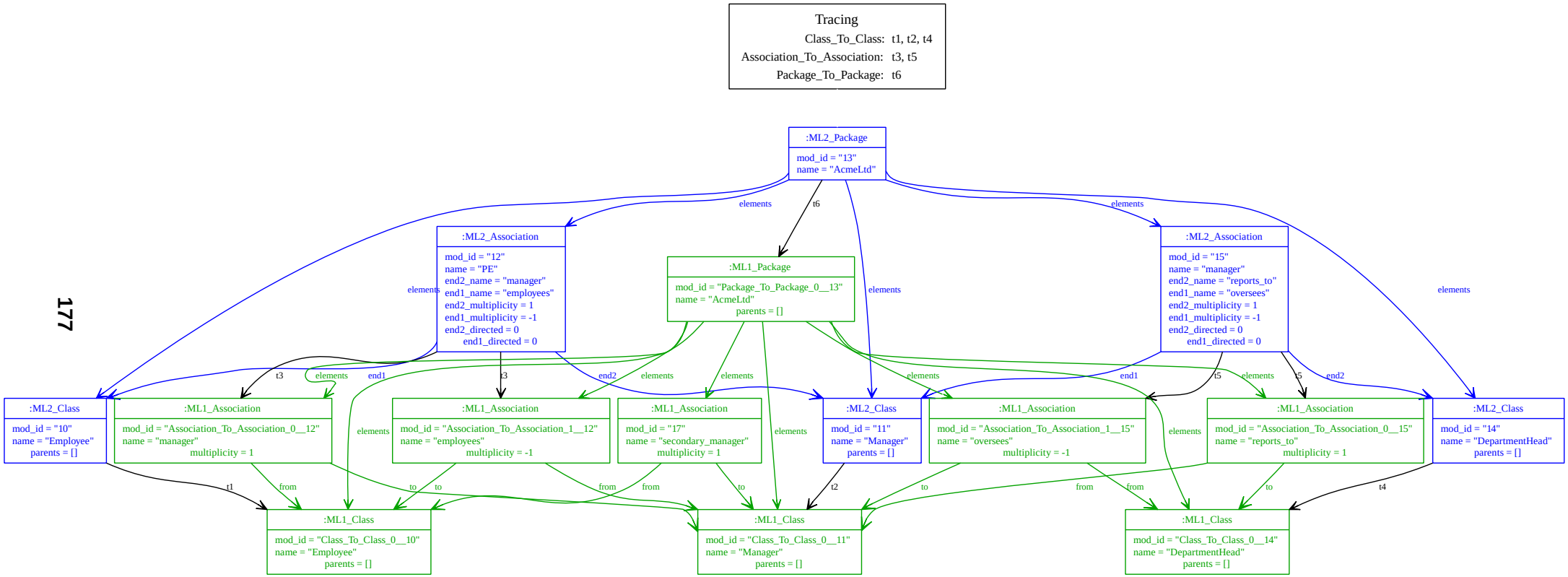


Figure 6.7.: Visualization of the ML2 to ML1 transformation after change propagation.

no new underlying machinery in the implementation.

Creating unique target element identifiers

Concatenating the identifiers of source elements is not sufficient on its own to generate a unique target element identifier, since the same source elements may be used in more than one rule execution. PMT thus also integrates the name of the rule being executed into the target identifier to ensure that target element identifiers are unique. However this then raises the possibility that executing the same rule with the same source elements may lead to conflicting target identifiers being generated. To avoid this possibility, PMT rules keep a cache of source elements they have already transformed; if a rule matches against the same source elements as it did in a previous execution, then the target elements produced in that previous execution are returned. It should be noted that this is different from MT, which does not need to enforce such a constraint during its execution. This may potentially lead to differences in the execution of seemingly identical MT and PMT transformations.

The rules given thus far generate a single unique target element identifiers. This is sufficient when a rules target clauses contain a single model element expression which executes only once. If a rule has multiple model element expressions in its target clauses, or if a model element expression can execute more than once in a single execution of a rule (e.g. when a model element expression is suffixed with `for`, as in MT), then a single target element identifier would result in multiple target elements being created with the same identifier. For example, the `Association_To_Association` rule in section 6.2.2 has two model element expressions in its `tgtp` clause. In such cases it is vital that each model element expression is passed a unique target element identifier. In order to ensure that this is the case, each rule execution keeps a counter of how many times model element expressions have been executed during the rules execution. This counter is incorporated into the target element identifier of model element expressions, thus ensuring the uniqueness of the identifiers even when a rule executes more than one model element expression.

The general form of a target element identifier in PMT is as follows:

```
<rule name>_<model element expression execution #>__<union of source
  identifiers>
```

Using this template, one can interpret the identifiers of target elements in figure 6.7 with respect to the transformation of section 6.2.2.

It should be noted that in the current implementation when primitive data types are used in model element expressions, it is possible for PMT to generate non-unique identifiers, since instances of primitive data types do not have a proper element identifier. I consider this to be a relatively trivial implementation detail.

Deterministically creating target element identifiers

It is important for PMT that the target element identifiers it create be deterministic; that is, if a transformation is rerun with exactly the same source elements as before, it should create exactly the same target element identifiers. If target element identifiers are created differently over multiple transformation executions then PMT will not able to identify target elements correctly. Although the scheme outlined previously has proved reasonably successful in practise, using the model element expression execution counter leads to a subtle, but potentially significant, flaw.

Non-ordered datatypes such as sets can cause the model element expression execution to become de-synchronised over multiple transformation executions due to their inherent non-determinism. Similarly, ordered data types such as lists can have elements inserted in them in-between transformation executions; if elements are inserted at any point other than the end of the ordered datatype, then the counter can again become de-synchronised.

A possible solution to this problem is as follows. Each model element expression in the target clauses is statically assigned a number, starting from 1, and incremented with each model element expression encountered during compilation. For model element expressions that can only be executed once, this is sufficient to ensure uniqueness and determinism of the resultant target element identifiers. For model element expressions which can be executed more than once, it is then necessary to add something further to the target element identifier to ensure uniqueness. For example, one could determine which source elements (which, in general, one would expect to be a strict subset of the overall source elements matched by a rule) led to the creation of that particular model element, and make their identifiers part of the target element identifier; note that in this scheme it would be common for source element identifiers to appear more than once in a target element identifier. In some cases PMT may be able to automatically determine which source elements are involved in the creation of specific target elements, but in general this is not possible; the user will therefore need a way to inform PMT of the required information. Note that whilst this solution is largely immune to non-determinism problems, it still has some conceptual problems e.g. when dealing with ordered lists which contain duplicated elements.

While solutions such as the one outlined may provide a more robust approach to creating target element identifiers, I believe that further research will be needed to find the best solution. For the purposes of this thesis, PMT's current solution, whilst not robust, is adequate for exploring change propagation.

6.2.4. Making target elements conformant

When a model element expression is executed, it looks in the TM object repository to see if an element with the same identifier as the target element identifier already exists. If no such element exists, PMT executes largely as MT. However if such an element exists, PMT executes rather differently from MT. The object in question is taken from the object repository and PMT and is altered into a form conformant with the model element expression.

It is important to note the use of language in this subsection. When an element already exists it is not necessarily changed to match the exact values dictated by the model element expression. Instead the element has the minimal number of changes applied to it that make it conformant to the model element expression. The word 'conformant' is important since, in the general case, an infinite number of differing target elements may be conformant to a given execution of a model element expression. This is because the user can make manual changes and additions to the target model which the transformation writer can, if they choose, allow to remain even when changes are propagated.

In order to achieve this, model element expressions in PMT have additional syntax compared to MT. Most importantly a model element expression in PMT comprises zero or more *slot conformances* (which are directly analogous to slot comparisons in model element patterns). In the example shown earlier, one can see the use of two *conformance operators*. PMT's conformance operators are partially inspired by operators found in xMOF (see section 3.3.9). Some conformance operators are as follows:

Operator	Name	Description
`x := y`	*update*	Forcibly sets the value of slot `x` to `y`.
`x :== y`	*update if not equal*	If the value of slot `x` is not equal to `y`, forcibly sets the value of slot `x` to `y`.
`x :>= y`	*update superset*	The value of slot `x` must be a non-strict superset of `y`'s value. Any elements in `y` not present in `x` will be added to `x`. `x` may contain elements not present in `y`.

The update conformance operator forcibly propagates changes from the updated source model to the target model. The update if not equal conformance operator performs the same action, but only after checking that the value of the slot in the target element is not equal to the value generated by its associated model expression. In practise, the two operators are very similar; however, since in some cases distinct objects can compare equal the user may wish to specify precisely whether they wish the slot value and model expression to hold exactly the same value, or merely two values which are equal. The update superset conformance operator is more interesting since it does not imply,

or force, the value of the slot in the target element to be directly equal to the value generated by the model expression. Instead, the value of the slot in the target element is altered to make sure it contains all the elements that the model expression says it should have; if it has extra elements then those are left intact. In practise this operator is the chief means of allowing changes to be propagated non-destructively.

One important point that may not be immediately obvious is that transformation writers still need to use careful thought to determine when each should be used. For example, an inexperienced transformation writer may choose to use the update operator in all slot conformances, since this will ensure that all changes made to the source model. However if the slot in question contains a set then the users' manual changes made in the target model will be destroyed. In such cases, one would generally expect the transformation writer to use the updating slot conformance operator. In some cases, however, the transformation writer may deliberately wish to ensure that the target model contains the transformed set elements, and nothing else, in which case the update conformance operator is the correct choice. Knowledge of the appropriate situations for each conformance operator is likely to be gathered only through knowledge of the source and target domains, and experience with the change propagating approach.

Later in this chapter I will examine other conformance operators. However the three conformance operators detailed in this section are currently the only ones which forcibly alter target elements (the other conformance operators described in section 6.4 check, rather than enforce, conformance). The reason for this is that, between them, these operators appear to cover a very large part of the spectrum of change propagation – certainly, they are sufficient for all examples in this chapter.

Changes which can not be propagated

There are various types of changes which PMT is incapable of propagating. The most obvious class of such problems relates to when the propagation of a change results in an ill-formed model (i.e. one which does not conform to its meta-model). In such cases, a standard TM exception is thrown, and the user is informed. Whilst this is currently a somewhat crude mechanism, it does prevent incorrect target models being created. The checking conformance operators detailed in section 6.4 provided an alternative means of detecting, and reporting, changes which can not be propagated.

6.2.5. Running a PMT transformation

Running a PMT transformation is very different to MT (see section 5.3.6), which is largely a direct result of the underlying conceptual difference between a stateless and a change propagating model transformation approach. An MT transformation is passed a source model which it instantly trans-

forms into a target model, creating tracing information as it executes. Since a PMT transformation may be executed multiple times, and since between executions its data may have been serialized to permanent storage, it operates in a fundamentally different fashion.

When run for the first time, a PMT transformation is initialized with only a source model. After the transformation executes, the user can extract the target model and tracing information created during the transformations execution. There are then two scenarios before change propagation will occur. The first scenario is that, whilst the transformation is still 'active', the user modifies the source and target models. Propagating changes then becomes a simple case of re-executing the transformation, which will automatically pick up the changes made to the models. The second scenario is that after execution, the source and target models, along with the tracing information, are serialized to a persistent store. The transformation itself is then destroyed. Subsequent executions of the transformation thus require the transformation to be reinitialized with the possibly updated source and target models, and the tracing information (which must not have have been changed), all of which will have been deserialized from their persistent store. Once suitably reinitialized, the transformation can then be executed to propagate changes. Both these scenarios are likely to occur in the real world. Whilst the former scenario is likely to occur in short-lived tasks, or when efficiency is key, the latter scenario reflects the practicalities of long-term use and development of particular models. PMT transformations are designed to deal sensibly with both scenarios.

The code to run the example of section 6.2.2 looks as follows:

```
employee := ML2.ML2_Class("Employee")
manager := ML2.ML2_Class("Manager")
employee_manager := ML2.ML2_Association("PE", employee, manager, 0, 0, -1, 1, \
  "employees", "manager")
personnel := ML2.ML2_Package("Personnel", Set{employee, manager, \
  employee_manager})

transformation := ML2_to_ML1(personnel)
transformation.do_transform()
```

The unassuming, but important, difference between this and running an MT transformation is the `do_transform` function on a transformation object. This function can potentially be called multiple times. Each time it is called it will propagate changes from the source model to the target model.

Extracting the target model and tracing information from a PMT transformation is identical to MT. For those instances when models need to be serialized to a persistent store, the TM package defines a `Serializer` module. This is capable of serializing (i.e. saving) and deserializing (i.e. loading) models and tracing information via the `serialize`, `serialize_tracing`, `deserialize`, and `deserialize_tracing` functions. A slightly simplified version of the code which serializes the ML2 to ML1 transformation is as follows:

```
src_file.write(Serializer.serialize(transformation.get_source()))
tgt_file.write(Serializer.serialize(transformation.get_target()))
tracing_file.write(Serializer.serialize_tracing(transformation.get_tracing(), \
```

```
    transformation.get_tracing_rules()))
```

Appendix E.2 shows the output from serializing the source and target models, and tracing information after the first execution of the example in section 6.2.2.

Reinitializing a PMT transformation involves initializing the transformation not only with the updated source and target models, but also with the tracing information generated on the previous transformation run. The tracing information generated by the previous execution does not play a direct part in the transformation; it is used to determine which elements can be safely deleted from the target model (see section 6.2.6). An entirely fresh set of tracing information is generated on each execution. A simplified version of the code which deserializes the ML2 to ML1 transformation, and propagates changes is as follows:

```
src_model := Serializer.deserialize(src_file.read())
tgt_model := Serializer.deserialize(tgt_file.read())
old_tracing, old_tracing_rules := Serializer.deserialize_tracing( \
  tracing_file.read())

transformation := ML2_to_ML1(personnel)
transformation.set_target(tgt_model)
transformation.set_old_tracing(old_tracing)

transformation.do_transform()
```

Models can be transformed, serialized, altered and have changes propagated into them an arbitrary number of times.

6.2.6. Removing elements from the target model

An important part of change propagation is to ensure that when elements are removed from the source model, target elements which were created by transforming the source elements in question are removed from the target model. This requirement may at first appear to be solved by examining all target elements at the end of a transformation execution, and removing all target elements which were not created as the direct result of transforming one or more source elements. However this simple solution would also delete any elements manually added to the target model by the user, and as such is clearly not suitable for the use cases PMT is aimed at. The critical problem is therefore to distinguish which seemingly superfluous elements in the target model have been manually added by the user, and which are no longer a part of the transformation.

In order to determine which elements can be safely deleted in the target model, PMT utilises tracing information – both that generated by an execution of the transformation, and that generated by its previous execution. After changes have been propagated, a PMT transformation examines every element in the target model, checking whether it is referenced in either or both of the current and previous tracing information. Based on this, PMT draws a conclusion about the origins of the element and whether it is a candidate for removal. The four possibilities for an element are as follows:

In previous tracing info.?	In current tracing info.?	Conclusion	Candidate for removal?
√	√	Target element previously manually created by PMT.	×
×	√	Target element newly created by PMT.	×
×	×	Target element previous added to target by user.	×
√	×	Target element previously created by PMT; corresponding source element now deleted.	√

Once every element has been examined, PMT performs a garbage collection style 'mark and sweep' [JL99], using the transformed root set of source elements as the starting point. Any self-contained cycle consisting solely of elements marked as being candidates for removal, is then removed from the target model. The need to identify self-contained cycles of such elements is to prevent the removal of elements cause the target model to become ill-formed. This could occur if elements are removed from the model even though they are referred to by other objects. An example of elements being removed after change propagation can be seen in section 6.3.2.

6.2.7. Propagating changes between containers

Propagating changes between containers (e.g. sets and lists) raises two challenges not tackled earlier. The first relates to the removal of elements in containers. The second challenge relates to the synchronising of ordered containers. In this subsection I detail PMT's solutions to these challenges.

Removing elements when propagating changes between containers

When elements are deleted from a container in a source model, and that container is transformed into a container in the target model, PMT needs to be able to work out which elements in the target container should be removed. This is a less than easy task because PMT needs to distinguish elements in the target container which have been manually added by the user, and those that are the result of transforming a now absent source element. In order to make this distinction, PMT uses a technique similar to the general element removal technique of section 6.2.6.

When the updating superset operator attempts to propagate the changes from a container y to a slot x's value in a target element, it first adds every element of y to x's value if it is not already present therein. It then iterates over x's value, noting any elements in x's value which are not present in y. When it finds such elements, it first checks to see if the element is present in the tracing information of the previous transformation execution. If the element is not present, PMT assumes the

additional element in x's value is a manually added element, and ignores it. If the element is present in the previous transformation execution's tracing information, PMT assumes that the element was originally added to the container by PMT, and can now be removed from the container.

Due to a lack of sufficiently fine-grained information, this scheme has one notable problem – if a user manually adds a target element into a container, and the source element that led to the creation of that particular target element is subsequently deleted, then the element will be erroneously removed from the container upon change propagation. Note that does not imply that the element will necessarily be removed from the model; the element will only be removed – in the mark and sweep phase – if its membership of the container was its only reference within the model.

Propagating change in ordered containers

Propagating changes to ordered containers is considerably more complex than into unordered containers. Not only are elements ordered, but the same element may appear more than once. This means that, for example, it is not acceptable to merely check for the existence of a given element, since it may appear more than once. Similarly, between transformation executions, elements may move their position within a list. When a user is adding, removing, or moving elements within an ordered container, the purpose of each individual change is generally self-evident to them. From the point of view of a system viewing an arbitrary number of such changes, any such intentions are lost.

The update superset conformance operator takes a simple minded approach to the problem. Given a target slot x, and an ordered container y, it will ensure that x's value contains every element of y in the order that those elements are contained within y. However it will tolerate an arbitrary number of extra elements within x. Elements from y are added into x as necessary. Looked at a different way, this mechanism ensures that there is an ordered sublist of x which is exactly equal to y. This scheme is less than ideal, since it can lead to an incorrect duplication of elements in the target container.

6.3. The execution of a PMT transformation

Up until this point I have been deliberately vague on exactly what actually happens when a PMT transformation is executed. The reason for this is that PMT's execution strategy runs contrary to a standard intuition – as exemplified by Johann and Egyed [JE04] – of change propagation in operation. By deferring the explanation of a PMT transformation until this point in the chapter, I hope that enough material has been presented to make explanation of this vital point practical.

Intuitively, the concept of change propagation seems simple: given a change in the source model, one simply needs to rerun the few transformation rules which relate to the changed source elements

in order to propagate the change to the target model. For many small, localised changes – such as the renaming of a class, as seen in the earlier example in this section – this strategy is adequate. Whilst this intuition is highly appealing, it leads to a solution that can not propagate many types of changes correctly. At best this may lead to a target model that is not synchronised with the source model; at worst, it may cause the target model to become ill-formed.

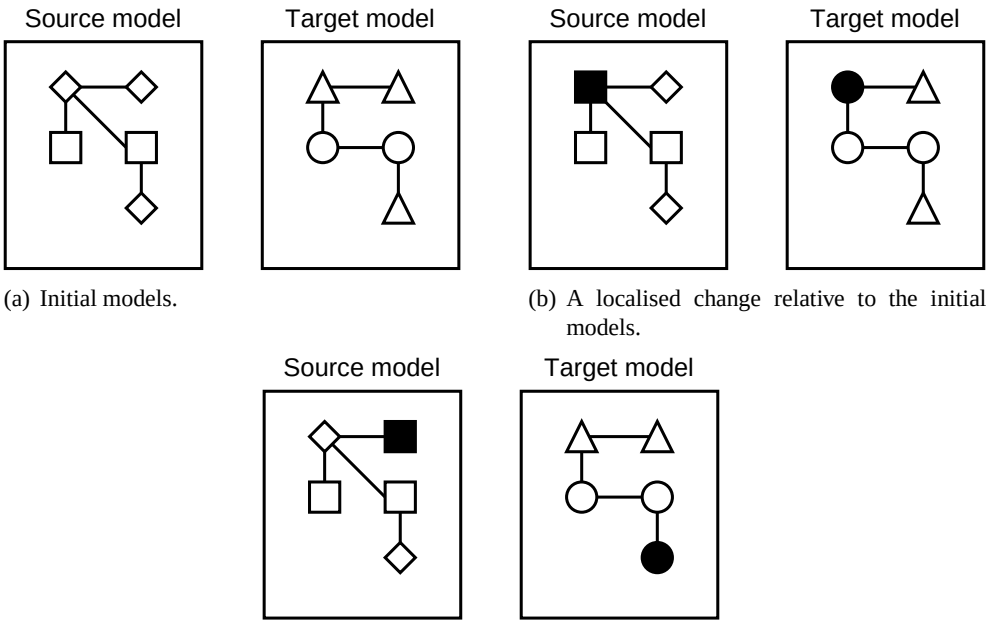
In this section I first point out the problems with the intuitive change propagation approach, before presenting PMT's approach to transformation execution.

6.3.1. Propagating localised changes

The change propagating example of section 6.2.2 saw two main types of changes to the source model: the alteration of the values of elements fields (e.g. changing a packages name), and the addition of elements. The former type of change is intuitively simple to propagate. When the `Personnel` package was renamed to `AcmeLtd` in figure 6.6, all that is required to propagate the change is to rerun the transformation rule(s) linked to by the tracing from the source element. A quick examination of figure 6.4 shows that rerunning the `Package_To_Package` rule with the source package in question as input will result in the change being correctly propagated. The latter type of change is slightly more complex, but intuitively somewhat similar. One approach would be to first pass the new source element to the `transform` function; any source elements which have new links to the new element will be transformed using the same approach as for propagating the change in package name.

The fundamental premise behind this intuitive notion is that the propagated changes are what I term *localised*. Note that this term does not directly relate to the locality of alterations in the source model, but instead to the locality of the necessary changes to be propagated to the target model and the relation of those changes to the altered source elements. Figure 6.8 shows an abstract example of a transformation, and localised and non-localised change propagation. If changes are localised, then changes to elements in the source model can be propagated by rerunning the rules which originally applied to the those elements. This has two implications. Firstly, that changes in the source model will lead to changes in the target model of a similar granularity; in other words, that changes local to a particular part of the source model should lead to similarly local changes in the appropriate part of the target model. The second implication follows from the first: that the source and target models are likely to be mostly, or wholly, isomorphic.

Before I justify these two implications, it is instructive to see why they are implicit in the, rather limited, literature on the subject. For example, Johann and Egyed [JE04] describe a system that is almost wholly targeted at localised changes; despite not being directly model related, Varró and Varró describe a similar system [VV04]. By assuming that changes are localised, both approaches



(a) Initial models.

(b) A localised change relative to the initial models.

(c) A non-localised change relative to the initial models.

Figure 6.8.: The concept of localised changes.

are able to make change propagation highly efficient by only running the rules directly related to a particular change. The ability to highly optimise change propagation in the face of localised changes is a compelling reason to treat such changes as a special case. Unfortunately neither approach is capable of propagating non-localised changes correctly. Johann and Egyed [JE04] describe what they term 'semantic changes' as 'simple changes in the source model that cause a variety of ripple effects among multiple/many target elements', but do not present a solution to this problem. I believe the reason for this omission is that many toy transformations, such as the example of section 6.2.2, are expressed in such a way that only localised changes will ever need to be propagated.

Non-localised changes in practise

Two concrete examples demonstrate the problem of non-localised change. In order to demonstrate this, I return to the advanced variant of the UML modelling language to relational database transformation, as defined in section 5.5.1. I assume that the hypothetical change propagating transformation which would perform this task follows a similar structure to the MT solution for this problem, as defined in section 5.5.3.

Consider first the (slightly elided) source model of figure 6.9, and the corresponding target model in figure 6.10. Imagine first what would happen were we to change the value of the `is_persistent` slot in `Address` class of figure 6.9 to `1`. When we execute the transformation to propagate transformations, intuitively we would expect to see the target model contain two tables, and for all the columns

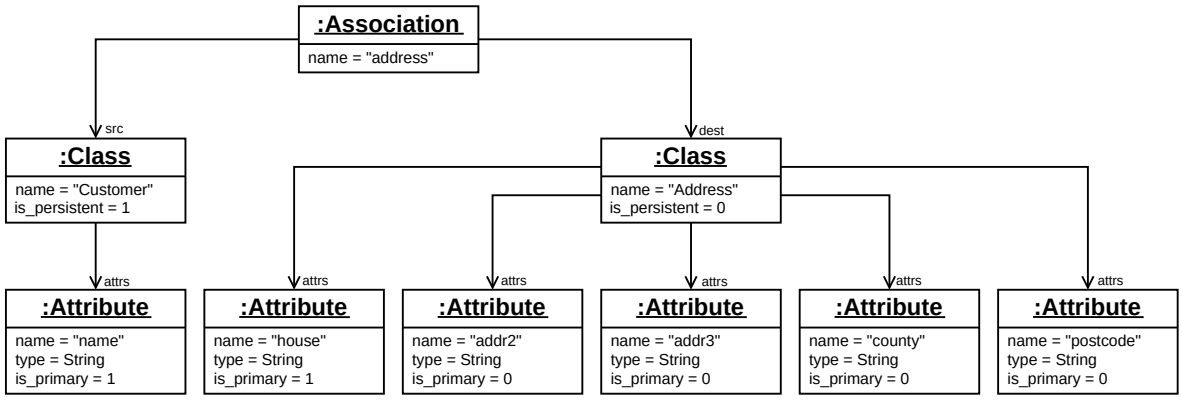


Figure 6.9.: Source model.

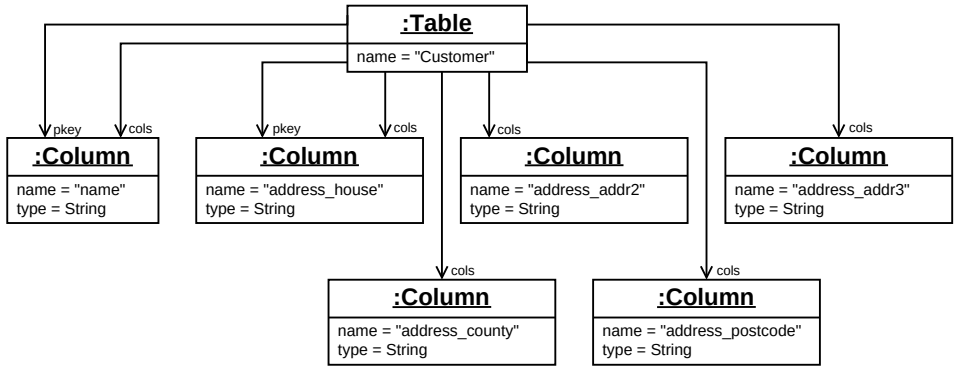


Figure 6.10.: Target model.

prefixed with `address_` to be removed from the `Customer` table. Using a technique similar to that outlined by Johann and Egyed, this intuitive idea may or may not be matched by reality. In the initial transformation execution the `Association_Non_Persistent_Class_To_Columns` would have matched the `Address` class and transformed it. However by marking it as persistent, that rule is no longer able to match (the `Persistent_Association_To_Columns` would however now match), and so change propagation can not occur using the original rule. Johann and Egyed are vague as to what happens when an alteration to the source model means that change propagation can not occur with the original rule which transformed that element. However one can imagine that when such a case is detected the transformation system would look for a different rule which does match the changed source element.

Taking the same source model of figure 6.9, and the corresponding target model in figure 6.10 as the basis for the second example, consider the effect of changing the `postcode Attribute`'s `is_primary` key to `1`. Upon change propagation, one would expect to see a new `pkey` link from the `Customer` class to the `address_postcode` column. Assuming, as in the previous example, that alternative rules can be executed when an alteration to a source element invalidates the original rule that transformed it, Johann and Egyed's scheme will not be able to create this link – in fact,

the change propagation will not make any changes to the target model at all. This is due to the non-localised nature of the change. Intuitively, although the `postcode Attribute` is changed, the rule which will be rerun (in this case `Primary_Primitive_Type_Attribute_To_Columns`) will only transform the `Attribute` itself; any new primary key links it created will be discarded as the transformation will be unaware that the link needs to be considered in an outer context. In other words, although the primary key link will be created, since the transformation rule which transforms classes to tables is not rerun, it will not be incorporated into the transformed table. In general, since the appropriate outer context that needs to be considered may be an arbitrary number of levels away from the element changed, and since the appropriate context can not be determined in advance, rerunning only part of the transformation can never be guaranteed to propagate all changes correctly.

It is left as an exercise to the reader to spot other cases in this example which will similarly foil a change propagation scheme only capable of propagating localised changes. As the examples of this subsection have demonstrated, such schemes have a fundamental weakness when propagating such changes. In the following section, I demonstrate how PMT's more general scheme is capable of propagating such changes correctly.

6.3.2. PMT's approach

The fundamental challenge with non-localised changes is to determine the particular rules to execute given a particular alteration of the source model. This requires an analysis of all the transformation rules in a system to determine which are relevant to particular changes. In a fully declarative approach such analysis may be possible, although it may be impractical or even impossible depending on the expressive power of the approach. However in a hybrid declarative / imperative approach such as PMT's, analysis of this sort is impossible in the general case – whilst PMT's use of patterns may facilitate analysis in some cases, any use of imperative code (particularly code which calls out to Converge libraries) irreparably muddies the waters. The criteria for PMT's execution approach is thus simple: it must be capable of propagating non-localised changes successfully, and it must be capable of doing so even when it can not analyse the transformation and its rules.

PMT's execution approach thus takes the only solution which can ensure correct operation in all cases: change propagation involves a complete re-execution of the transformation. By executing the transformation from the beginning, PMT implicitly propagates even non-localised changes. The downside to this approach is that rerunning the entire transformation is not efficient. However since PMT is, by design, a batch change propagation approach (see section 6.1.3), I believe this is considerably less of a problem than it would be for an immediate change propagation approach.

The efficacy of PMT's approach is best seen by example. In order to present a meaningful compar-

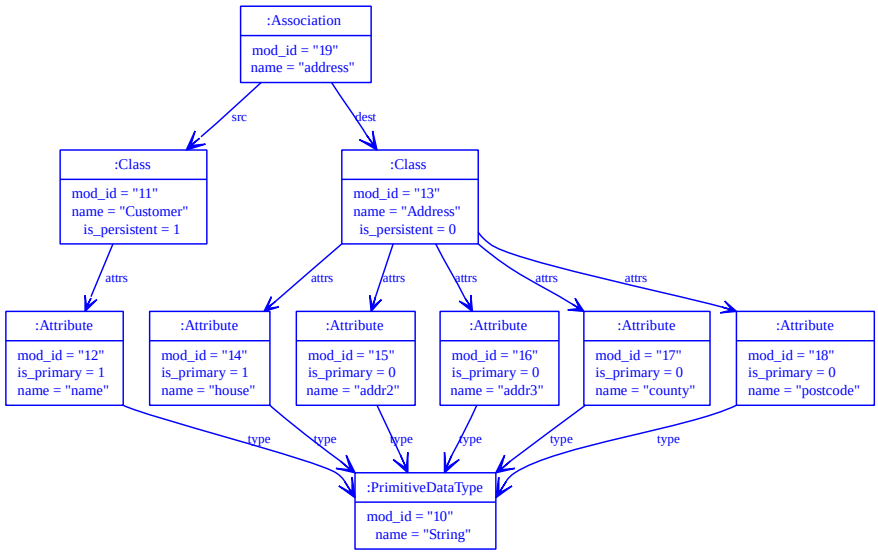


Figure 6.11.: Initial source model.

ison, I use exactly the same example as in the previous subsection. In order to have a PMT version of the advanced variant of the UML modelling language to relational database transformation from section 5.5.3, one simply needs to substitute `$<PMT.mt>` for `$<MT.mt>` in the transformation code. Although this does not lead to a particularly idiomatic PMT transformation, it saves duplicating the code, and demonstrates how close MT and PMT are in many aspects. Figure 6.11 shows the initial source model, and figure 6.12 the target model[1] created by running the `Classes_To_Tables` transformation. Figure 6.13 shows the updated source model, with the `Address` class marked as being persistent, and the `postcode` attribute marked as being part of a primary key. Figure 6.14 shows the result of change propagation on the target model.

As this example shows, PMT's change propagation approach ensures that all changes – including non-localised changes – are propagated successfully. I believe the relative inefficiency of this method is thus offset by its ability to propagate non-localised changes correctly. Section 6.6 discusses potential techniques to increase the efficiency of PMT change propagation in some circumstances.

6.4. Checking conformance operators

In some situations in a change propagating transformation, the transformation writer may wish to explicitly prevent some types of change propagation from occurring, or ensure that certain relationships between the source and target models always hold. This is potentially very important for PMT's use cases, where the transformation writer may need to constrain the modifications that the user can perform to the target model in order to ensure correct change propagation.

[1] Note that the occurrence of four '_' characters in target identifiers is the result of an implementation detail regarding the identifier of built-in Converge data types such as strings, and can be safely ignored.

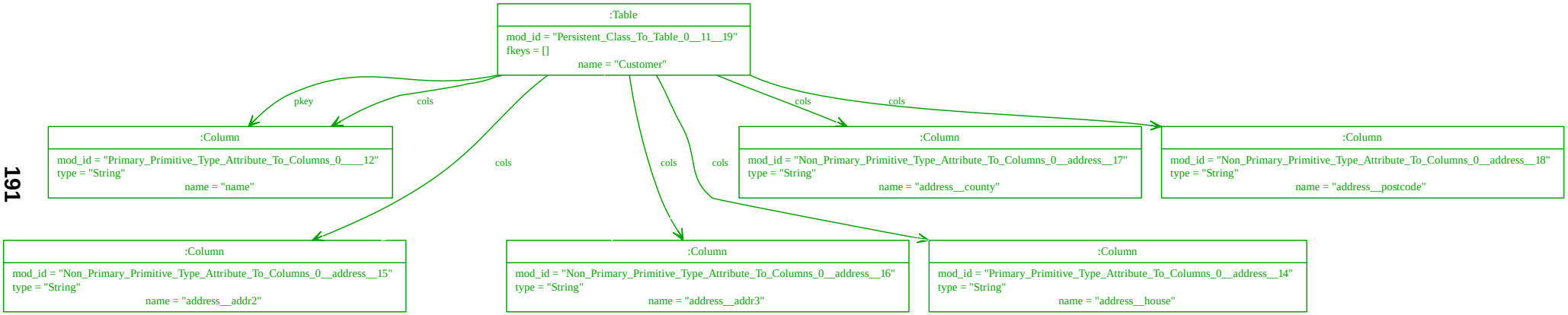


Figure 6.12.: Initial target model.

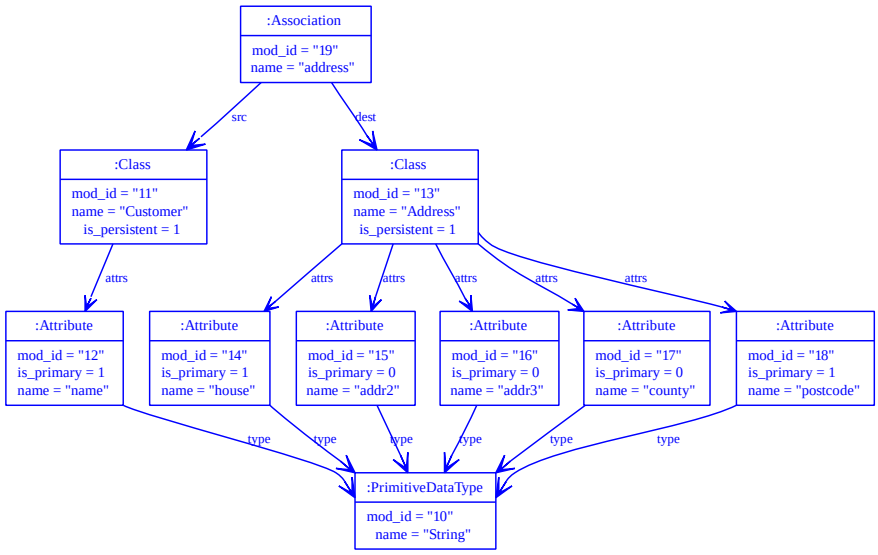


Figure 6.13.: Updated source model before change propagation.

PMT provides support for such use cases by providing *checking* conformance operators (in contrast to the updating conformance operators of section 6.2.4). By using checking conformance operators, transformation writers are able to write change propagation specifications. Note that any given model element expression may contain updating *and* checking conformance operators; change propagation specifications thus may live directly alongside change propagation implementations.

The following checking conformance operators are defined by PMT:

Operator	Name	Description
`x == y`	*equality*	Check that the value of slot `x` is equal to the value of `y`.
`x != y`	*inequality*	Check that the value of slot `x` is not equal to the value of `y`.
`x >= y`	*superset*	Check that the value of slot `x` is a non-strict superset of `y`'s value.
`x <= y`	*subset*	Check that the value of slot `x` is a non-strict subset of `y`'s value.

These operators perform the checks specified in the table, and produce a *conflict report* if the checks fail. A conflict report consists of a number of conflict records. A conflict record pinpoints a specific part of the target model as being non-conformant relative to the rule containing the failing checking conformance operator. Individual conflict records may optionally be able to show what changes would make the target model conformant. The intention of such reports is to report to the user a particular sequence of modifications which, if manually applied to the target model by the user, would make it conformant.

In order to demonstrate checking conformance operators, I once again reuse the example of section 6.2.2 replacing the `Package_To_Package` rule with the following:

```
rule Package_To_Package:
  srcp:
```

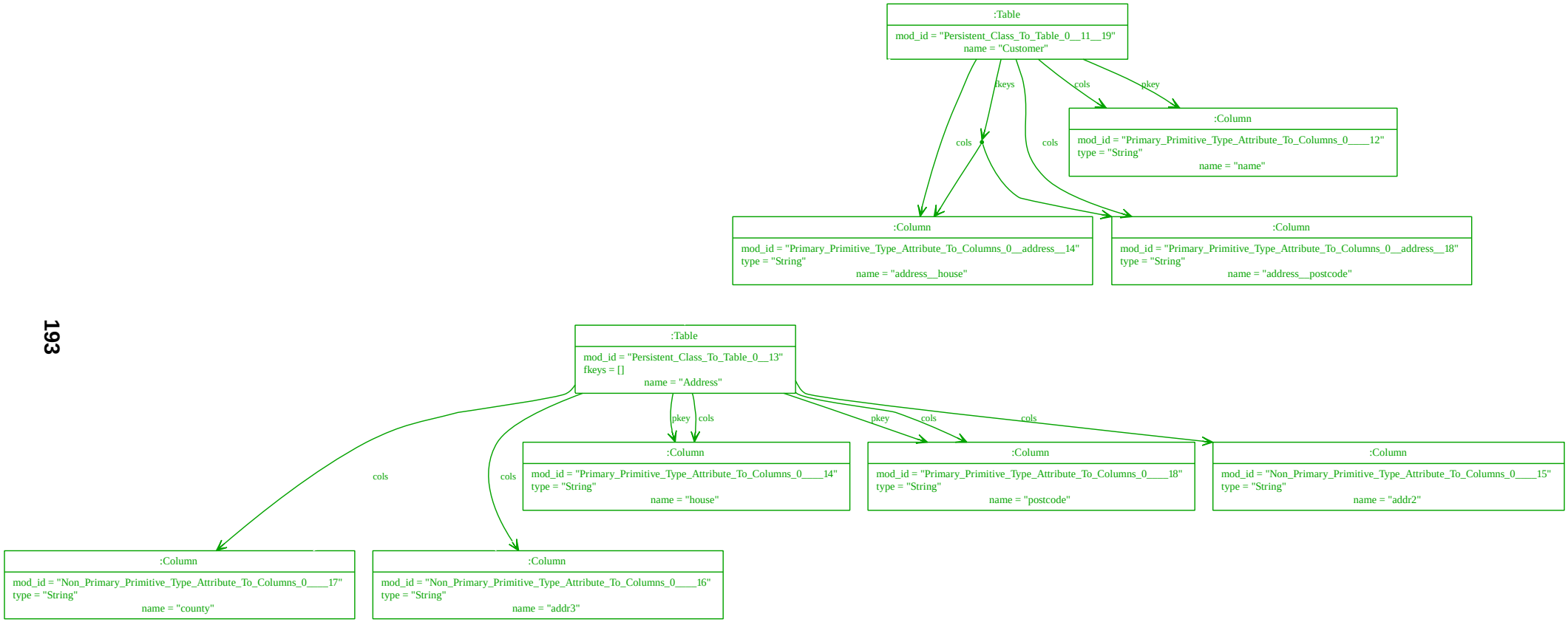


Figure 6.14.: Updated target model after change propagation.

```
    (ML2_Package)[name == <n>, elements == <elements>]

  tgtp:
    (ML1_Package)[name == n, elements >= tgt_elements]
```

Essentially this is the same rule as before, but with the updating conformance operators in the `tgtp` clauses' pattern replaced with equivalent checking conformance operators. Similarly I reuse the initial source model of figure 6.3, which leads to the creation of the same target model as figure 6.4. I then assume the user alters the target model as per figure 6.5, and the source model as per figure 6.6. When propagating changes with the new `Package_To_Package` rule in place, the result of the change propagation is shown in figure 6.15. Conflicts are clearly shown in red.

The visualization of conflicts in PMT intentionally reuses the visualization techniques from other parts of PMT, with the aim of reducing the learning burden for the user. The 'Conflict report' in figure 6.15 is analogous to the 'Tracing' report. In a similar fashion to traces, conflicts are named c*n* where *n* is an integer starting from 1. Each separate conflict is generated during a particular execution of a transformation rule. Figure 6.15 shows two types of conflicts. Conflict 'c1' shows that the `name` slot in the `Personnel` package has an incorrect value. Note that the conflict text is surrounded by a rounded box, and the link to the element is a dotted line – these visualizations only occur in conflict reports, and can not be confused with the normal visualization of elements. Conflict 'c2' shows elements missing from the `elements` slot of the `Personnel` package. Model elements, and links, in solid (as opposed to broken) red lines show that such elements need to be added to the target model in order to make it conformant. The '+' prefix is a reinforcement of this. Note that the conflict report itself denotes only that the two `ML1_Association` elements, the `ML1_Class` element and the links from the `Personnel` package to those elements, need be added to the target model. However the visualization of the conflict also shows the links between these elements (the `to` and `from` links), since these are implicitly required in order to make the target model well formed. It is important that this information is shown to the user; if it was not, then fixing a conflict report may simply result in another conflict report being generated for a part of the model just added.

Conflict reports create some interesting corner cases. To give a simple example of this, I assume a fresh execution of the `Classes_To_Tables`, once again reusing the initial source and target models of figures 6.3 and 6.4 respectively. Removing the `PE` association from the source model and executing the transformation to propagate changes leads to figure 6.16. The long dashes on the links from the `Personnel` package (combined with the '-' preceding the conflict name on the link) indicate that they should be removed from the target model in order to make it conformant. However one might have expected to see the two `ML1_Association` elements also being drawn in red dashed lines to signify their removal. However, PMT is unable to do this because although the links from the `Personnel` *should* be deleted from the target model, they are not yet deleted. Therefore

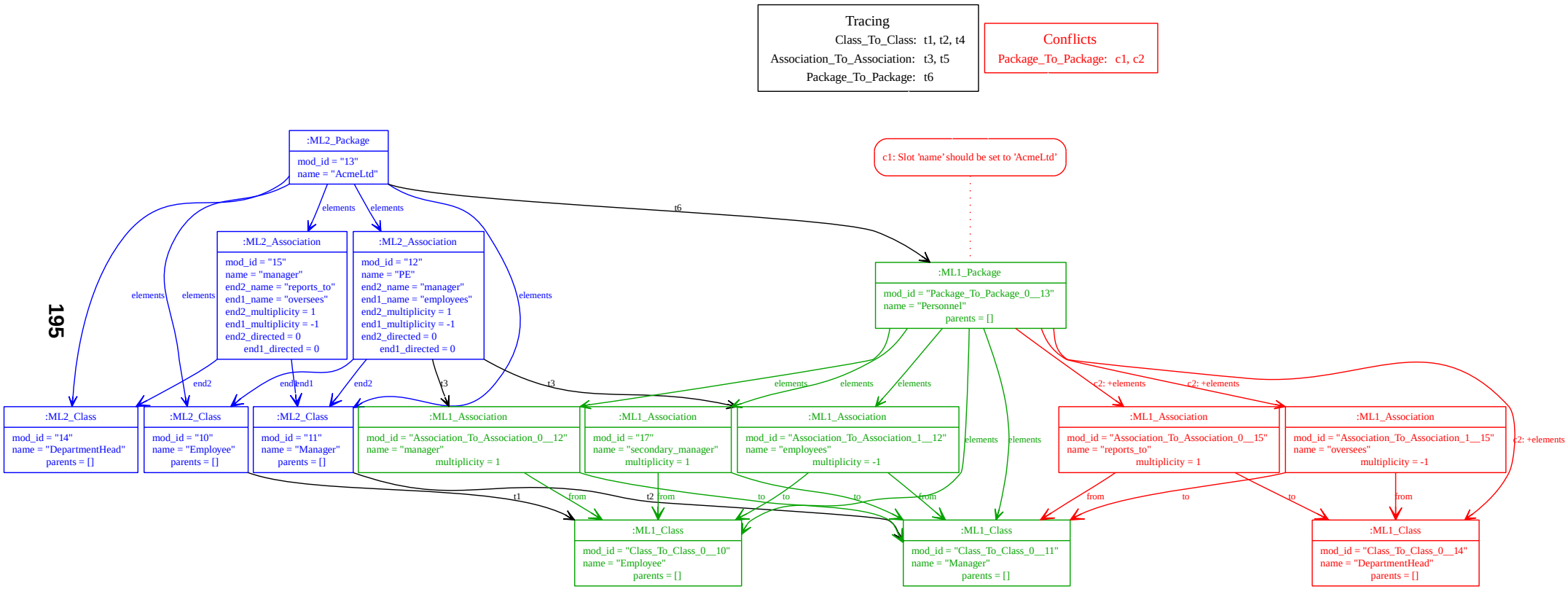


Figure 6.15.: Target model with conflicts.

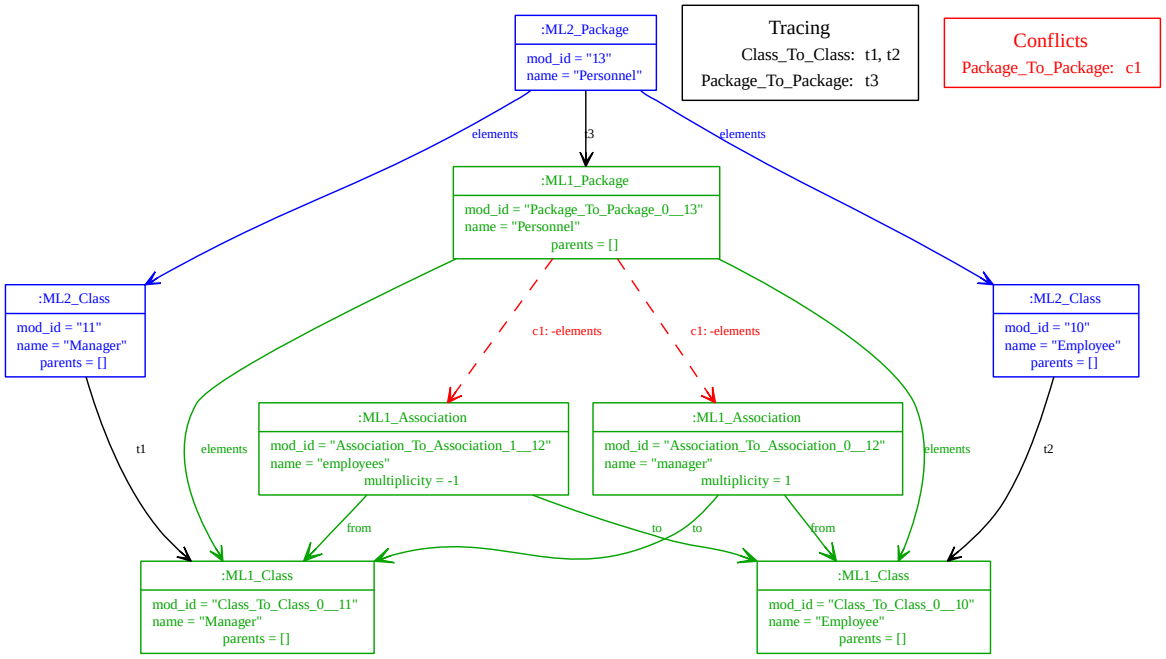


Figure 6.16.: Target model with conflicts after elements are removed from the source model.

the two `ML1_Association` elements are reachable via these links and via the garbage collection style algorithm that PMT runs at the end of the transformation (see section 6.2.6) these two elements are considered to be a valid part of the target model.

Section 6.5.2 explains the implementation of conflicts in PMT in more detail.

6.5. Implementation

Unsurprisingly, given its origins, PMT's implementation is largely similar to MT's. The majority of PMT's features are simple changes to MT code using the techniques outlined in section 5.6, and as such are not documented in detail in this section. Instead I detail two particular parts of PMT's implementation that are of additional interest over MT's implementation. PMT's grammar, which is referenced throughout this section, can be found in appendix B.2.

6.5.1. Conformance operators

A simplified version of the `_t_pt_mep_pattern` traversal function, which only contains the code for the `>=` checking conformance operator operating on unordered containers, is given below:

```
1  func _t_pt_mep_pattern(node):
2    // pt_mep_pattern ::= "(" "ID" ")" "[" "ID" pt_mep_pattern_op expr { ","
3    //                    "ID" pt_mep_pattern_op expr }* "]"
4
5    class_ := [| TM._CLASSES_REPOSITORY[$<<CEI.lift(node[2].value)>>] |]
6    conformance_operators := []
7    i := 5
8    while i < node.len() & node[i].type == "ID":
```

```
9       if node[i + 1][2].type == ">=":
10        // pt_mep_pattern_op ::= ":" ">="
11        conformance_operators.extend([|
12          val := $<<self.preorder(node[i + 2])>>
13          if Func_Binding(&obj, Object.fields["get_slot"])("_is_initialized") \
14            == 0:
15            &obj.$<<CEI.name(node[i].value)>> := val
16          elif val.conforms_to(Set):
17            should_be_in_the_set := []
18            should_not_be_in_the_set := []
19
20            for set_elem := &obj.$<<CEI.name(node[i].value)>>.iterate():
21              if not val.contains(set_elem):
22                for in_objs, out_objs := &self._old_tracing.iterate():
23                  if out_objs.contains(set_elem):
24                    should_not_be_in_the_set.append(set_elem)
25                    break
26
27            for set_elem := val.iterate():
28              if not &obj.$<<CEI.name(node[i].value)>>.contains(set_elem):
29                should_be_in_the_set.append(set_elem)
30
31            if should_be_in_the_set.len() == 0 & \
32              should_not_be_in_the_set.len() == 0:
33              pass
34            else:
35              &self._conflict_objects.append(Conflict.Set_Conflict( \
36                $<<CEI.lift(self._rule_name)>>, &matched_objs, &obj, \
37                $<<CEI.lift(node[i].value)>>, should_be_in_the_set, \
38                should_not_be_in_the_set))
39          else:
40            raise Type_Exception(Set)
41        |])
42
43    return [|
44      func () {
45
46        new_id := identifier based on rule name union of source elements etc.
47
48        if TM.OBJECTS_REPOSITORY.contains(new_id):
49          obj := TM.OBJECTS_REPOSITORY[new_id]
50        else:
51          obj := $<<class_>>.new_with_id(new_id)
52
53        $<<conformance_operators>>
54
55        return obj
56      }()
57    |]
```

There are two distinct parts to this function. Lines 43 – 55 show the core of the extended model element expressions in PMT. Line 45 calculates the identifier for the model element expression (see section 6.2.3). Line 47 then checks the TM model element repository to see whether an element with such an identifier already exists. If it does, that element is plucked from the repository (line 48). If it does not, a blank element of the correct type is created (line 50). The element, blank or otherwise, is then handed to the various conformance operators (line 52).

The superset operator is indicative of the the conformance operators in general (sections 6.2.4 and 6.4). Firstly the the model element expression is evaluated in line 12. Line 13 then checks to see whether the element has been initialized (meaning that a blank element was created in line 50); if

it has not, then the value of the user expression is simply assigned to the appropriate slot and the conformance operator automatically succeeds. If the slot does contain a value, then lines 16 – 37 check the value of the slot for conflicts against the user expression. Lines 19 – 24 check for elements in the slots value that PMT tentatively believes should not be there (see section 6.2.7), while lines 26 – 28 check for elements in the user expression which should be present in the list. If PMT detects that there are elements in the set which should or should not be there, then it generates a conflict report in lines 34 – 37.

6.5.2. Conflicts

Although conflict reports are generated by PMT, the conflict concept is housed within TM since it needs to understand conflicts in order to be able to visualize them. TM defines a simple model of conflicts which is used to record the required information. Although the model of conflicts is largely an internal detail to PMT and TM, the model presented in this subsection captures the required information in a simple manner; I hope that as other types of conflict reports are needed, it serves as a practical and efficient base for expansion.

TM currently defines three types of conflict records: slot conflicts, list conflicts, and set conflicts. Conflict records conform to the model of figure 6.17. As this shows, all conflict records share certain things in common. All conflicts are generated from a particular rule (captured by the `rule_name` slot), are the result of transforming one or more source elements (the `src_objs` association), and are specific to a particular `slot_name` within a give target element (the `tgt_obj` association).

Slot conflicts show when a slot with a primitive type (e.g. strings or ints) has an incorrect value. In such a case, the `conflict_obj` records the value the slot should have. List and set conflicts can be considered together, since they store highly similar information. In each case they record zero or more elements which should be in the given container, and zero or more elements which should not be in the container. As explained in section 6.2.7, at the time a conflict record is generated the list of elements which should not be in the container is only tentative; PMT and TM currently record all such elements, but dynamically filter them out when required to display conflict information.

6.6. Future work

Given its inherently experimental nature, PMT raises many questions and challenges for further work. As part of this, several engineering issues will need to be addressed before real-world usage is a possibility. Such issues include devising a practical mechanism for creating target identifiers that is more robust than the current string concatenation method, and so on. However I believe that once

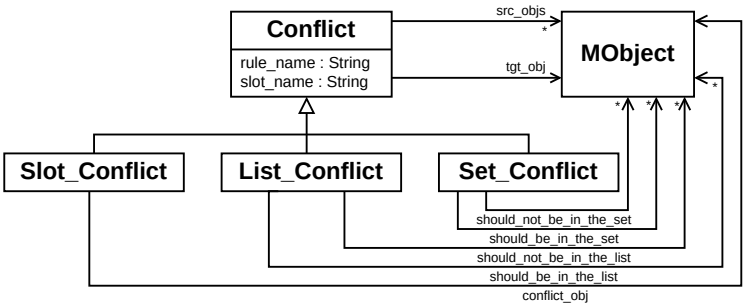


Figure 6.17.: Conflict report model.

engineering issues are put to one side, two higher-level challenges are of particular interest.

The first is a relatively short term goal. PMT's approach to removing extraneous elements from the target model is often effective, but fails to remove elements if the links to those elements have existed for more than one round of change propagation. Since PMT uses the tracing information of the previous execution, if an element survives being removed in more than one round of change propagation, then PMT incorrectly assumes it has been manually added to the target model by the user. PMT can also, in some rarer cases, erroneously delete manually added links from the target model. Finding a practical means of accurately determining which elements can be safely removed from the target model would considerably improve the overall user experience of change propagation in PMT.

The second challenge I would consider to be a longer term goal, and relates to the efficiency of the approach. As explained in section 6.3.2, change propagation in PMT involves executing the whole transformation from the beginning. Whilst has the advantage that it can propagate even non-localised changes correctly, it is inevitably somewhat slow. On the other hand, approaches like Johann and Egyed optimise change propagation, but at the considerable expense of correctness. I believe that PMT's approach is a necessary 'fall back' option, but that there are two ways that may allow PMT to execute only a subset of the transformation in some cases. The first mechanism is directly influenced by Johann and Egyed. It may be possible to perform detailed analysis of some transformation rules, since model element patterns and model element expressions not containing arbitrary Converge code are effectively declarative statements relating two models. In such cases, it may then be possible to use this knowledge to determine that certain small changes only affect certain rules. The second mechanism may be complementary to the first: often the user will know whether certain of their transformations will be involved in the propagation of certain changes. If the user knows that certain types of changes are the ones most frequently propagated, they may be willing to 'mark up' parts of the transformation to indicate that certain paths need not be taken or, alternatively, that certain paths must be taken, in the context of specific changes. I believe that working out appropriate analyses, and

also practical mechanisms for ‘marking up’ a transformation for change propagation are considerable, but highly worthwhile, challenges.

6.7. Summary

In this chapter I presented the PMT change propagating model transformation language. I started the chapter by examining in more depth some of the issues, and design decisions, facing any change propagating model transformation approach. The motivating use case for PMT – allowing the user to manually alter the target model, whilst still allowing changes to be propagated into the altered model non-destructively – is important in understanding several of PMT’s design decisions. I then presented PMT itself, exploring its approach to change propagation by example. PMT was shown to be capable of propagating even non-localised changes correctly. This led to an identification of some areas where PMT’s change propagation techniques were effective, and some areas where they fell short of what one may wish for.

Despite its immaturity – particularly in comparison to MT upon which it is based – I believe that PMT is among the very first change propagating model transformation approaches to make a genuine attempt at exploring techniques for facilitating likely real-world scenarios. Although it can by no means be considered to be production ready in its current form, I believe it provides a basis for further exploration of this challenging and exciting area.

Chapter 7.

Conclusions

7.1. Summary

In this thesis I presented a clear identification of the significant types of model transformations. I then described the Converge programming language, a dynamic OO programming language, with compile-time meta-programming. Converge integrates of a number of hitherto largely separate paradigms – Python's dynamicity, Icon's generators and backtracking, ObjVLisp's data model, and Template Haskell's compile-time meta-programming – into a coherent whole. In so doing, I listed a number of insights into the design decisions necessary to integrate such features into similar languages. Converge's compile-time meta-programming facility also contains several innovative features, such as the ability to control error reporting via nested quasi-quoting. Compile-time meta-programming was then used to provide a syntax extension facility in Converge, allowing DSLs to be directly embedded within Converge code. As a simple demonstration of this, I presented a simple DSL for defining typed modelling languages.

Converge was then used to implement the model transformation language MT. MT can be seen in several ways as an evolution of the QVT-Partners model transformation approach. To demonstrate this, I presented an in-depth analysis of the QVT-Partners approach, identifying a number of flaws and limitations. MT was presented as solving many of these problems, as well as providing useful new features such as pattern multiplicities. By integrating such features into a model transformation approach, I was able to express relatively powerful model transformations concisely. MT is also notable because of its implementation as a Converge DSL. To the best of my knowledge, it is the first model transformation approach to use a non-specialised programming language (Converge) to augment the model transformation language. Through example it was seen that MT's integration of imperative and declarative features provided a coherent model transformation environment. I also believe MT is the first model transformation approach to present a detailed description of its implementation. By making use of Converge features such as generators and backtracking, MT's implementation is small

enough to be documented in the confines of a thesis. MT is also unique in several other areas e.g. its detailed visualizations of transformations, and tracing information creation techniques.

PMT was then built as an extension of MT. PMT is, by its very nature, much more of an experiment than MT. By using a number of simple techniques (e.g. its mechanism for creating target element identifiers) PMT provides a syntactically simple language capable of expressing both change propagating transformation implementations and specifications. Although several documents talk about change propagating approaches of various forms, few appear to have a corresponding implementation. PMT is thus one of the very first concrete instances of a change propagating model transformation approach. I also believe that it is the first approach to identify many of the fundamental issues in change propagation, and the first to provide partial (although not complete) solutions to some of them. For example, PMT provides a real solution to the problem of non-localised changes, which has been otherwise ignored in the slim literature on this area. PMT's concept, and visualization, of conflicts is novel in this context, and an important step towards making such transformations usable.

7.2. Conclusions

The three main parts of this thesis – the Converge programming language, and the MT and PMT model transformation approaches – form a natural pyramid, with Converge at the bottom, MT in the middle, and PMT at the top.

Looking at the pyramid from an evolutionary perspective, one can clearly see why the order of the layers is important. Converge is a stand alone technology which facilitates the development of DSLs. Once stable, Converge was used to design and implement the MT model transformation approach. Practically speaking, MT could not have been conceived without Converge. Although this thesis documents the final design of MT, it went through many wild varying iterations before arriving at that point. Converge's low-burden development environment not only allowed such experimentation, but means that MT is small enough in size that it can be described in the space confines of a thesis. Only when MT was finished did PMT become a realistic research goal. Building on top of MT implicitly meant that many design decisions were fixed, and that PMT needed only to focus on the novel aspects of change propagation. Despite PMT's relative immaturity compared to MT and Converge, I believe it to be the first practical approach to change propagations, and the first with a publicly available implementation.

Looking at the pyramid from a usability perspective, Converge is the most fully realised of the three parts of this thesis, both in its design and implementation. This is not surprising – if Converge was less than robust, then designing and implementing MT and PMT would have become a much harder task.

Converge has already been used for tasks other than model transformations, and is currently being evaluated and used by several international users, in both industry and academia. Relative to other model transformation approaches, MT is feature rich, and its implementation relatively robust and efficient. Whilst I do not consider MT to be anywhere near as complete as Converge, it has already proved useful and is being evaluated by a handful of international users. I hope that MT provides a good platform upon which more refined model transformation may be based. PMT, on the other hand, is realistically only useful at the research level, its design being less complete than MT, and its implementation relatively fragile.

7.2.1. Future work

In terms of language design, Converge is essentially feature complete; the only major exception to this is the syntax extension feature, which needs further research to uncover a way of more seamlessly integrating it into the main language. In terms of engineering issues, the current implementation is lacking in its library support and its efficiency. Because of Converge's applicability to many different areas, I anticipate tackling all of these issues in the very near future.

Not considering engineering issues related to efficiency and robustness are, I believe that MT contains the majority of features necessary to make it a useful, real-world model transformation approach. However there is one area in which MT – along with every other model transformation approach of which I am aware – is distinctly lacking: scalability. I believe the most important area of future work for MT will be to investigate techniques for ordering, combining, and prioritising transformation rules. This will be necessary not only for efficiency purposes, but more importantly for human comprehension. Currently transformations contain every possible transformation rule needed by the transformation; ultimately this lack of modularity leads to many of the same usability issues noted with XSLT [PBG01]. As suggested in section 5.8, I believe that analysing work on combinators in functional languages may lead to new insights on how to better structure transformations. I further believe that such analysis will be applicable to many model transformation approaches, and not just MT.

As befits the most experimental, and least mature, of the three major parts of this thesis, PMT offers countless opportunities for further research. Whilst there are many small to medium sized issues – e.g. examining better approaches to creating and storing target element identifiers – I believe there are two major issues which need to be addressed before a PMT-esque technology could be considered fit for real world use. Firstly, PMT's approach to removing extraneous elements from the target model is often effective, but fails to remove elements in some cases; in one unpleasant corner case, links can be erroneously removed from the target model. Finding a practical means of accurately

determining which elements can be safely removed from the target model is likely to have a profound effect on the user experience of such technologies. Secondly, PMT is currently very inefficient. This is largely inherent, given PMT's approach to propagating non-localised changes. However, in some instances – particularly for simple, localised changes such as changing the name of a class in a source model – PMT may, possibly with some help from the user, be able to analyse transformations and determine that only a small part of the transformation need be rerun. If such cases can be determined and optimised, then change propagation may become a much more appealing prospect from a users perspective.

Appendix A.

Converge grammar

This section lists the CPK grammar for Converge. This is extracted directly from the Converge compiler file `Compiler/CV_Parser.cv`:

```
top_level ::= definition { "NEWLINE" definition }*
          ::=

definition  ::= class_def
            ::= func_def
            ::= import
            ::= var { "," var }* ":=" expr
            ::= splice

import      ::= "IMPORT" dotted_name import_as { "," dotted_name import_as }*
dotted_name ::= "ID" { "." "ID" }*
import_as   ::= "AS" "ID"
            ::=

class_def       ::= "CLASS" class_name class_supers class_metaclass ":" "INDENT"
                    class_fields "DEDENT"
class_name      ::= "ID"
                ::= splice
class_supers    ::= "(" expr { "," expr }* ")"
                ::=
class_metaclass ::= "METACLASS" expr
                ::=
class_fields    ::= class_field { "NEWLINE" class_field }*
class_field     ::= class_def
                ::= func_def
                ::= var ":=" expr
                ::= splice
                ::= "PASS"

func_def           ::= func_type func_name "(" func_params ")" ":" "INDENT"
                       func_nonlocals expr_body "DEDENT"
                   ::= func_type func_name "(" func_params ")" "{" "INDENT"
                       func_nonlocals expr_body "DEDENT" "NEWLINE" "}"
func_type          ::= "FUNC"
                   ::= "BOUND_FUNC"
                   ::= "UNBOUND_FUNC"
func_name          ::= "ID"
                   ::= "+"
                   ::= "-"
                   ::= "/"
                   ::= "*"
                   ::= "<"
                   ::= ">"
                   ::= "=="
                   ::= "!="
                   ::= ">="
```

```
                  ::= "<="
                  ::= splice
                  ::=
func_params       ::= func_params_elems "," func_varargs
                  ::= func_params_elems
                  ::= func_varargs
                  ::=
func_params_elems ::= var func_param_default { "," var func_param_default }*
                  ::= splice
func_param_default ::= ":=" expr
                  ::=
func_varargs      ::= "*" var
                  ::= splice
func_nonlocals    ::= "NONLOCAL" "ID" { "," "ID" }* "NEWLINE"
                  ::=

expr_body ::= expr { "NEWLINE" expr }*

expr ::= class_def
     ::= func_def
     ::= while
     ::= if
     ::= for
     ::= try
     ::= number
     ::= var
     ::= dict
     ::= set
     ::= list
     ::= dict
     ::= string
     ::= slot_lookup     %precedence 50
     ::= list
     ::= application     %precedence 40
     ::= lookup          %precedence 40
     ::= slice           %precedence 40
     ::= exbi
     ::= return
     ::= yield
     ::= raise
     ::= assert
     ::= break
     ::= continue
     ::= conjunction     %precedence 10
     ::= alternation     %precedence 10
     ::= assignment      %precedence 15
     ::= not             %precedence 17
     ::= neg             %precedence 35
     ::= binary          %precedence 30
     ::= comparison      %precedence 20
     ::= pass
     ::= import
     ::= splice          %precedence 100
     ::= quasi_quotes
     ::= brackets

if      ::= "IF" expr ":" "INDENT" expr_body "DEDENT" { if_elif }* if_else
        ::= "IF" expr "{" "INDENT" expr_body "DEDENT" "NEWLINE" "}" { if_elif }*
            if_else
if_elif ::= "NEWLINE" "ELIF" expr ":" "INDENT" expr_body "DEDENT"
        ::= "NEWLINE" "ELIF" expr "{" "INDENT" expr_body "DEDENT" "NEWLINE" "}"
if_else ::= "NEWLINE" "ELSE" ":" "INDENT" expr_body "DEDENT"
        ::= "NEWLINE" "ELSE" "{" "INDENT" expr_body "DEDENT" "NEWLINE" "}"
        ::=

while ::= "WHILE" expr ":" "INDENT" expr_body "DEDENT" exhausted broken
      ::= "WHILE" expr

for ::= "FOR" expr ":" "INDENT" expr_body "DEDENT" exhausted broken
```

```
    ::= "FOR" expr

try           ::= "TRY" ":" "INDENT" expr_body "DEDENT" { try_catch }* try_else
try_catch     ::= "NEWLINE" "CATCH" expr try_catch_var ":" "INDENT" expr_body
                  "DEDENT"
try_catch_var ::= "INTO" var
              ::=
try_else      ::= "NEWLINE" "ELSE" ":" "INDENT" expr_body "DEDENT"
              ::=

exhausted ::= "NEWLINE" "EXHAUSTED" ":" "INDENT" expr_body "DEDENT"
          ::=

broken ::= "NEWLINE" "BROKEN" ":" "INDENT" expr_body "DEDENT"
       ::=

number ::= "INT"

var ::= "ID"
    ::= "&" "ID"
    ::= splice

string ::= "STRING"

slot_lookup ::= expr "." "ID"
            ::= expr "." splice

list ::= "[" expr { "," expr }* "]"
     ::= "[" "]"

dict ::= "DICT{" expr ":" expr { "," expr ":" expr }* "}"
     ::= "DICT{" "}"

set ::= "SET{" expr { "," expr }* "}"
    ::= "SET{" "}"

application ::= expr "(" expr { "," expr }* ")"
            ::= expr "(" ")"

lookup ::= expr "[" expr "]"

slice ::= expr "[" expr ":" expr "]"
      ::= expr "[" ":" expr "]"
      ::= expr "[" expr ":" "]"
      ::= expr "[" ":" "]"

exbi ::= "EXBI" expr "." "ID"

return ::= "RETURN" expr
       ::= "RETURN"

yield ::= "YIELD" expr

raise ::= "RAISE" expr

assert ::= "ASSERT" expr

break ::= "BREAK"

continue ::= "CONTINUE"

conjunction ::= expr "&" expr { "&" expr }*

alternation ::= expr "|" expr { "|" expr }*

assignment        ::= assignment_target { "," assignment_target }*
                      assignment_type expr
assignment_target ::= var
                  ::= slot_lookup
```

```
                  ::= lookup
                  ::= slice
assignment_type   ::= ":="
                  ::= "*="
                  ::= "/="
                  ::= "+="
                  ::= "-="

not ::= "NOT" expr

neg ::= "-" expr

binary    ::= expr binary_op expr
binary_op ::= "*"       %precedence 40
          ::= "/"       %precedence 30
          ::= "%"       %precedence 30
          ::= "+"       %precedence 20
          ::= "-"       %precedence 20

comparison    ::= expr comparison_op expr
comparison_op ::= "IS"
              ::= "=="
              ::= "!="
              ::= "<="
              ::= ">="
              ::= "<"
              ::= ">"

pass ::= "PASS"

splice       ::= expr_splice
             ::= block_splice
expr_splice  ::= "$" "<" "<" expr ">" ">"
block_splice ::= "$" "<" expr ">" ":" "INDENT" "JUMBO" "DEDENT"

quasi_quotes      ::= expr_quasi_quotes
                  ::= defn_quasi_quotes
expr_quasi_quotes ::= "[|" "INDENT" expr { "NEWLINE" expr }* "DEDENT"
                      "NEWLINE" "|]"
                  ::= "[|" expr { "NEWLINE" expr }* "|]"
defn_quasi_quotes ::= "[D|" definition { "NEWLINE" definition }* "|]"
                  ::= "[D|" "INDENT" definition { "NEWLINE" definition }*
                      "DEDENT" "NEWLINE" "|]"

brackets ::= "(" expr ")"
```

Appendix B.

DSL grammars

B.1. MT Grammar

```
mt_rules ::= "TRANSFORMATION" "ID" "NEWLINE" mt_rule { "NEWLINE" mt_rule }*

mt_rule ::= "RULE" "ID" ":" "INDENT" mt_in "NEWLINE" mt_out "DEDENT"

mt_in   ::= mt_inp mt_inc
mt_inp  ::= "SRCP" ":" "INDENT" pt_ipattern { "NEWLINE" pt_ipattern }* "DEDENT"
mt_inc  ::= "NEWLINE" "SRC_WHEN" ":" "INDENT" pt_ipattern "DEDENT"
        ::=

mt_tgt  ::= mt_tgtp mt_tgtw mt_tracing
mt_tgtp ::= "TGTP" ":" "INDENT" mt_tgt_expr { "NEWLINE" expr }* "DEDENT"
mt_tgtw ::= "NEWLINE" "TGT_WHERE" ":" "INDENT" expr { "NEWLINE" expr }* "DEDENT"
        ::=
mt_tracing ::= "NEWLINE" "TRACING_ADD" ":" "INDENT" expr "DEDENT"
           ::= "NEWLINE" "TRACING_OVERRIDE" ":" "INDENT" expr "DEDENT"
           ::=

pt_ipattern ::= pt_ipattern_expr pt_ipattern_qualifier

pt_ipattern_expr ::= pt_iobj_pattern   %precedence 10
                 ::= pt_iset_pattern   %precedence 10
                 ::= pt_ivar           %precedence 10
                 ::= expr

pt_ipattern_qualifier ::= ":" pt_multiplicity "<" "ID" ">"
                      ::=

pt_multiplicity             ::= pt_multiplicity_upper_bound
                            ::= expr "!"
                            ::= "*" "!"
                            ::= expr "." "." pt_multiplicity_upper_bound
pt_multiplicity_upper_bound ::= expr
                            ::= expr "?"
                            ::= "*"
                            ::= "*" "?"

pt_iobj_pattern            ::= "(" pt_iobj_pattern_self ")" "[" pt_iobj_slot
                               pt_iobj_pattern_comparison pt_ipattern_expr {
                 "," pt_iobj_slot pt_iobj_pattern_comparison
                               pt_ipattern_expr }* "]"
                           ::= "(" pt_iobj_pattern_self ")" "[" "]"
pt_iobj_pattern_self       ::= "ID" "," "<" "ID" ">"
                           ::= "ID"
                           ::=
pt_iobj_slot               ::= "ID"
                           ::= "ID" "(" expr { "," expr }* ")"
```

```
                            ::= "ID" "(" ")"
pt_iobj_pattern_comparison ::= "=="
                            ::= "!="
                            ::= "<"
                            ::= ">"
                            ::= "<="
                            ::= ">="
               ::= "IS"

pt_iset_pattern       ::= "Set{" pt_iset_pattern_elems "|"
                          pt_iset_pattern_elems "}"
                      ::= "Set{" pt_iset_pattern_elems "}"
pt_iset_pattern_elems ::= pt_ipattern { "," pt_ipattern }*
                      ::=

pt_ivar ::= "<" "ID" ">"

mt_tgt_expr           ::= expr mt_tgt_expr_qualifier
mt_tgt_expr_qualifier ::= "FOR" expr
                      ::=

expr ::= pt_mep_pattern

pt_mep_pattern ::= "(" "ID" ")" "[" "ID" ":=" expr { "," "ID" ":=" expr }* "]"
               ::= "(" "ID" ")" "[" "]"
```

B.2. PMT Grammar

PMT's grammar is identical to MT's with the exception of the `pt_mep_pattern` rule whose updated definition is as follows:

```
pt_mep_pattern ::= "(" "ID" ")" "[" "ID" pt_mep_pattern_op expr { "," "ID"
                   pt_mep_pattern_op expr }* "]"
               ::= "(" "ID" ")" "[" "]"
pt_mep_pattern_op ::= ":="
                  ::= "=="
                  ::= ":" "=="
                  ::= "!="
                  ::= ">="
                  ::= ":" ">="
                  ::= "<="
```

Appendix C.

Additional examples

C.1. Converting associations to foreign keys

This transformation is a simple example of a standard transformation: removing associations from a model, replacing them with foreign key attributes in the appropriate classes. This is done in the context of a simple UML-esque modelling language which allows attributes to be marked as constituting part of a primary key or not. The metamodel is shown in figure C.1. When an association is replaced by attributes, the name of the target class is prepended to attribute names to aid uniqueness of the resultant names, and also as a grouping mechanism.

The full transformation module is as follows:

```
import Sys
import MT.MT
import TM.Visualizer
import Simple_UML


$<MT.mt>:

  transformation Associations_To_Foreign_Keys
  rule Class_To_Class:
    srcp:
      (Class, <c>)[name == <name>, attrs == <attrs>]
      (Association)[src == c, dest == <dest>] : * <assocs>

    tgtp:
      (Class)[name := name, attrs := self.transform_all(attrs) + new_attrs]

    tgt_where:
      new_attrs := Set{}
      for dict := assocs.iterate():
        for attr := dict["dest"].attrs.iterate():
          if attr.is_primary:
            new_attr := (Attribute)[name := dict["dest"].name + "_" + \
              attr.name, type := self.transform([attr.type]), \
              is_primary := 0]
            new_attrs.add(new_attr)

  rule Remove_Association:
    srcp:
      (Association, <a>)[]

    tgtp:
```

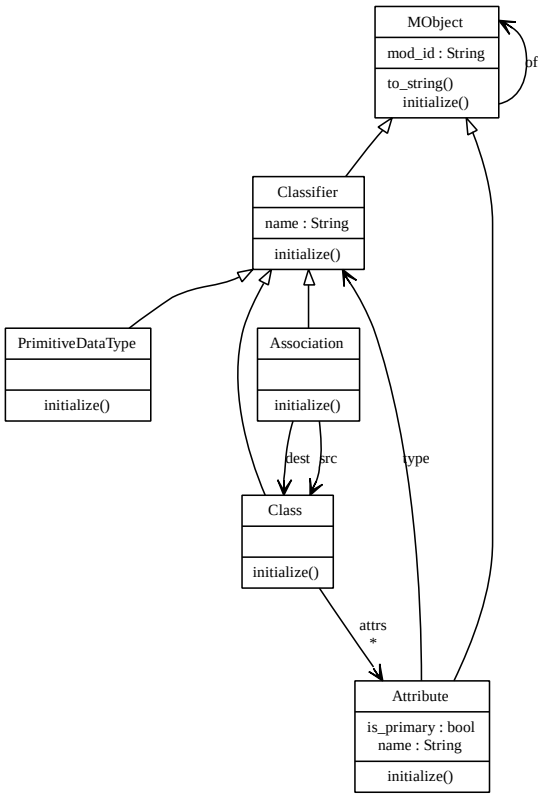


Figure C.1.: Simple UML modelling language, with primary key support.

```
      null

  rule Default:
    srcp:
      (MObject, <mo>)[]

    tgtp:
      self.clone_and_transform(mo)


func main():

  customer := Simple_UML.Class("Customer", Set{Simple_UML.Attribute("name", \
    Simple_UML.String, 1)})
  order := Simple_UML.Class("Order", Set{Simple_UML.Attribute("order_no", \
    Simple_UML.Integer, 1)})
  employee := Simple_UML.Class("Employee", Set{Simple_UML.Attribute("name", \
    Simple_UML.String, 1), Simple_UML.Attribute("age", Simple_UML.Integer, 1)})

  customer_order := Simple_UML.Association("order", customer, order)
  order_employee := Simple_UML.Association("fulfilled_by", order, employee)

  class_model := [customer, order, employee, customer_order, order_employee]

  Visualizer.visualize_model(class_model, [], fail)

  transformation := Associations_To_Foreign_Keys.new.apply(class_model)
  transformed := transformation.get_target()

  Visualizer.visualize_model(transformed, [], 1)
```

An example of the input to this transformation can be seen in figure C.2, and the result of the

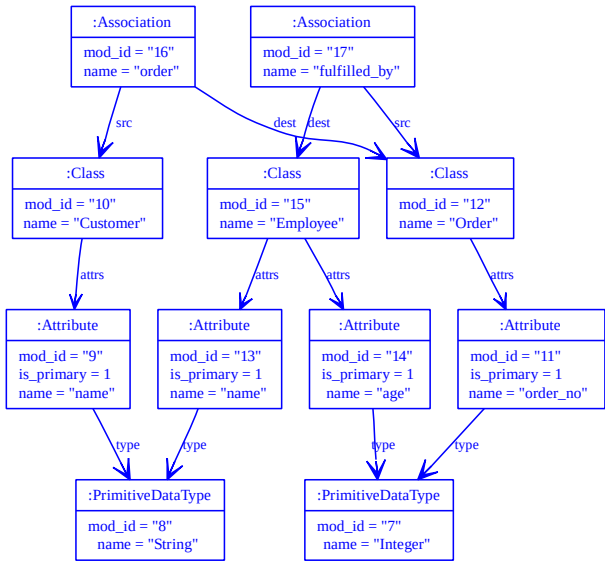


Figure C.2.: ER source model.

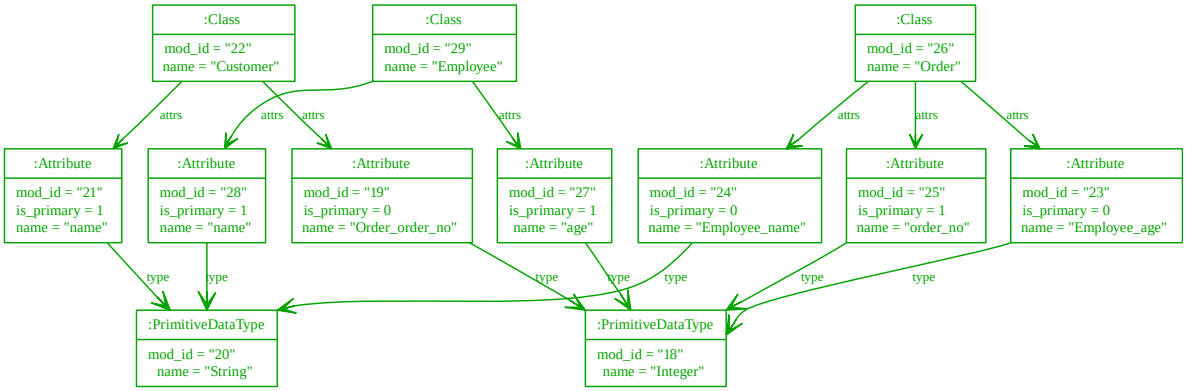


Figure C.3.: ER target model.

transformation seen in figure C.3.

C.2. Removing ‘many to many’ relations

This transformation is a simple example of a standard transformation: removing many to many associations in an Entity-Relationship diagram. Figure C.4 shows a simple meta-model of ER diagrams — this example is only concerned with entities and relationships.

The full transformation module is as follows:

```
import Sys
import PMT.PMT
import TM.Visualizer
import ER

$<PMT.mt>:
  transformation Remove_Many_To_Many_Relations

  rule Statemachine_To_Statemachine:
    srcp:
```

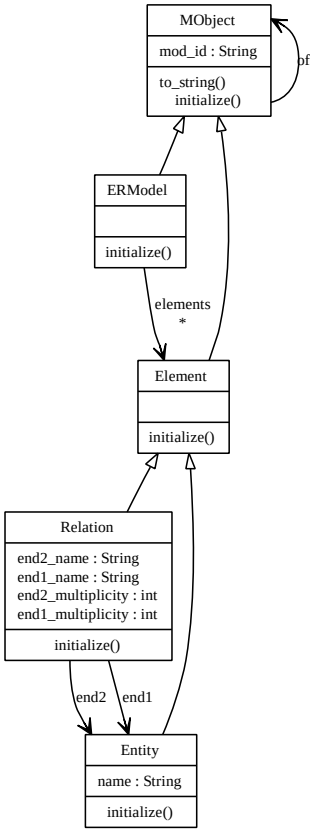


Figure C.4.: ER diagram metamodel.

```
    (ERModel)[elements == <ielements>]

  tgtp:
    (ERModel)[elements := oelements]

  tgt_where:
    oelements := []
    for element := ielements.iterate():
      oelements.extend(self.transform([element]))


rule Many_To_Many_Association:
  srcp:
    (Relation)[end1_multiplicity == -1, end2_multiplicity == -1, \
      end1_name == <iend1_name>, end2_name == <iend2_name>, end1 == \
      <iend1>, end2 == <iend2>]

  tgtp:
    (Relation)[end1_multiplicity := 1, end2_multiplicity := -1, \
      end1 := self.transform([iend1]), end2 := intermediate_data, \
      end1_name := iend1_name, end2_name := intermediate_name]
    (Relation)[end1_multiplicity := -1, end2_multiplicity := 1, \
      end1 := intermediate_data, end2 := self.transform([iend2]), \
      end1_name := intermediate_name, end2_name := iend2_name]

  tgt_where:
    intermediate_name := iend1_name + "_" + iend2_name
    intermediate_data := (Entity)[name := intermediate_name]


rule Default:
  srcp:
    (MObject, <mo>)[]
```

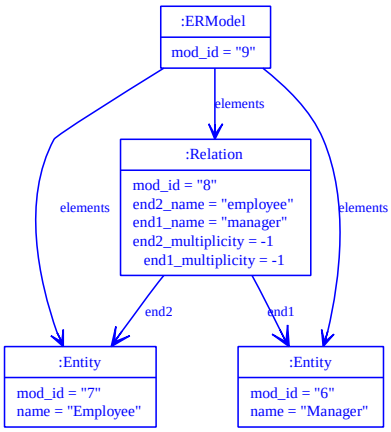


Figure C.5.: ER source model.

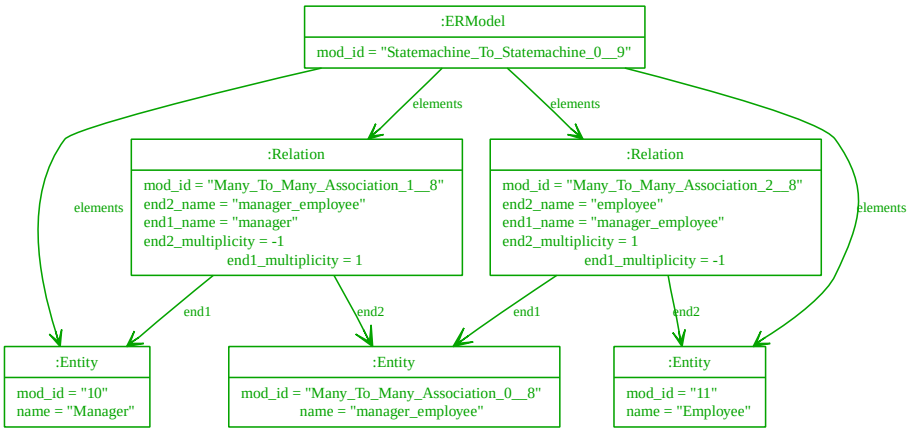


Figure C.6.: ER target model.

```
    tgtp:
      [self.clone_and_transform(mo)]

    tgt_where:
      Sys.println("Default")


func main():

  manager := ER.Entity("Manager")
  employee := ER.Entity("Employee")
  manager_employee := ER.Relation(-1, -1, "manager", "employee", manager, \
    employee)

  ermodel := ER.ERModel(Set{manager, employee, manager_employee})

  Visualizer.visualize_model(ermodel, [], fail)

  transformation := Remove_Many_To_Many_Relations(ermodel)
  transformation.do_transform()
  transformed := transformation.get_output()

  Visualizer.visualize_model(transformed, [], 1)
```

An example of the input to this transformation can be seen in figure C.5, and the result of the transformation seen in figure C.6.

Appendix D.

Example translations

D.1. The ‘Simple UML’ modelling language

This section shows the pretty printed ITree that results from translating the Simple UML modelling language shown in section 4.5.1. Note that this particular translation is slightly naïve in nature – it would be possible to engineer a modelling DSL that would cause variable capture (e.g. a model attribute called `for_expr` would lead to unpredictable results). As the more sophisticated translation of the model transformation language shows, such problems can be avoided, albeit at the cost of increased implementation effort.

```
$$1$$ := bound_func initialize_Classifier(*args){
  super_attrs := TM._all_attrs(TM.MObject, 1)
  if args.len() > super_attrs.len() + 1:
    raise TM.Exceptions.Parameters_Exception("Too many args")
  super_args_pos := TM.Maths.min(super_attrs.len(), args.len())
  TM.Func_Binding(self, TM.MObject.methods["initialize"]).apply(args[0 : \
    super_args_pos])
  if 0 < args.len() - super_args_pos:
    self.name := args[super_args_pos + 0]
}

Classifier := TM.MClass(1, "Classifier", TM.MObject, Dict{"name" : [3]}, \
  ["name"], Dict{"initialize" : $$1$$}, [])

$$2$$ := bound_func initialize_PrimitiveDataType(*args){
  super_attrs := TM._all_attrs(TM._CLASSES_REPOSITORY["Classifier"], 1)
  if args.len() > super_attrs.len() + 0:
    raise TM.Exceptions.Parameters_Exception("Too many args")
  super_args_pos := TM.Maths.min(super_attrs.len(), args.len())
  TM.Func_Binding(self, TM._CLASSES_REPOSITORY["Classifier"]. \
    methods["initialize"]).apply(args[0 : super_args_pos])
}

PrimitiveDataType := TM.MClass(0, "PrimitiveDataType",
  TM._CLASSES_REPOSITORY["Classifier"], Dict{}, [], Dict{"initialize" : \
    $$2$$}, [])

$$3$$ := bound_func initialize_Class(*args){
  super_attrs := TM._all_attrs(TM._CLASSES_REPOSITORY["Classifier"], 1)
  if args.len() > super_attrs.len() + 2:
    raise TM.Exceptions.Parameters_Exception("Too many args")
  super_args_pos := TM.Maths.min(super_attrs.len(), args.len())
  TM.Func_Binding(self, TM._CLASSES_REPOSITORY["Classifier"]. \
    methods["initialize"]).apply(args[0 : super_args_pos])
```

```
  if 0 < args.len() - super_args_pos:
    self.parents := args[super_args_pos + 0]
  if 0 >= args.len() - super_args_pos:
    self.parents := TM.TM_List(self)
  if 1 < args.len() - super_args_pos:
    self.attrs := args[super_args_pos + 1]
  if 1 >= args.len() - super_args_pos:
    self.attrs := TM.TM_List(self)
}

Class := TM.MClass(0, "Class", TM._CLASSES_REPOSITORY["Classifier"],
  Dict{"parents" : [0, "Class"], "attrs" : [0, "Attribute"]}, ["parents", \
  "attrs"], Dict{"initialize" : $$3$$}, [["unique_names", unbound_func \
  unique_names(self){
  return unbound_func ocl_for(){
    for_expr := self.attrs
    for a1 := for_expr.iterate():
      for a2 := for_expr.iterate():
        if not unbound_func ocl_implies(){
          if not unbound_func ocl_not_equals(){
            lhs := a1
            if lhs.conforms_to(TM.Int) | lhs.conforms_to(TM.String):
              return lhs != a2
            else:
              return not lhs is a2
          }():
            return 1
          if unbound_func ocl_not_equals(){
            lhs := a1.name
            if lhs.conforms_to(TM.Int) | lhs.conforms_to(TM.String):
              return lhs != a2.name
            else:
              return not lhs is a2.name
          }():
            return 1
          return TM.fail
        }():
          return TM.fail
    return 1
  }()
}]])

$$4$$ := bound_func initialize_Attribute(*args){
  super_attrs := TM._all_attrs(TM.MObject, 1)
  if args.len() > super_attrs.len() + 3:
    raise TM.Exceptions.Parameters_Exception("Too many args")
  super_args_pos := TM.Maths.min(super_attrs.len(), args.len())
  TM.Func_Binding(self, TM.MObject.methods["initialize"]).apply( \
    args[0 : super_args_pos])
  if 0 < args.len() - super_args_pos:
    self.name := args[super_args_pos + 0]
  if 1 < args.len() - super_args_pos:
    self.type := args[super_args_pos + 1]
  if 2 < args.len() - super_args_pos:
    self.is_primary := args[super_args_pos + 2]
}

Attribute := TM.MClass(0, "Attribute", TM.MObject, Dict{"name" : [3], \
  "type" : "Classifier", "is_primary" : [5]}, ["name", "type", "is_primary"], \
  Dict{"initialize" : $$4$$}, [])
```

D.2. Simple classes to tables transformation

```
Classes_To_Tables := class Classes_To_Tables:
  _rule_names := ["Class_To_Table", "User_Type_Attr_To_Column", \
    "Primitive_Type_Attr_To_Column"]
```

```
bound_func init(*root_set){
  self._root_set := root_set
  self._transformed_cache := Dict{}
  self._matched_objs := Set{}
  self._tracing := []
  self._tracing_rule := []
  for rule_name := self._rule_names.iterate():
    self._transformed_cache[rule_name] := Dict{}
  self._output := self.transform_all(root_set)
}
bound_func get_source(){
  return self._root_set
}
bound_func get_target(){
  return self._output
}
bound_func get_conflict_objects(){
  return []
}
bound_func transform(*objs){
  for obj := objs.iterate():
    if not obj.conforms_to(MT.List):
      raise MT.Exceptions.Type_Exception(MT.List, obj.instance_of, \
        obj.to_str())
  for rule_name := self._rule_names.iterate():
    if output := self.get_slot(rule_name).apply(objs):
      return output
  raise MT.Exceptions.Exception(MT.Strings.format( \
    "Unable to transform '%s'.", objs.to_str()))
}
bound_func transform_all(objs){
  if objs.conforms_to(MT.List):
    output_objs := []
    for obj := objs.iterate():
      output_objs.append(self.transform([obj]))
  elif objs.conforms_to(MT.Set):
    output_objs := Set{}
    for obj := objs.iterate():
      output_objs.add(self.transform([obj]))
  else:
    raise MT.Exceptions.Exception(objs.instance_of.name)
  return output_objs
}
bound_func Class_To_Table(*objs){
  $$18$$self$$ := self
  if matched_objs, bindings := unbound_func (*args){
    if args.len() > 1:
      return Input_Pattern_Creator.fail
    matched_mp_elems := Set{}
    bindings := Dict{}
    if 0 < args.len():
      $$7$$elements$$ := args[0]
    else:
      $$7$$elements$$ := self._root_set
    $$8$$matched_mp_elems_backup$$ := matched_mp_elems
    $$9$$bindings_backup$$ := bindings
    for $$10$$new_matched_mp_elems$$, $$11$$new_bindings$$, \
      $$12$$matched_elem$$ := unbound_func (bindings, elements){
      for element := elements.iterate() & yield unbound_func (bindings, \
        element){
        if not Input_Pattern_Creator.TM.type_match("Class", element):
          return Input_Pattern_Creator.fail
        for yield $$2$$ := unbound_func (bindings, element){
          if bindings.contains("c") & not bindings["c"] == element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"c" : element}, element]
        }(bindings, element) & $$1$$ := bindings + $$2$$[1] & \
          $$4$$ := unbound_func (bindings){
          slot_element := element.name
```

```
        for matched_mp_elems, new_bindings, \
          matched_elem := unbound_func (bindings, element){
          if bindings.contains("n") & not bindings["n"] == \
            element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"n" : element}, element]
        }(bindings, slot_element):
          if slot_element == matched_elem:
            yield [matched_mp_elems, new_bindings, \
              matched_elem]
        return Input_Pattern_Creator.fail
      }($$1$$) & $$3$$ := $$1$$ + $$4$$[1] & $$6$$ := \
        unbound_func (bindings){
        slot_element := element.attrs
        for matched_mp_elems, new_bindings, matched_elem := \
          unbound_func (bindings, element){
          if bindings.contains("A") & not bindings["A"] == \
            element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"A" : element}, element]
        }(bindings, slot_element):
          if slot_element == matched_elem:
            yield [matched_mp_elems, new_bindings, \
              matched_elem]
        return Input_Pattern_Creator.fail
      }($$3$$) & $$5$$ := $$3$$ + $$6$$[1] & [Set{element}, \
        Input_Pattern_Creator.Functional.foldl( \
          Input_Pattern_Creator._adder, \
          Input_Pattern_Creator.Functional.map( \
            Input_Pattern_Creator._element1, [$$2$$, $$4$$, \
            $$6$$])), element]
      return Input_Pattern_Creator.fail
    }(bindings, element)
    return Input_Pattern_Creator.fail
  }(bindings, $$7$$elements$$):
    matched_mp_elems := matched_mp_elems + \
      $$10$$new_matched_mp_elems$$
    bindings := bindings + $$11$$new_bindings$$
    return [matched_mp_elems, bindings]
    matched_mp_elems := $$8$$matched_mp_elems_backup$$
    bindings := $$9$$bindings_backup$$
  return Input_Pattern_Creator.fail
}.apply(objs):
  self._matched_objs.extend(matched_objs)
  if not rtn := unbound_func (matched_objs, bindings){
    concatted_id := Output_Pattern_Creator.concat_id(matched_objs)
    if self._transformed_cache["Class_To_Table"].contains( \
      concatted_id):
      return self._transformed_cache["Class_To_Table"][concatted_id]
    tracing_i := self._tracing.len()
    $$13$$c$$ := bindings["c"]
    $$14$$n$$ := bindings["n"]
    $$15$$A$$ := bindings["A"]
    $$16$$columns$$ := []
    for $$17$$attr$$ := $$15$$A$$.iterate():
      $$16$$columns$$.extend($$18$$self$$.transform([""], \
        [$$17$$attr$$]).flatten())
    out_elems := []
    out_elems.append(unbound_func (){
      user_args := Dict{"name" : $$14$$n$$, "cols" : \
        $$16$$columns$$}
      all_args := []
      args_processed := 0
      for attr_name := Output_Pattern_Creator.TM. \
        all_attrs_in_order(Output_Pattern_Creator.TM. \
          _CLASSES_REPOSITORY["Table"]).iterate():
        if args_processed == user_args.len():
          break
        if user_args.contains(attr_name):
```

```
            all_args.append(user_args[attr_name])
            args_processed += 1
          else:
            all_args.append(Output_Pattern_Creator.null)
        return Output_Pattern_Creator.TM._CLASSES_REPOSITORY \
          ["Table"].new.apply(all_args)
      }())
      out_elem := out_elems[0]
      self._transformed_cache["Class_To_Table"][concatted_id] := \
        out_elem
      if not out_elem is Output_Pattern_Creator.null:
        tracing := [Output_Pattern_Creator.List(matched_objs), \
          out_elems]
        tracing := Output_Pattern_Creator.trace_reduce(tracing)
        self._tracing.insert(tracing_i, tracing)
        self._tracing_rule.insert(tracing_i, "Class_To_Table")
      return out_elem
    }(matched_objs, bindings):
      raise MT.Exceptions.Exception(MT.Strings.format(\
        "Output pattern of Class_To_Table failed to generate " +
        "anything for '%s'.", objs.to_str()))
    return rtn
  else:
    return MT.fail
}
bound_func User_Type_Attr_To_Column(*objs){
  $$48$$concat_name$$ := concat_name
  $$49$$self$$ := self
  if matched_objs, bindings := unbound_func (*args){
    if args.len() > 2:
      return Input_Pattern_Creator.fail
    matched_mp_elems := Set{}
    bindings := Dict{}
    if 0 < args.len():
      $$37$$elements$$ := args[0]
    else:
      $$37$$elements$$ := self._root_set
    $$38$$matched_mp_elems_backup$$ := matched_mp_elems
    $$39$$bindings_backup$$ := bindings
    for $$40$$new_matched_mp_elems$$, $$41$$new_bindings$$, \
      $$42$$matched_elem$$ := unbound_func (bindings, elements){
      for element := elements.iterate() & yield \
        unbound_func (bindings, element){
        if not Input_Pattern_Creator.TM.type_match("String", element):
          return Input_Pattern_Creator.fail
        for yield $$20$$ := unbound_func (bindings, element){
          if bindings.contains("prefix") & \
            not bindings["prefix"] == element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"prefix" : element}, element]
        }(bindings, element) & $$19$$ := bindings + $$20$$[1] & \
          [Set{element}, Input_Pattern_Creator.Functional.foldl( \
            Input_Pattern_Creator._adder, \
          Input_Pattern_Creator.Functional.map( \
            Input_Pattern_Creator._element1, [$$20$$])), element]
        return Input_Pattern_Creator.fail
      }(bindings, element)
      return Input_Pattern_Creator.fail
    }(bindings, $$37$$elements$$):
      matched_mp_elems := matched_mp_elems + \
        $$40$$new_matched_mp_elems$$
      bindings := bindings + $$41$$new_bindings$$
      if 1 < args.len():
        $$31$$elements$$ := args[1]
      else:
        $$31$$elements$$ := self._root_set
      $$32$$matched_mp_elems_backup$$ := matched_mp_elems
      $$33$$bindings_backup$$ := bindings
      for $$34$$new_matched_mp_elems$$, $$35$$new_bindings$$, \
```

```
$$36$$matched_elem$$ := unbound_func (bindings, elements){
for element := elements.iterate() & yield \
  unbound_func (bindings, element){
  if not Input_Pattern_Creator.TM.type_match("Attribute", \
    element):
    return Input_Pattern_Creator.fail
  for yield $$21$$ := bindings & $$23$$ := \
    unbound_func (bindings){
    slot_element := element.name
    for matched_mp_elems, new_bindings, matched_elem := \
      unbound_func (bindings, element){
      if bindings.contains("n") & \
        not bindings["n"] == element:
        return Input_Pattern_Creator.fail
      return [Set{}, Dict{"n" : element}, element]
    }(bindings, slot_element):
      if slot_element == matched_elem:
        yield [matched_mp_elems, new_bindings, \
          matched_elem]
    return Input_Pattern_Creator.fail
  }($$21$$) & $$22$$ := $$21$$ + $$23$$[1] & $$30$$ := \
    unbound_func (bindings){
    slot_element := element.type
    for matched_mp_elems, new_bindings, matched_elem := \
      unbound_func (bindings, element){
      if not Input_Pattern_Creator.TM.type_match(\
        "Class", element):
        return Input_Pattern_Creator.fail
      for yield $$24$$ := bindings & \
        $$26$$ := unbound_func (bindings){
        slot_element := element.name
        for matched_mp_elems, new_bindings, \
          matched_elem := unbound_func (bindings, \
            element){
          if bindings.contains("cn") & \
            not bindings["cn"] == element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"cn" : element}, \
            element]
        }(bindings, slot_element):
          if slot_element == matched_elem:
            yield [matched_mp_elems, \
              new_bindings, matched_elem]
        return Input_Pattern_Creator.fail
      }($$24$$) & $$25$$ := $$24$$ + $$26$$[1] & \
        $$28$$ := unbound_func (bindings){
        slot_element := element.attrs
        for matched_mp_elems, new_bindings, \
          matched_elem := unbound_func (bindings, \
            element){
          if bindings.contains("CA") & \
            not bindings["CA"] == element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"CA" : element}, \
            element]
        }(bindings, slot_element):
          if slot_element == matched_elem:
            yield [matched_mp_elems, \
              new_bindings, matched_elem]
        return Input_Pattern_Creator.fail
      }($$25$$) & $$27$$ := $$25$$ + $$28$$[1] & \
        [Set{element}, \
        Input_Pattern_Creator.Functional.foldl( \
          Input_Pattern_Creator._adder, \
        Input_Pattern_Creator.Functional.map( \
        Input_Pattern_Creator._element1, \
        [$$26$$, $$28$$])), element]
      return Input_Pattern_Creator.fail
    }(bindings, slot_element):
```

```
              if slot_element == matched_elem:
                yield [matched_mp_elems, new_bindings, \
                  matched_elem]
            return Input_Pattern_Creator.fail
          }($$22$$) & $$29$$ := $$22$$ + $$30$$[1] & \
            [Set{element}, \
            Input_Pattern_Creator.Functional.foldl( \
            Input_Pattern_Creator._adder, \
            Input_Pattern_Creator.Functional.map( \
            Input_Pattern_Creator._element1, [$$23$$, $$30$$])), \
            element]
          return Input_Pattern_Creator.fail
        }(bindings, element)
        return Input_Pattern_Creator.fail
      }(bindings, $$31$$elements$$):
        matched_mp_elems := matched_mp_elems + \
          $$34$$new_matched_mp_elems$$
        bindings := bindings + $$35$$new_bindings$$
        return [matched_mp_elems, bindings]
        matched_mp_elems := $$32$$matched_mp_elems_backup$$
        bindings := $$33$$bindings_backup$$
      matched_mp_elems := $$38$$matched_mp_elems_backup$$
      bindings := $$39$$bindings_backup$$
    return Input_Pattern_Creator.fail
  }.apply(objs):
    self._matched_objs.extend(matched_objs)
    if not rtn := unbound_func (matched_objs, bindings){
      concatted_id := Output_Pattern_Creator.concat_id(matched_objs)
      if self._transformed_cache["User_Type_Attr_To_Column"]. \
        contains(concatted_id):
        return self._transformed_cache["User_Type_Attr_To_Column"] \
          [concatted_id]
      tracing_i := self._tracing.len()
      $$43$$prefix$$ := bindings["prefix"]
      $$44$$n$$ := bindings["n"]
      $$45$$cn$$ := bindings["cn"]
      $$46$$CA$$ := bindings["CA"]
      out_elems := []
      out_elems.append(unbound_func (){
        output_objs := []
        for $$47$$ca$$ := $$46$$CA$$.iterate():
          output_objs.append($$49$$self$$.transform( \
            [$$48$$concat_name$$($$43$$prefix$$, $$44$$n$$)], \
            [$$47$$ca$$]))
        return output_objs
      }())
      out_elem := out_elems[0]
      self._transformed_cache["User_Type_Attr_To_Column"] \
        [concatted_id] := out_elem
      if not out_elem is Output_Pattern_Creator.null:
        tracing := [Output_Pattern_Creator.List(matched_objs), \
          out_elems]
        tracing := Output_Pattern_Creator.trace_reduce(tracing)
        self._tracing.insert(tracing_i, tracing)
        self._tracing_rule.insert(tracing_i, \
          "User_Type_Attr_To_Column")
      return out_elem
    }(matched_objs, bindings):
      raise MT.Exceptions.Exception(MT.Strings.format( \
        "Output pattern of User_Type_Attr_To_Column failed to " + \
        "generate anything for '%s'.", objs.to_str()))
    return rtn
  else:
    return MT.fail
}
bound_func Primitive_Type_Attr_To_Column(*objs){
  $$75$$concat_name$$ := concat_name
  if matched_objs, bindings := unbound_func (*args){
    if args.len() > 2:
```

```
  return Input_Pattern_Creator.fail
matched_mp_elems := Set{}
bindings := Dict{}
if 0 < args.len():
  $$66$$elements$$ := args[0]
else:
  $$66$$elements$$ := self._root_set
$$67$$matched_mp_elems_backup$$ := matched_mp_elems
$$68$$bindings_backup$$ := bindings
for $$69$$new_matched_mp_elems$$, $$70$$new_bindings$$, \
  $$71$$matched_elem$$ := unbound_func (bindings, elements){
  for element := elements.iterate() & yield \
    unbound_func (bindings, element){
    if not Input_Pattern_Creator.TM.type_match("String", element):
      return Input_Pattern_Creator.fail
    for yield $$51$$ := unbound_func (bindings, element){
      if bindings.contains("prefix") & \
        not bindings["prefix"] == element:
        return Input_Pattern_Creator.fail
      return [Set{}, Dict{"prefix" : element}, element]
    }(bindings, element) & $$50$$ := bindings + $$51$$[1] & \
      [Set{element}, \
      Input_Pattern_Creator.Functional.foldl( \
      Input_Pattern_Creator._adder, \
      Input_Pattern_Creator.Functional.map( \
      Input_Pattern_Creator._element1, [$$51$$])), element]
    return Input_Pattern_Creator.fail
  }(bindings, element)
  return Input_Pattern_Creator.fail
}(bindings, $$66$$elements$$):
  matched_mp_elems := matched_mp_elems + \
    $$69$$new_matched_mp_elems$$
  bindings := bindings + $$70$$new_bindings$$
  if 1 < args.len():
    $$60$$elements$$ := args[1]
  else:
    $$60$$elements$$ := self._root_set
  $$61$$matched_mp_elems_backup$$ := matched_mp_elems
  $$62$$bindings_backup$$ := bindings
  for $$63$$new_matched_mp_elems$$, $$64$$new_bindings$$, \
    $$65$$matched_elem$$ := unbound_func (bindings, elements){
    for element := elements.iterate() & yield \
      unbound_func (bindings, element){
      if not Input_Pattern_Creator.TM.type_match("Attribute", \
        element):
        return Input_Pattern_Creator.fail
      for yield $$52$$ := bindings & $$54$$ := \
        unbound_func (bindings){
        slot_element := element.name
        for matched_mp_elems, new_bindings, matched_elem \
          := unbound_func (bindings, element){
          if bindings.contains("n") & \
            not bindings["n"] == element:
            return Input_Pattern_Creator.fail
          return [Set{}, Dict{"n" : element}, element]
        }(bindings, slot_element):
          if slot_element == matched_elem:
            yield [matched_mp_elems, new_bindings, \
              matched_elem]
        return Input_Pattern_Creator.fail
      }($$52$$) & $$53$$ := $$52$$ + $$54$$[1] & $$59$$ := \
        unbound_func (bindings){
        slot_element := element.type
        for matched_mp_elems, new_bindings, matched_elem := \
          unbound_func (bindings, element){
          if not Input_Pattern_Creator.TM.type_match( \
            "PrimitiveDataType", element):
            return Input_Pattern_Creator.fail
          for yield $$55$$ := bindings & $$57$$ := \
```

```
              unbound_func (bindings){
              slot_element := element.name
              for matched_mp_elems, new_bindings, \
                matched_elem := unbound_func (bindings, \
                  element){
                if bindings.contains("pn") & \
                  not bindings["pn"] == element:
                  return Input_Pattern_Creator.fail
                return [Set{}, Dict{"pn" : element}, \
                  element]
              }(bindings, slot_element):
                if slot_element == matched_elem:
                  yield [matched_mp_elems, \
                    new_bindings, matched_elem]
              return Input_Pattern_Creator.fail
            }($$55$$) & $$56$$ := $$55$$ + $$57$$[1] & \
              [Set{element}, \
              Input_Pattern_Creator.Functional.foldl( \
              Input_Pattern_Creator._adder, \
              Input_Pattern_Creator.Functional.map( \
              Input_Pattern_Creator._element1, [$$57$$])), \
              element]
            return Input_Pattern_Creator.fail
          }(bindings, slot_element):
            if slot_element == matched_elem:
              yield [matched_mp_elems, new_bindings, \
                matched_elem]
          return Input_Pattern_Creator.fail
        }($$53$$) & $$58$$ := $$53$$ + $$59$$[1] & \
          [Set{element}, \
          Input_Pattern_Creator.Functional.foldl( \
          Input_Pattern_Creator._adder, \
          Input_Pattern_Creator.Functional.map( \
          Input_Pattern_Creator._element1, [$$54$$, $$59$$])), \
          element]
        return Input_Pattern_Creator.fail
      }(bindings, element)
      return Input_Pattern_Creator.fail
    }(bindings, $$60$$elements$$):
      matched_mp_elems := matched_mp_elems + \
        $$63$$new_matched_mp_elems$$
      bindings := bindings + $$64$$new_bindings$$
      return [matched_mp_elems, bindings]
      matched_mp_elems := $$61$$matched_mp_elems_backup$$
      bindings := $$62$$bindings_backup$$
    matched_mp_elems := $$67$$matched_mp_elems_backup$$
    bindings := $$68$$bindings_backup$$
  return Input_Pattern_Creator.fail
}.apply(objs):
  self._matched_objs.extend(matched_objs)
  if not rtn := unbound_func (matched_objs, bindings){
    concatted_id := Output_Pattern_Creator.concat_id(matched_objs)
    if self._transformed_cache["Primitive_Type_Attr_To_Column"]. \
      contains(concatted_id):
      return self._transformed_cache \
        ["Primitive_Type_Attr_To_Column"][concatted_id]
    tracing_i := self._tracing.len()
    $$72$$prefix$$ := bindings["prefix"]
    $$73$$n$$ := bindings["n"]
    $$74$$pn$$ := bindings["pn"]
    out_elems := []
    out_elems.append([unbound_func (){
      user_args := Dict{"name" : $$75$$concat_name$$( \
        $$72$$prefix$$, $$73$$n$$), "type" : $$74$$pn$$}
      all_args := []
      args_processed := 0
      for attr_name := Output_Pattern_Creator.TM. \
        all_attrs_in_order(Output_Pattern_Creator.TM. \
        _CLASSES_REPOSITORY["Column"]).iterate():
```

```
          if args_processed == user_args.len():
            break
          if user_args.contains(attr_name):
            all_args.append(user_args[attr_name])
            args_processed += 1
          else:
            all_args.append(Output_Pattern_Creator.null)
        return Output_Pattern_Creator.TM._CLASSES_REPOSITORY \
          ["Column"].new.apply(all_args)
      }()])
      out_elem := out_elems[0]
      self._transformed_cache["Primitive_Type_Attr_To_Column"] \
        [concatted_id] := out_elem
      if not out_elem is Output_Pattern_Creator.null:
        tracing := [Output_Pattern_Creator.List(matched_objs), \
          out_elems]
        tracing := Output_Pattern_Creator.trace_reduce(tracing)
        self._tracing.insert(tracing_i, tracing)
        self._tracing_rule.insert(tracing_i, \
          "Primitive_Type_Attr_To_Column")
      return out_elem
    }(matched_objs, bindings):
      raise MT.Exceptions.Exception(MT.Strings.format( \
        "Output pattern of Primitive_Type_Attr_To_Column failed to " + \
        "generate anything for '%s'.", objs.to_str()))
    return rtn
  else:
    return MT.fail
}
```

Appendix E.

Model serializer

E.1. Overview

The TM `Serializer` module comprises functions to serialize and deserialize TM models, and tracing information. The serializer is essentially a simple graph walking function which flattens a model into an XML tree structure; references between nodes are made by using model elements' identifiers and an XML attribute `id`.

The deserializer is slightly more complex in operation. It utilizes Converge's `XML.Whole_Parser` module which provides a simple mechanism for parsing and traversing an XML file. The problem the deserializer faces is that as it works through its input creating appropriate model elements, it may find an `id` reference to an element which has not yet been created. In such cases, it creates a blank TM model element which it uses as a dummy holder to be filled in later when the full definition of the element is encountered in the file. This however means that during the process of deserialization the model being created may not be conformant to its meta-model. In order to prevent exceptions being raised whilst the model is deserialized, the deserializer sets the `_is_initialized` field of each element to `0`, ensuring that checks against the meta-model are not made. When all elements are completely deserialized, it then goes back over each element, setting this field to `1`, finally running the meta-models constraints against the meta-model to ensure that it has been recreated correctly.

E.2. Example output

This section shows the XML output from the TM `Serializer` model on the example of section 6.2.2. Firstly the ML2 input model:

```
<Model>
  <Element id="13" of="ML2_Package">
    <Attribute name="name">
      <String val="Personnel" />
    </Attribute>
    <Attribute name="elements">
```

```
      <Set>
        <Ref ref="12" />
        <Ref ref="11" />
        <Ref ref="10" />
      </Set>
    </Attribute>
  </Element>
  <Element id="12" of="ML2_Association">
    <Attribute name="name">
      <String val="PE" />
    </Attribute>
    <Attribute name="end2_name">
      <String val="manager" />
    </Attribute>
    <Attribute name="end1_name">
      <String val="employees" />
    </Attribute>
    <Attribute name="end2_multiplicity">
      <Int val="1" />
    </Attribute>
    <Attribute name="end1_multiplicity">
      <Int val="-1" />
    </Attribute>
    <Attribute name="end2_directed">
      <Int val="0" />
    </Attribute>
    <Attribute name="end1_directed">
      <Int val="0" />
    </Attribute>
    <Attribute name="end2">
      <Ref ref="11" />
    </Attribute>
    <Attribute name="end1">
      <Ref ref="10" />
    </Attribute>
  </Element>
  <Element id="11" of="ML2_Class">
    <Attribute name="name">
      <String val="Manager" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
  </Element>
  <Element id="10" of="ML2_Class">
    <Attribute name="name">
      <String val="Employee" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
  </Element>
</Model>
```

Then the ML1 target model produced by the transformation on its initial execution:

```
<Model>
  <Element id="Package_To_Package_0__13" of="ML1_Package">
    <Attribute name="name">
      <String val="Personnel" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
```

```
    </Attribute>
    <Attribute name="elements">
      <Set>
        <Ref ref="Association_To_Association_0__12" />
        <Ref ref="Association_To_Association_1__12" />
        <Ref ref="Class_To_Class_0__11" />
        <Ref ref="Class_To_Class_0__10" />
      </Set>
    </Attribute>
  </Element>
  <Element id="Association_To_Association_0__12" of="ML1_Association">
    <Attribute name="name">
      <String val="manager" />
    </Attribute>
    <Attribute name="multiplicity">
      <Int val="1" />
    </Attribute>
    <Attribute name="to">
      <Ref ref="Class_To_Class_0__11" />
    </Attribute>
    <Attribute name="from">
      <Ref ref="Class_To_Class_0__10" />
    </Attribute>
  </Element>
  <Element id="Association_To_Association_1__12" of="ML1_Association">
    <Attribute name="name">
      <String val="employees" />
    </Attribute>
    <Attribute name="multiplicity">
      <Int val="-1" />
    </Attribute>
    <Attribute name="to">
      <Ref ref="Class_To_Class_0__10" />
    </Attribute>
    <Attribute name="from">
      <Ref ref="Class_To_Class_0__11" />
    </Attribute>
  </Element>
  <Element id="Class_To_Class_0__11" of="ML1_Class">
    <Attribute name="name">
      <String val="Manager" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
  </Element>
  <Element id="Class_To_Class_0__10" of="ML1_Class">
    <Attribute name="name">
      <String val="Employee" />
    </Attribute>
    <Attribute name="parents">
      <List>

      </List>
    </Attribute>
  </Element>
</Model>
```

And finally the tracing information generated by the transformation on its initial execution:

```
<Tracing>
  <Trace rule="Class_To_Class">
    <From>
      <Ref ref="10" />
    </From>
    <To>
      <Ref ref="Class_To_Class_0__10" />
```

```
    </To>
  </Trace>
  <Trace rule="Class_To_Class">
    <From>
      <Ref ref="11" />
    </From>
    <To>
      <Ref ref="Class_To_Class_0__11" />
    </To>
  </Trace>
  <Trace rule="Association_To_Association">
    <From>
      <Ref ref="12" />
    </From>
    <To>
      <Ref ref="Association_To_Association_0__12" />
      <Ref ref="Association_To_Association_1__12" />
    </To>
  </Trace>
  <Trace rule="Package_To_Package">
    <From>
      <Ref ref="13" />
    </From>
    <To>
      <Ref ref="Package_To_Package_0__13" />
    </To>
  </Trace>
</Tracing>
```

Appendix F.

Glossary

Bidirectional A transformation which can both transform instances of M1 into instances of M2, and instances of M2 into M1.

Bound function A Converge function which has its self variable bound to a particular object. Equivalent to the term ‘method’ in many OO languages.

Change propagation The ability to take two models related in a transformation and, given changes in one or the other model, to make the corresponding changes to the other model.

Conflict report A record of inconsistencies, relative to a transformation, between two models involved in change propagation.

Conformance operator An operator relating model elements in a change propagating transformation.

Conjunction The Converge `&` operator which conjoins expressions. This also serves as Converge’s equivalent of the standard `and` operator.

Disjunction The Converge `|` operator which successively generates each of its expressions. This also serves as Converge’s equivalent of the standard `or` operator.

DSL block A block of code in a user-specified syntax embedded within a Converge source file.

DSL implementation function The function which introduces a DSL block, and which is responsible, at compile-time, for converting the DSL block into an ITree.

Declaration quasi-quotes A form of quasi-quotes which does not perform α-renaming at the top-level.

Generator A Converge function which generates multiple values via the `yield` keyword.

Goal-directed evaluation The evaluation strategy, inherited from Icon, which allows backtracking amongst Converge expressions to find values which allow execution to continue.

ITree A converge Abstract Syntax Tree.

Key One or more attributes which collectively identify a model element.

Lifting The process of converting a normal Converge value, such as a string, into its ITree equivalent.

Metamodel Literally 'the model of a model'. Often referred to as a modelling language. Defines what its valid instances are, possibly by a denotational or creational style.

Model element expression An MT or PMT expression which creates model elements.

Model element pattern An MT or PMT expression which matches model elements.

Model expression An overarching MT and PMT term encompassing both normal Converge expressions, model element expressions and model element patterns.

MT The MT language is a unidirectional stateless model transformation language.

Multiplicity A constraint specifying the number of times a pattern may match against model elements.

Pattern A syntactic convenience for matching data types, most commonly realised in the real world as textual regular expressions.

PMT The PMT language is a unidirectional change propagating model transformation language.

Quasi-quotes A mechanism for expressing ITree's via standard Converge concrete syntax.

Root set of source elements The elements initially passed to a transformation.

Rule A transformation is comprised of one or more transformation rules. A rule can be thought of as being equivalent to a function.

Slot comparisons A standard MT model element pattern is comprised of one or more slot comparisons. Each slot comparison checks to see whether a given slot in a source model element is correctly related to the value returned by a model expression.

Slot conformances A standard PMT model element expression is comprised of one or more slot conformances. Each slot conformance checks to see whether a given slot in a target model element is correctly related to the value returned by a model expression.

Source / target clauses A transformation rule is said to be comprised of two or more clauses; at least one source clause, matching source elements, and at least one target clause creating target elements.

Splice A splice is an expression in a Converge file which will be evaluated at compile-time.

Splicing The act of replacing a splice with the ITree created by evaluated the expression at compile-time.

TM The Typed Modelling (TM) language allows simple modelling languages to be expressed, and model elements to be created from those meta-models.

Variable binding An MT or PMT variable which is bound to a particular model element as matching proceeds.

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[ACE+02] Biju Appukuttan, Tony Clark, Andy Evans, Stuart Kent, Girish Maskeri, Paul Sammut, Laurence Tratt, and James S. Willans. Unambiguous uml submission to uml 2 infrastructure rfp, September 2002. OMG document `ad/2002-06-14`.

[ACR+03] Biju K. Appukuttan, Tony Clark, Sreedhar Reddy, Laurence Tratt, and R. Venkatesh. A pattern based model driven approach to model transformations, November 2003. Metamodelling for MDA 2003.

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. Technical Report 7, University of Bremen, 1999.

[AGGI04] Jim Amsden, Tracy Gardner, Catherine Griffin, and Sridhar Iyengar. Draft UML 1.4 profile for automated business processes with a mapping to BPEL 1.0, April 2004. `http://dwdemos.dfw.ibm.com/wstk/common/wstkdoc/services/demos/uml2bpel/docs/UMLProfileForBusinessProcesses1.1.pdf`.

[AH02] John Aycock and R. Nigel Horspool. Practical Earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

[AK02] David H. Akehurst and Stuart J. H. Kent. A relational approach to defining transformations in a metamodel. In Jean-Marc Jézéquel, Heinrich Hussmann, and Steve Cook, editors, *UML 2002 – The Unified Modeling Language : 5th International Conference*, pages 243 – 258. Springer-Verlag, 2002.

[AKS03] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph transformations on domain-specific models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, November 2003.

[AP04] Marcus Alanen and Ivan Porres. Change propagation in a model-driven development tool. Presented at WiSME part of UML 2004, October 2004.

[BÓ5] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[Bar99] Roman Barták. Constraint programming: What is behind? In *Proceedings of CPDC99*, pages 7 – 15, June 1999.

[Baw99] Alan Bawden. Quasiquotation in LISP. Workshop on Partial Evaluation and Semantics-Based Program Manipulation, January 1999.

[Baw00] Alan Bawden. First-class macros have types. In *Proc. 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–141, January 2000.

[BC89] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proc. OOPSLA '89*, October 1989.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.

[BG02] Jean Bézivin and Sébastien Gérard. A preliminary identification of MDA components. In *Generative Techniques in the context of Model Driven Architecture*, Nov 2002.

[BH90] Heinz-Dieter Böcker and Jürgen Herczeg. What tracers are made of. In *Proc. ECOOP '90*, pages 89–99, 1990.

[BHW04] Simon M. Becker, Thomas Haase, and Bernhard Westfechtel. Model-based a-posteriori integration of engineering tools for incremental development processes. *SoSYM*, 2004. To appear.

[Big98] Ted J. Biggerstaff. Pattern matching for program generation: A user manual. Technical Report TR-98-55, Microsoft Research, 1998.

[BJR00] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. OMG unified modeling language specification, 2000.

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. Elan: A logical framework based on computational systems. In *Proc. first international workshop on rewriting logic*, 1996.

[BM02] Peter Braun and Frank Marschall. Transforming object oriented models with BOTL. *International Workshop on Graph Transformation and Visual Modeling Techniques*, 72(3), 2002.

[BM03] Peter Braun and Frank Marschall. Botl – the bidirectional object oriented transformation language. Technical Report TUM-I0307, Institut für Informatik der Technischen Universität München, May 2003.

[BMN02] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 28(1):21–39, 2002.

[BP99] Jonathan Bachrach and Keith Playford. D-expressions: Lisp power, Dylan style, 1999. `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf` Accessed Sep 22 2004.

[BP01] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proc. OOPSLA*, pages 31–42, November 2001.

[BS00] Claus Brabrand and Michael Schwartzbach. Growing languages with metamorphic syntax macros. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN. ACM, 2000.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proc. OOPSLA'04*, Vancouver, Canada, October 2004. ACM SIGPLAN.

[CEK01] Tony Clark, Andy Evans, and Stuart Kent. Initial submission to OMG RFP's ad/00-09-01 (UML 2.0 infrastructure) ad/00-09-03 (UML 2.0 OCL), 2001.

[CEL+96] Manuel Clavel, Steven Eker, Patrick Lincoln, José, and Meseguer. Principles of maude. In José Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, September 1996.

[CEM+04] Tony Clark, Andy Evans, Girish Maskeri, Paul Sammut, and James S. Willans. Modelling language transformations. *L'Objet*, 9, April 2004.

[CESW04] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language driven development, September 2004. Available from `http://www.xactium.com/` Accessed Sep 22 2004.

[CMA93] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible grammars for language specialization. In *Proc. Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 11–31, August 1993.

[Coi87] Pierre Cointe. Metaclasses are first class: the ObjVLisp model. In *Object Oriented Programming Systems Languages and Applications*, pages 156–162, October 1987.

[Cor04] James R. Cordy. TXL - a language for programming language tools and applications. In *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications*, April 2004.

[COST04] Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. 3016:50–71, 2004.

[CP05] Howard Chivers and Richard Paige. XRound: Bidirectional transformations via a reversible template language. In *To appear Proc. European Conference on MDA (ECMDA)*, November 2005.

[CS03] Compuware and Sun. XMOF queries, views and transformations on models using MOF, OCL and patterns, August 2003. OMG document `ad/2003-08-07`.

[DDDCG02] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle II. *ACM Transactions On Programming Languages and Systems*, 24(2):153–191, 2002.

[DGR01] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53–73, Nov 2001.

[DHB92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. In *Lisp and Symbolic Computation*, volume 5, pages 295–326, December 1992.

[DIC03] DSTC, IBM, and CBOP. MOF query / views / transformations first revised submission, August 2003. OMG document `ad/2003-08-03`.

[DM95] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proc. IJCAI'95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI*, pages 29–38, August 1995.

[dR03] Daniel de Rauglaudre. *Camlp4 - Reference Manual*, September 2003. `http://caml.inria.fr/camlp4/manual/` Accessed Sep 22 2004.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), February 1970.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 2002.

[Ecl04] IBM. *Eclipse*, 2004. `http://www.eclipse.org/`.

[Egy01] Alexander Egyed. A scenario-driven approach to traceability. In *Proc. 23rd International Conference on Software Engineering*, pages 123 – 132, 2001.

[Eva98] Andy Evans. Reasoning with UML class diagrams. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, October 1998.

[FD98] Ira R. Forman and Scott H. Danforth. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming*. Addison-Wesley, 1998.

[FHK+92] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint logic programming: An informal introduction. In *Logic Programming in Action*, number 636 in LNCS. Springer, 1992.

[For02] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming*, pages 36–47, October 2002.

[GG93] Ralph E. Griswold and Madge T. Griswold. History of the : programming language. *j-SIGPLAN*, 28(3):53–68, March 1993.

[GG96a] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[GG96b] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[GGKH03] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. Query / views / transformations submissions & recommendations towards final standard, August 2003. OMG document `ad/03-08-02`.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Orientated Software*. Addison Wesley, 1994.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[GLR+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation: First International Conference, ICGT 2002*, pages 90–105, October 2002.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[Gog00] Martin Gogolla. Graph transformations on the UML metamodel. In Jose D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and Joe B. Wells, editors, *ICALP Workshop on Graph Transformations and Visual Modeling Techniques*, pages 359–371. Carleton Scientific, 2000.

[GPP98] Martin Gogolla and Francesco Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, January 1989.

[GST01] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *Proc. International Conference on Functional Programming (ICFP)*, volume 36 of *SIGPLAN*. ACM, September 2001.

[Gud92] David A. Gudeman. Denotational semantics of a goal-directed language. *ACM Transactions on Programming Languages and System*, 14(1):107–125, January 1992.

[GZK03] Martin Gogolla, Paul Ziemann, and Sabine Kuske. Towards an integrated graph based semantics for UML. In Paolo Bottoni and Mark Minas, editors, *Proc. Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2002)*, volume 72 of *Electronic Notes in Theoretical Computer Science*, 2003.

[HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini and H.-J. Kreowski, editors, *Proceedings First International Conference on Graph Transformation (ICGT 02)*, pages 161 – 176. Springer-Verlag, October 2002.

[HM76] James Hunt and M. Douglas McIlroy. An algorithm for differential file comparison. Technical Report 41, Bell Laboratories Computing Lab, July 1976.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[Jaf03] Aubrey Jaffer. *SCM Scheme Implementation*, November 2003. `http://www.swiss.ai.mit.edu/~jaffer/scm_toc` Accessed Sep 16 2004.

[JE04] Sven Johann and Alexander Egyed. Instant and incremental transformation of models. September 2004.

[Jef02] Clinton L. Jeffery. *Godiva Language Reference Manual*, November 2002. `http://www.cs.nmsu.edu/~jeffery/godiva/godiva.html`.

[JL99] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1999.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, July 1994.

[JMPP03] Clinton Jeffery, Shamim Mohamed, Ray Pereda, and Robert Parlett. *Programming with Unicon*, April 2003. `http://unicon.sourceforge.net/book/ub.pdf`.

[Jon03] Simon Peyton Jones. *Haskell 98 Languages and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised(5) report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kep02] Stephan Kepser. A proof of the Turing-completeness of XSLT and XQuery. Technical Report SFB 441, Eberhard Karls Universität Tübingen, June 2002.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Symposium on Lisp and Functional Programming*, pages 151–161. ACM, 1986.

[KHE03] Jochen M. Küster, Reiko Heckel, and Gregor Engels. Defining and validating transformations of UML models. In *IEEE Symposium on Visual Languages and Formal Methods*, October 2003.

[KRW04] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An adaptable TGG interpreter for in-memory model transformations. In *Proc. FUJABA Days 2004*, pages 35–38, September 2004.

[LB98] Kevin Lano and Juan Bicarregui. UML refinement and abstraction transformations. In *Second Workshop on Rigorous Object Oriented Methods: ROOM 2, Bradford*, May 1998.

[LKM+02] Tihamer Levendovszky, Gabor Karsai, Miklos Maroti, Akos Ledeczi, and Hassan Charaf. Model reuse with metamodel-based transformations. In Cristina Gacek, editor, *ICSR*, volume 2319 of *LNCS*. Springer, 2002.

[MDJ02] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Proc. 1st International Conf. Graph Transformation*, volume 2505 of *LNCS*, pages 286–301, 2002.

[MG04] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation and its application to graph transformation. Unpublished, November 2004.

[MHS03] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. Technical report, Centrum voor Wiskundeen Informatica, December 2003.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. Tool demonstration: The fujaba environment. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE)*, pages 742–745. ACM Press, 2000.

[OJ04] Compuware. *OptimalJ*, 2004. `http://www.compuware.com/products/optimalj/`.

[OMG97] Object constraint language specification, 1997. OMG document `ad/97-08-08`.

[OMG00] Object Management Group. *Meta Object Facility (MOF) Specification*, 2000. `formal/00-04-03`.

[OMG02] Object Management Group. *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, 2002. OMG document `ad/2002-04-10`.

[OMG03] OMG. XML Metadata Interchange (XMI), May 2003. `http://www.omg.org/cgi-bin/doc?formal/2003-05-02`.

[OQV03] OpenQVT. Response to the MOF 2.0 query / views / transformations RFP, August 2003. OMG document `ad/2003-08-05`.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[PBG01] Mikaël Peltier, Jean Bézivin, and Gabriel Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In *WTUML 2001, Italy*, April 2001.

[PR03] Richard Paige and Alek Radjenovic. Towards model transformation with TXL, November 2003. Metamodelling for MDA 2003.

[Pre00] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[Pro97] Todd A. Proebsting. Simple translation of goal-directed evaluation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–6, 1997.

[Que96] Christian Queinnec. Macroexpansion reflective tower. In *Proc. Reflection'96*, pages 93–104, April 1996.

[QVT03a] QVT-Partners. First revised submission to QVT RFP, August 2003. OMG document `ad/03-08-08`.

[QVT03b] QVT-Partners initial submission to QVT RFP, 2003. OMG document `ad/03-03-27`.

[RR93] Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proc. 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510, 1993.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *Proc. International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163, 1994.

[Sch05] Friedrich Wilhelm Schröer. *The ACCENT Grammar Language*, 2005. `http://accent.compilertools.net/language.html` Accessed Jan 25 2005.

[SCK03] Sean Seefried, Manuel M. T. Chakravarty, and Gabriele Keller. Optimising embedded DSLs using Template Haskell. In *Draft Proc. Implementation of Functional Languages*, 2003.

[SeABP99] Tim Sheard, Zine el Abidine Benaissa, and Emir Pasalic. DSL implementation using staging and monads. In *Proc. 2nd conference on Domain Specific Languages*, volume 35 of *SIGPLAN*, pages 81–94. ACM, October 1999.

[Sen03] Shane Sendall. Combining generative and graph transformation techniques for model transformation: An effective alliance? In *Generative techniques in the context of MDA*, October 2003.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop 2002*. ACM, 2002.

[SMO04] Kamil Skalski, Michal Moskal, and Pawel Olszta. Meta-programming in Nemerle, 2004. `http://nemerle.org/metaprogramming.pdf` Accessed Oct 1 2004.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, second edition, March 1994.

[Ste99] Guy L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221 – 236, October 1999.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[Sut63] Ivan Sutherland. Sketchpad: a man–machine graphical communication system. In *Proceedings Spring Joint Computer Conference, IFIPS*, pages 329–346, 1963.

[SW98] Andy Schürr and Andreas J. Winter. UML packages for programmed graph rewriting systems. In *Proc. TAGT'98 - Theory and Application of Graph Transformations*, November 1998.

[SWZ99] Andy Schürr, Andreas J. Winter, and Albert Zündorf. *Handbook of Graph Grammars and Graph Transformation*, volume 2, chapter PROGRES: Language and Environment, pages 487–550. 1999.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, October 1999.

[TC03] Laurence Tratt and Tony Clark. Issues surrounding model consistency and QVT. Technical Report TR-03-08, Department of Computer Science, King's College London, December 2003.

[TCIK99] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In *Proc. 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, 1999.

[TH00] David Thomas and Andrew Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[Tra05] Laurence Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005.

[Var03] Dániel Varró. *Automated Model Transformations for the Analysis of IT Systems*. PhD thesis, Budapest University of Technology and Economics, December 2003.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. volume 35 of *SIGPLAN Notices*, pages 26–36, June 2000.

[Vel95] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[VP03] Dániel Varró and András Pataricza. UML action semantics for model transformation systems. *Periodica Politechnica*, 47(3):167–186, 2003.

[vR01] Guido van Rossum. Python 2.2 reference manual, 2001. `http://www.python.org/doc/2.2/ref/ref.html`.

[vR03] Guido van Rossum. Python 2.3 reference manual, 2003. `http://www.python.org/doc/2.3/ref/ref.html` Accessed Aug 31 2005.

[VV04] Gergely Varró and Dániel Varró. Graph transformation with incremental updates. In *Proc. GT-VMT 2004, International Workshop on Graph Transformation and Visual Modelling Techniques*, ENTCS, March 2004. To appear.

[VVP02] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

[W3C99a] W3C. *XML Path Language (XPath) 2.0*, November 1999. `http://www.w3.org/TR/xpath`.

[W3C99b] W3C. *XSL Transformations (XSLT)*, 1999. `http://www.w3.org/TR/xslt`.

[WC93] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. SIGPLAN*, pages 156–165, 1993.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, third edition, 2000.

[Whi02] Jon Whittle. Transformations and software modeling languages: Automating transformations in UML. In Jean-Marc Jézéquel, Heinrich Hussmann, and Steve Cook, editors, *UML 2002 – The Unified Modeling Language : 5th International Conference*, pages 227 – 242. Springer-Verlag, 2002.

[Wil03] Edward D. Willink. UMLX : A graphical transformation language for MDA. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.

[Wil05] Gregory V. Wilson. Extensible programming for the 21st century. *Queue*, 2(9):48–57, January 2005.