UNIVERSITY OF TWENTE.

Domain Specific Embedded Languages and Model Driven Engineering

MSc CS SE
Stefan Kroes

SE Group

Graduation Committee:
Dr. Ivan Kurtev
Dr. ir. Klaas van den Berg
A. Göknil, MSc

February 14, 2010

Contents

List of Figures

List of Tables

Abbreviations

AOP	Aspect Oriented Programming
API	Application Programmer Interface
ATL	ATLAS Transformation Language
DSEL	Domain Specific Embedded Language
DSL	Domain Specific Language
EMF	Eclipse Modelling Framework
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MTEL	Model Transformation Embedded Language
MTL	Model Transformation Language
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query/View/Transformation
UML	Unified Modelling Language

Abstract

Model Driven Engineering (MDE) [20] is an emerging approach to software engineering. The main concepts in MDE are models, meta-models and transformations. Models are the first class entities of any MDE environment. A model is a representation of its subject, describing only the relevant aspects of it and omitting everything else. While a model's subject is *represented* by the model, the model can be *interpreted* to obtain information about its subject. The subjects of these models are parts of aspects of the software system under development. Models can be refined and (de)composed using transformations, defined in a Model Transformation Language (MTL).

MTL's are often implemented using a traditional approach to language design, using a dedicated compiler and/or interpreter. We will explore an alternative but equally valid method of developing a MTL, namely developing it as an Domain Specific Language (DSL) within the context of an existing general purpose programming language (host language). This kind of DSL is often referred to as an DSEL [18] and it's essentially an advanced form of the Application Programmer Interface (API).

We chose Ruby as our host language because of its concise and flexible syntax, extensive features for the creation of DSEL's, support for declarative and functional programming, reflective features and mature standard library. For our goal, the most important of these are code blocks that can be evaluated in any context and operator overloading.

This report introduces and explains our Model Transformation Embedded Language (MTEL) called TRans. The defining characteristics of TRans, apart from it being a MTEL, are support multiple source and target models, automatic execution ordering using a topological sort algorithm, advanced tracing functionality and Aspect Oriented Programming (AOP) features. By implementing TRans as a MTEL we where able to reduce implementation effort significantly. We deem that this reduction far outweighs the disadvantages of this approach, especially for research purposes.

The research that is most obviously related to our research is the RubyTL language for model transformation. It too is a MTEL based on Ruby. When compared to RubyTL, TRans has the following advantages: some AOP support, decoupling from modelling language, a more advanced DSEL and a cleaner implementation. We where able to achieve these advantages because Ruby DSEL creation techniques, as well as the MDE domain, have evolved and TRans is a more recent effort.

Chapter 1

Introduction

Most innovative efforts in the field of software engineering attempt to abstract away from the specifics of the actual machine. We strive to create environments in which software engineers can consider functionality as much as possible rather than having to worry about the specifics of their implementation platform. The main consideration for these abstractions is the need to hide as much complexity as possible while limiting possibilities as little as possible. We started off in this direction quite nicely by building two layers of abstraction of top of the actual machine code; assemblers and programming languages. Programming languages where in turn used develop interpreters and virtual machines, paving the way for multi-platform and dynamic programming languages. While many efforts are being made to improve programming languages themselves, a new layer of abstraction that builds on them has not emerged for a long time. Model Driven Engineering (MDE) [20] is believed by many to be this next layer of abstraction.

MDE is concerned with models and transformations between these models. From a practical point of view models are essentially simple data structures that can be used to express anything. From a theoretical point of view however, the concept of a model is much broader. A model is an abstraction of its subject, describing only the relevant aspects of it and omitting everything else. This means multiple models can be abstractions of the same subject, for example a software system, each model describing different aspects of it. In the ideal model driven software project, models expressing individual concerns are mapped (optionally through intermediate models) to models expressing an executable system, often in the form of programming code. This is an exciting new idea, promising true separation of concerns, better maintainability and more effective communication through different and often graphical representations of models.

Transformations and mappings between models are two terms describing the same thing, chosen from an imperative and declarative point of view respectively. In the rest of this report we will refer to this activity as model transformation, as it's so much more common. The process of model transformation takes one or more models as input and produces one or more output models from them. These output models can either be an updated version of an input model or a newly generated model. Model transformations are expressed in a model transformation language and are essential to model driven engineering.

This chapter discusses the current and desired situations in the field of model transformation languages, our approach to solving some of these problems and finally the outline of this thesis.

1.1 Problem Statement

MDE is currently not the prevalent method of implementing software systems, even though the first examples of MDE can be traced back to the 1980's. MDE has some serious problems that need to be solved before it can assume a more significant role in software development. From a theoretical point of view, a major problem with MDE is the inability to unambiguously and completely describe a system using only conceptual models. This applies particularly to the modelling of the behavioural aspects of the system. Additionally, there are practical problems with adoption and tool support. The theoretical and practical problems enforce each other in a vicious cycle. A lack of adoption causes MDE to evolve at a very slow pace while at the same time the immaturity of MDE is preventing it from being adopted on a large scale.

This study is concerned with the more practical problems of MDE, and model transformations in particular. A language that is designed specifically for this purpose is called a Model Transformation Language (MTL). A handful of relatively mature languages exist in this area, the most notable of which are QVT [27], ATLAS Transformation Language (ATL) [5] [16] and Tefkat [23]. These languages were developed using a classic approach which means defining a grammar and developing a dedicated parser and interpreter or compiler for them. While this is a valid approach that provides maximum flexibility, history has shown that these kinds of languages take a long time to mature. Some major languages from different areas of computer programming that where developed in this fashion are C, Java and Ruby, which all took the better part of a decade to become mature, standardised and generally accepted.

We also observe an overlap in desirable language features between modern programming languages and MTL's. Aspects like parsing and interpretation, expressions, data structures, garbage collection, I/O and other libraries, method dispatching, control structures and exception handling are all desirable in a MTL and have already been implemented in most major programming languages. Ideally we could borrow these language features from a mature programming language and focus solely on model transformation specific problems when developing a MTL. Borrowing all these facilities from another language wouldn't only save us the burden of implementing them ourselves but would also result in a better product. Large parts of the MTL would have matured much longer than the lifespan of the MTL itself and would already be familiar to programmers who are familiar with the chosen programming language.

Our problem statement is in a single sentence:

Designing a MTL from scratch is a time consuming endeavour that requires re-implementation of many language features that already exist (mature and thoroughly tested) in other programming languages.

1.2 Approach

We will explore an alternative but equally valid method of developing a MTL, namely developing it as an Domain Specific Language (DSL) within the context of an existing general purpose programming language. This kind of DSL is often referred to as an Domain Specific Embedded Language (DSEL) [18] and it's essentially an advanced form of the Application Programmer Interface (API).

In our opinion, the most appropriate way to explore this approach is to perform a case study, which in this case means the implementation of a MTL using this approach. In order to do this we'll need to pick a general purpose programming language to base our MTL on, which we will call the host language. We will need to establish some criteria for picking our host language and examine its facilities for DSEL creation. Additionally we need to perform a survey of MTL characteristics and decide upon some requirements for our MTL.

The main advantage and in fact the goal of this approach is that it allows us to focus on the model transformation specific aspects of the language as described in Chapter 3 rather than on aspects that are commonplace in other computer languages. Performing the case study will allow us to determine to what extend this goal can be reached and what other (unforeseen) advantages our method yields. The obvious disadvantage of this method is decreased flexibility in defining our language. The case study will allow us to determine to what extend this actually is a problem and what can be done to counter it.

1.3 Related Work

The language we will be developing is based on the declarative Mistral [21] language for Model transformations to a large extend. The topological sort algorithm used to determine trace dependencies introduced by Mistral will be at the heart of our language as well. Other important influences are RubyTL [19] and RGen [29]. RubyTL was the first MTL that we know of that is defined as an DSEL and RGen is a more recent project that does the same thing but also broadens the scope to a more complete MDE environment. Incidentally, both of these projects use the Ruby programming language as their host language.

1.4 Contributions

This section summarises the contributions made by this research. They are described in more details in Chapter 8.

- **Architectural Improvements** TRans's architecture offers a very clean and extensible approach to model transformation. It allows the model querying and instantiation mechanisms to be a separate DSEL that is all but completely decoupled from the actual model transformation DSEL.
- **Comprehensive Approach** Because TRans is so indifferent to the interpretation of the model concept everything can be a model whit its own mechanisms for instantiation and querying. This means that any type of computer data, like for instance a code repository can be wrapped and used as a source or target model.
- **AOP Functionality for Model Transformation** TRans features the ability to apply Aspect Oriented Programming in its model transformations. This ability is especially powerful since the full featured Ruby language can be used to write advice.
- **Complete Survey of DSEL Techniques** This report presents a very complete survey of DSEL creation techniques in the context of the Ruby language. Many of these techniques have been applied for the creation of the TRans language which leads to it being a more concise DSEL for model transformations than RubyTL.

1.5 Outline

We start by provide a more complete analysis and decomposition of the problem and the approach we just introduced in Chapter 2. To provide context, we will describe different aspects of MDE and directional declarative MTL's in particular in Chapter 3. Chapter 4 will give more context on the Ruby programming language and its facilities for creating DSEL's in particular. If the reader is already familiar with MDE and DSEL creation in Ruby, these chapters can be skipped. Chapter 5 introduces our case study of developing of a MTL as a DSEL, which we will call a Model Transformation Embedded Language (MTEL). The architecture and design of the language is detailed in Chapter 6 in the form of development guidelines, dependencies, a logical overview, class diagrams and sequence diagrams. We evaluate the usability, flexibility and performance of our language in Chapter 7. Finally, we present our conclusions in Chapter 8. Possible paths for reading through these chapters are depicted in figure 1.1.

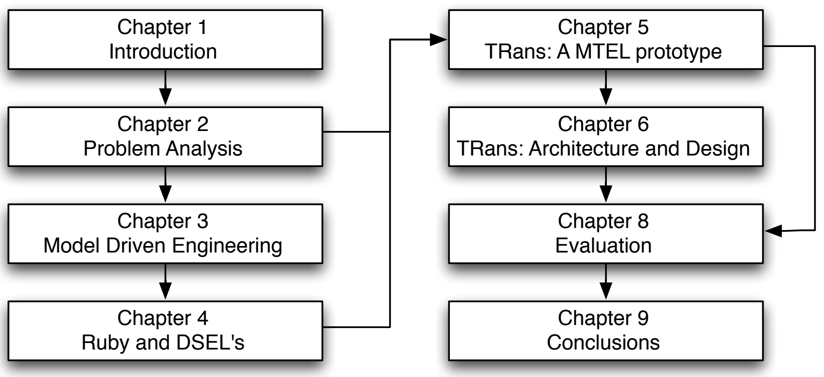


Figure 1.1: Reading guide for this report

Chapter 2

Problem Analysis

In this chapter we decompose the problem stated in our introduction into subproblems and describe our approach to solving to those problems on a high level of abstraction. First, lets reconsider our problem statement:

Designing a MTL from scratch is a time consuming endeavor that requires re-implementation of many language features that already exist (mature and thoroughly tested) in other programming languages.

The obvious solution to this problem is the reuse of language features from an existing programming language for the development of a MTL. This approach provides the MTL with mature and tested implementations of those language features which improves its quality and allows more focus on the development of model transformation specific language features. We refer to the programming language that is reused for implementation of the MTL as the host language. We have identified five ways in which a programming language can be reused to create a MTL:

- Design the MTL from scratch, reusing source code from the host language implementation for the common features.
- Refine the host language implementation to be model transformation specific.
- Design the MTL as an extension to an existing programming language that allows this.
- Design the MTL from scratch and compile it to the host language, leaving common language constructs intact, essentially forwarding them to the host language.
- Design the MTL in the host language as an Domain Specific Embedded Language (DSEL).

The possibilities are ordered from maximum flexibility and minimum reuse to minimum flexibility and maximum reuse, meaning that they require a descending amount of effort for the development of the MTL. This inverse correlation between reuse and flexibility might seem counter intuitive. However, by reuse we mean reusing parts of a *single* host language. In this case there is only a single alternative for every component you decide to reuse and these components contain certain decisions in their implementation that you won't be able to make yourself.

We decided to implement our MTL as a DSEL in the our host language because this approach allows maximum reuse and requires the least amount of effort. This means that if this is a viable solution, it is the most desirable one. After making this choice, our new problem is how to develop a MTL as a DSEL in a host language, enabling maximum reuse of the host language features, and requiring a minimum amount of compromise in our MTL because of the lack of flexibility. This problem raises the follow questions:

Q1 What host language should we use and because of which requirements?

Q2 Which facilities in the host language can be used to create a DSEL?

Q3 What features do we want our MTL to support in order to make it a valid proof of concept?

Q4 To what degree where we able to achieve the reuse we where aiming for?

Q5 What advantages and disadvantages did the achieved reuse yield?

Q6 How does our implementation compare practically to implementations using the same approach?

The next sections present high level descriptions of our solutions to each of these research questions.

2.1 Host Language Choice and Features (Q1 and Q2)

In order to be able to develop a viable MTL as a DSEL, we chose the following requirements for our host language:

- It should have a flexible syntax that isn't exotic in the context of model transformation.
- It should provide extensive features for the creation of DSEL's. In particular closures and operator overloading in order to implement OCL.
- It should provide syntax for declarative and functional programming as these paradigms are common in model transformation languages.
- It should have a mature standard library in order to deal with data structures, strings and file formats.

It turns out that even for these straightforward requirements, there is only a single language that's the obvious choice. We chose Ruby as our host language because it is, to the best of our knowledge, the only mature language that meets all these needs and because it has proven itself repeatedly in the area of DSEL creation. Examples of this in the field of MDE are RubyTL [19] and RGen [29]. We provide an extensive investigation into Ruby DSEL creation in Chapter 4, in order to determine exactly which facilities it offers for this purpose.

2.2 MTL Features (Q3)

In order to establish a feature set for our language that makes it valid proof of concept we performed a literature study on MDE and more specifically an analysis of commonalities and variability's in model transformation specific language features. This analysis is presented in Chapter 3 and a selection of these features is presented in Chapter 5. Aside from model transformation specific language features, the MTL will also incorporate many common features that it borrows from the host language such as control structures, expressions, a type system, arithmetics, modularisation and the standard library. We spend less time on these because they basically come for free when choosing this approach.

2.3 Evaluating the degree of reuse (Q4)

Reuse is a simple concept; develop (sub)systems and use them across multiple, larger (sub)systems [2] [11]. In the case of DSEL creation, the (sub)systems we plan to reuse are classified as language features, in a very broad sense. A language feature can be a specific language construct like exception handling but also something with a broader scope like a parsing or a method dispatch mechanism. What we're really doing when developing a DSEL is reusing an entire existing language and building a new language (the DSEL) on top of it. In this sense, the DSEL is an extension of the host language, reusing every aspect of it. Even while every aspect of the host language is going to be present in the DSEL, it is possible to hide some aspects or discourage users from accessing them. In our opinion, the best way to assert the degree of reuse that can be achieved is to develop a prototype. Chapters 5 and 6 describe the development of this prototype and Chapter 7 evaluates the degree of reuse that was achieved that needed to be made. It is hard to state exactly the decrease in complexity and effort by reusing an existing language because there are so many variables involved. For a proper comparison you need two functionally equivalent MTL's, one developed as an DSEL, and one developed in the traditional way. Since we don't have these two languages, the motivation of the statements in our evaluation can only be a logical one.

2.4 Advantages and Disadvantages (Q5)

If we establish that a certain degree of reuse can be achieved, this will directly or indirectly impact a lot of characteristics of our language implementation in comparison to a language developed in the traditional way. We expect that the characteristics that will be impacted are the following:

- **Implementation effort** The *effort required for the implementation of our MTL* is expected to decrease with an increased degree of reuse. This statement only assumes that reusing something requires less effort than recreating it, something we feel is safe to say. This decrease in effort is considered desirable.
- **Flexibility** We define flexibility as the *degree of freedom in implementing the MTL* that the language designer has. When language features are reused from the host language, flexibility is expected to decrease. This assumption is based on the notion that reused language features inevitably contain design decisions. When unable to make all design decisions that go into a language yourself, your degree of freedom in creating that language is decreased. This decrease in flexibility is considered undesirable.
- **Familiarity** Familiarity is considered as *the amount of knowledge available about the MTL amongst new users.* We expect familiarity to increases with increased reuse, given that features are reused from a well established language such as Ruby. Familiarity is considered desirable.
- **Maturity and Reliability** Maturity and reliability are defined as the *density of defects in the MTL.* We expect it to increases where reuse is increased as well, given that features are reused from a mature reliable language such as Ruby. Maturity and reliability are considered desirable.
- **Library Availability** When a language is developed as a MTEL, host language libraries become available to that language as well. We consider the *ability to use libraries when writing model transformations* desirable.
- **Maintainability and adaptability** Maintainability and adaptability the *effort required for respectively repairing defects and making changes* to a system. We can assume that a small code base of a certain quality is more maintainable and adaptable than a much larger codebase of equal quality because changes in large codebase potentially have more impact. We therefore assume that these characteristics are increased for the language implementation where implementation effort is decreased. Obviously, maintainability and adaptability are desirable qualities of a system.
- **Superfluous functionality** When a language is developed as a MTEL, you inevitably reuse language features you don't actually want to reuse. Some of those features can be useless or even harmful in a model transformation domain. The *existence of useless or harmful features in a system* is considered undesirable.

Chapter 7 evaluates whether or not these characteristics where actually affected in the ways we expected. If so, this defines in which situations the DSEL approach is preferable based on which characteristics are important. Figure 2.1 shows a diagram of the different characteristics we defined, how we expect them to be affected by reuse or one another and whether we consider it desirable for them to increase.

2.5 Comparison to other implementations (Q6)

This research is not the first effort to develop a MTL as DSEL. RubyTL [19] is another one. It is important do evaluate the differences between our implementation and theirs. We will do so in Chapter 7.

2.6 Conclusions

In this chapter we analysed our problem and defined our research questions. We also immediately answered one of the research questions because there really is only one obvious answer. For the other research questions we formulated our strategy towards answering them throughout the rest of this report.

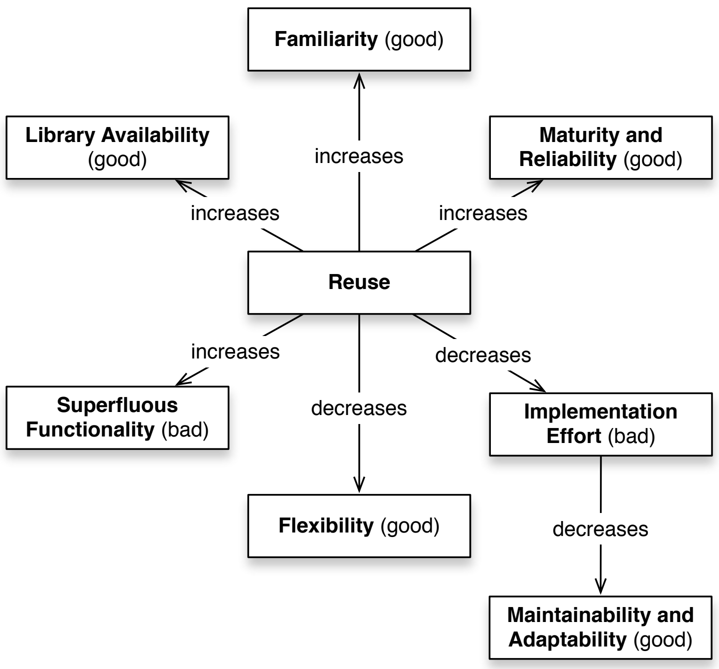

Familiarity (good)
Library Availability (good)
Maturity and Reliability (good)
increases
increases
increases
Reuse
increases
decreases
decreases
Superfluous Functionality (bad)
Implementation Effort (bad)
Flexibility (good)
decreases
Maintainability and Adaptability (good)

Figure 2.1: How we expect different characteristics of our implementation to be affected by increased reuse

Chapter 3

Model Driven Engineering

This chapter is an introduction intro the domain of MDE. Section 3.1 introduces the central concepts in MDE in order to provide context. Section 3.2 provides a formal definition of the most important concept of model. Sections 3.4 and 3.5 respectively describe the advantages that MDE promises to provide and the problems that are associated with it. Section 3.6 describes current tools and languages. Section 3.7 describes the features of these languages with special emphasis on commonalities and variabilities in order to identify important features for our own Model Transformation Language (MTL).

3.1 Conceptual Overview

The main concepts in MDE are models, meta-models and transformations. Models are the first class entities of any MDE environment.

A model is a representation of its subject, describing only the relevant aspects of it and omitting everything else. While a model's subject is *represented* by the model, the model can be *interpreted* to obtain information about its subject. While we prefer this terminology, the relation between a model and its subject is also often described as the model *modelling* its subject and the model *being an abstraction of* its subject. It's possible to have multiple models of the same subject, each describing different, possibly overlapping aspects. Additionally, it is possible for a model to be the subject of another model itself. The relation between a model and its subject is shown in figure 3.1.

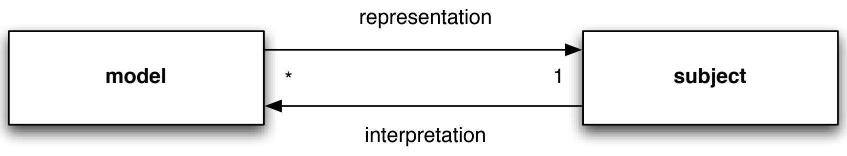


Figure 3.1: The relation between a model and its subject

Meta-models essentially define a subset of all possible models by specifying constraints on the them. In MDE, meta-models are often models themselves, and that way a meta-model for meta-models can be defined, which is called the meta-meta-model. A MDE environment has just one of these meta-meta-models, defining the subset of all models that classify as meta-models. A way to look at this is that the meta-meta-model is the type or class of meta-models, which in turn are the type or class of individual models. Another common terminology is that each model is an instance of its meta-model and that every meta-model is an instance of the meta-meta-model. Figure 3.2 shows these relations.

As mentioned earlier MDE is also concerned with transformations between models. These transformations are expressed in a Model Transformation Language (MTL). Section 3.7 describes many aspects of these MTL's. In most MDE toolsets, models are used to represent these model transformations, this is a clean approach that allows transformation of transformation definitions. In this case the syntactic part of the MTL is a meta-model. Figure 3.3 provides an overview of this process and its terminology.

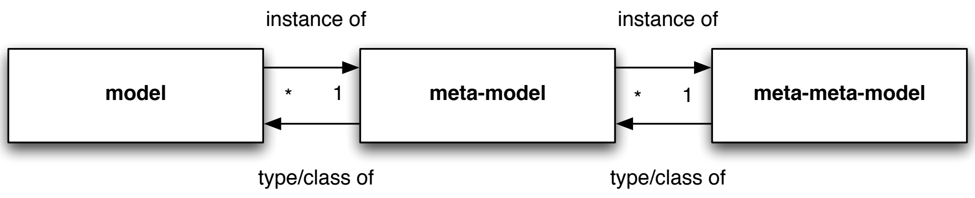


Figure 3.2: Models and their meta-models

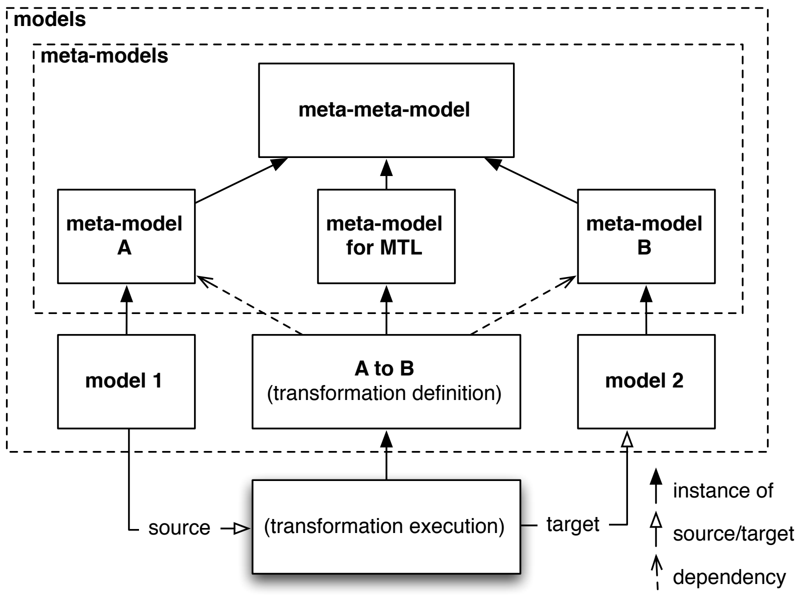


Figure 3.3: An overview of the process of model transformation and its terminology

While theoretical understanding about modelling and transformations is vital for the development of a MDE environment, models are a very simple concept from a pragmatic or implementation point of view. In this context a model can simply be viewed a simple data structure, traditionally often a graph with a map of key/value pairs associated with each node. These kinds of models are especially well suited to describe structural aspects of software. UML [17] is a good example of this.

Apart from model to model transformation, it's also important to perform transformation between models and other artefacts. These other artefacts will often include graphical representations and textual representation such as code. We call creation of models from other artefacts *reading*, the most common applications of which are loading a model from an editable graphical representation or other serialised form. We call creation of other artefacts from models *writing*, the most common applications of which are saving a model to an editable graphical representation or other serialised form and code generation.

3.2 Definition of Model

The first defining characteristic of a model is that it is a relative concept [21]; every model exists in *relation* to its subject. The model represents its subject and can be interpreted to obtain information about it. The model is therefore an abstraction of its subject, describing relevant aspects of it and omitting everything else.

Model interpretation can be described by the mapping of a model to the set of all possible subjects it could represent. This set of subjects is the subset of all possible subjects that have the characteristics that the model describes in common. When you look at the floor plans of a house for example you can derive certain aspects of it (room configuration, sizes, etc.) but others are still variable (location, materials, etc.). In this example the floor plans are a model, and all possible houses they could result in are the set of subjects the model could represent. From the resulting set of subjects, a single subject can be chosen based on its non-common (variable) characteristics.

Model interpretation is reversible which means a model can be derived from any of its possible subjects, we call this process *subject representation.* The other way around however, a single definitive subject can not be derived from a model without extra input. An exception to this rule only arrises if a model describes every single aspect of its subject. We argue however that such a model isn't a model at all because it isn't an abstraction of its subject. This point of view is reflected in our final definition of model. Furthermore we can classify model interpretation as an non-destructive process and subject interpretation as a destructive process.

Another import aspect of a model is that it's expressed in a modelling language. The modelling language often consists of formal syntactics in the form of a meta-model and informal semantics. Most of the rest of this report is devoted to the semantics of model transformations, which are a specific class of models.

Based on the literature [21] and these observations, we chose the following definition of model:

> *A model is a representation of its subject expressed in a modelling language. It can be interpreted to derive a proper subset of its subject's characteristics.*

3.3 MDE or MDA?

MDE was proposed as a generalisation of MDA, the original standard from the OMG [26]. The main difference is that MDA focusses on Platform Independent Models (PIM) and Platform Specific Models (PSM) and also on the set of OMG standards, among which is UML. Also, MDE focusses on the development process much more than MDA does. Since MDE is broader in scope it relates more closely to a general purpose MTL than MDA.

3.4 Promises of MDE

The original research on what was then called MDA focussed on a specific case of separation of concerns (PIM/PSM). They where aiming to separate the concerns of functionality and implementation platform. MDE is an effort to tackle the problem of separation of concerns in a more general sense. Promises of increased maintainability and better communication are closely related to proper separation of concerns. If MDE will ever achieve true separation of concerns in complex software systems it will deliver on these other promises as well.

3.4.1 Separation of Concerns

The obvious way MDE tries to achieve separation of concerns is by delegating each concern to a separate model. The challenges arise when attempting to assemble models into intermediate artefacts and ultimately into working software (transformation). A balance has to be found between describing system knowledge with models and embedding it in the transformations themselves. To have a separate model for every bit of system knowledge would achieve full separation of concerns but would result in a very complex system of models and transformations which would be hard to standardise. Embedding too much system knowledge in the transformations leads to concern tangling and decreases reusability of transformations. The current approach to this dilemma is having a different model driven approach for each system, further decreasing reusability of models and especially transformations.

3.4.2 Better communication

Separation of concerns leads to better and clearer code and models that in turn leads to more effective communication through code and models. Apart from that, there is also the separate notion of models being more easily readable by nature than code and thus being more adequate tools of communication. However, it might very well be that this notion has risen from the fact that models generally contain a lot less knowledge and complexity than code does. Models are inherently decoupled from their representation though, which opens up interesting prospects of communication through possibly different representations of models.

3.4.3 Maintainability

Separation of concerns is the most important aspect that makes a system maintainable. Change decisions target functionality rather than code and by localising that functionality to a specific part of the system change impact can be limited, thus achieving increased maintainability. The rest of this claim is based on improved communication through models, which a questionable claim by itself. Easily readable artefacts would however facilitate the efficiency of developers who are inexperienced with those artefacts. When it's easy to go back to a system after an extended period of time and make changes or bring in new people to do it, that system is more maintainable.

3.5 Problems of MDE

The main problems of MDE are those of the models themselves. A unified set of modelling languages that can completely and precisely describe any software system simply isn't available, the most significant attempt is UML. Another problem of MDE is the poor and often clunky tool support that is solely due to lack of adoption.

3.5.1 Behavioural Modelling

Current software modelling practices, namely the use of UML, are fairly adequate at modelling the structural aspects of a software system. However, there is a big gap as far behavioural modelling is concerned. An important reason for this is the lack of formal semantics for modelling languages. While over half of the UML model types aspire to some kind of behavioural modelling, interpretation and interaction of these types of models is often not an exact science. Especially the diagram types that specify by example like the sequence diagram are completely unusable in a MDE environment because they don't provide a full specification that includes all corner cases. MDE approaches are currently focussed on the structure of software systems, in which the behaviour is often defined manually using programming code. Behavioural models with strictly defined semantics will have to be developed in order to overcome this problem.

3.5.2 Modelling of Dynamic Languages

Another shortcoming of the dominant modelling language UML is that there are absolutely no facilities for modelling systems that make use of dynamic languages. This deficiency is related to the behavioural modelling gap because part of the behaviour of a dynamic program can be to change or extend its own structure. Runtime specialisation and creation of classes and runtime (re)definition of methods are just some of the things UML can't model at all. UML is not very unified in the sense that its main focus on effectively modelling Java systems and after noticing the package diagram one might even argue that Java Modelling Language would be a more appropriate name for UML.

3.6 Tool Support

Most MDE tools are build on the Eclipse Modelling Framework (EMF), examples of this are ATL and Borland Together. The EMF platform is a modelling framework and code generation facility for building MDE tools. EMF uses XMI to represent models and provides a basic graphical XMI editor along with tools for loading the models into a Java runtime environment where they can be traversed and updated. The meta-modelling architecture of EMF is called ECore. In this section we provide background information on some of the model transformation languages already in existence that are relevant to our research.

3.6.1 Query/View/Transformation (QVT)

QVT [27] is a standard created by the Object Management Group (OMG) of which several partial implementations exist, the most notable of which are SmartQVT and the implementation in Borland Together. The QVT standard actually consists of three DSL's named QVT Relations, QVT Core and QVT Operational Mappings. Relations and Core are declarative languages which both are extended by the Operational Mappings language which also includes some imperative features.

3.6.2 ATLAS Transformation Language (ATL)

ATL [5] is a slightly more pragmatic approach to model transformation because it originated as an implementation instead of as a standard. It originated later than QVT and has always supported many-to-many transformations. Like QVT it operates on models that comply to the MOF meta-meta-model.

3.6.3 RubyTL

RubyTL [19] is a MDE approach which uses Ruby DSL's for model transformation and code generation. It is distributed as part of the Agile Generative Environment (AGE) which is an Eclipse environment for MDE. It supports ECore and EMOF which makes it compatible with other MDE solutions based on Eclipse.

3.6.4 RGen

RGen [29] is a much more recent solutions to MDE using Ruby tooling. It was developed for commercial use and also integrates with ECore and provides code generation. It was developed in isolation from RubyTL and has focusses more on meta-modelling rather than transformations.

3.7 Features of MTL's

This section describes the commonalities and variability's of MTL's. These are things every MTL designer must think about in order to specify a new MTL. Many parts of this section where taken from literature [9] but some where stumbled upon during development for this thesis. This section is focussed on declarative directional rule-based MTL's because they are the most common and are the focus of this research.

3.7.1 Specification Mechanism

The specification mechanism refers to the language used to specify the actual model transformations. While transformations in most MTL's are specified using a custom syntax, it is also possible to build on a existing language that has a flexible syntax. An example of this is RubyTL [19], which builds on the general purpose programming language Ruby. Also, it's easy to imagine specifying a MTL in a general purpose data definition language like XML as another alternative in this area.

3.7.2 Rule, Source and Target Multiplicity

The basic building block of a declarative MTL is the rule. It specifies a mapping between source model elements and target model elements. Each rule has zero or more source expressions. Source expressions are evaluated to produce a result that functions as input for the rule. While this result is often a bunch of model elements it can also be something simple like a string or a number. It is possible to have multiple source expressions and combine their results somehow. The most common combination method is a cartesian product, but other possibilities are a set union, a set intersection or having a configurable combination method for each rule. The more elegant and powerful solution however, is to have a single source expression for a single rule and allow these set operations and the querying of multiple models in that single expression. Another valid but uncommon possibility is to allow rules without a source expression at all, in which case the rule would have no input. The single rule, single source alternative can however easily mimic this behaviour by having the single source return a void object (nil or null in most common languages).

While the source expression of a rules functions as its input, the target of a rule functions as its output or result. It creates, changes or deletes an element. Usually, target(s) are evaluated once for each item in the input. While a rule without any targets would make no sense, both allowing just a single one and allowing multiple are valid approaches. Allowing only a single one does not result in decreased functionality compared to the alternative and is thus the cleaner approach. Experience has thought us however that allowing multiple targets for a single rule is often convenient, because it can reduce redundancy in source expressions without the use of helper rules.

3.7.3 Source, Target or Hybrid Models

The models a transformation is concerned with often have specific roles. Some models can serve strictly as input, which means they can be queried(read) but can not be changed(written). Other models might serve strictly as output models, which means they can are generated(written) during transformation but may never be queried as the source of a rule. A MTL can also allow a hybrid form, which is model that can be both queried and changed. However, this is a controversial feature because it inevitably leads to non-determinism in declarative transformations. Hybrid models are desirable though because they facilitate in-place transformation, which allows for much simpler transformations for model refinement. An alternative way to achieve more simple model refinement transformations is to include powerful model element copying features in the MTL.

3.7.4 Tracing Facilities

The advantage MTL's have over the more common tree-based transformation languages like XSLT [6] is the ability to transform graphs that have cycles. MTL's accomplish this by offering tracing facilities. The basic idea is to keep a record of all the elements that where created and what source data was used to create them. This information can then be queried in order to create the target element.

A classic example is the transformation of 'properties' that have a 'class' as owner to 'columns' that have a 'table' as owner. When transforming 'columns' to 'properties' in this case, the transformation can use trace data to find the 'table' that was generated from the 'class' that owns the 'property' to serve as the owner of the 'column'. Figure 3.4 shows that the columns could have never found its owner without the trace information.

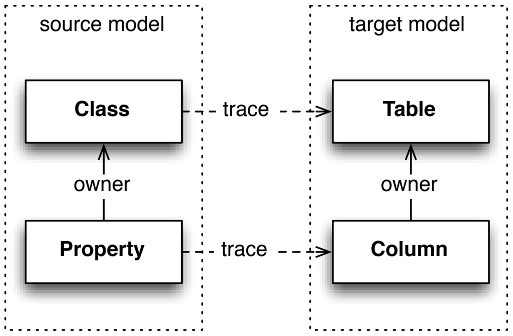


Figure 3.4: Example of keeping a trace information

3.7.5 Transformation Ordering

Even though declarative transformation don't have explicit rule ordering, rules do sometimes need to be evaluated in a specific order to complete a transformation. The case presented in Section 3.7.4 is an example of this. The rule that transforms 'classes' to 'tables' needs to be evaluated first in order for the rule that defines the 'columns' to find those 'tables' using trace information and set them as the 'column' owner. In a purely declarative transformation language, the transformation engine is responsible for determining this ordering. This ordering can be determined in an extra pass that is done before the actual transformation pass, but can also be done on the fly. In the second case the transformation can just start with the evaluation of any rule and evaluate other rules as they are needed.

While it is common to evaluate rules atomically (entirely or not at all), a more powerful approach is to do the same for rule targets or even target elements. These approaches allow tracing dependencies between targets of the same rule or between target elements of the same target respectively. An example case in favor of the approach that allows dependencies between target elements of the same rule target is the transformation of an inheritance hierarchy using a single rule.

Though we can transform entire tree structures in a single rule with this last approach, we can't do the same for graphs that contain cycles. This can however be done by forgetting about ordering altogether and propagating all the trace information queries to an extra pass that is executed after the main transformation pass. However, this approach poses a problem when the transformation wants to traverse the result of a trace query.

Transformation Order Control

When non-determinism is introduced into a model transformation because it manipulates a hybrid model, the need for rule order control arises. Several mechanisms have been invented to allow this. The first and simplest is to just specify the order in which rules have to be evaluated. This is very much at odds with the declarative nature of these transformations and it is not very well maintainable. A slightly more desirable approach is to group rules and specify the execution order of those groups, a mechanism often referred to as transformation steps. However, this approach suffers from exactly the same deficiencies as the first one. A less flimsy and more maintainable solution is to allow rule dependencies to specify ordering only where necessary. Nonetheless, all these alternatives are considered harmful because they severely interfere with the declarative nature of transformations.

3.7.6 Source and Target Model Multiplicity

All MTL's need to decide how many input and output models to accept and generate. The possibilities are to support one-to-one, many-to-one or many-to-many transformations. The one-to-one alternative is very constricting and so not very common, it means the MTL would never be able to combine multiple models into one. Supporting many-to-one transformation is the cleanest way of opening up all possible transformation scenarios because it can achieve many-to-many transformation by having a separate transformation for each of the output models. Many-to-many transformations open up possibilities for tangling within transformations because multiple unrelated models could be generated by a single transformation. However, this is the preferred alternative for most MTL's because it allows generation of side-products for a transformation like log and trace data. Especially trace data is crucial for implementing incremental or partial transformations.

3.7.7 Source Expression Language

This part of the MTL is subject to reuse and standardisation because it is far more tightly coupled with the modelling standard used than with the MTL itself. A mature MTL should make it easy to swap out one expression language for the other and could even allow multiple languages to be used in parallel. The most common standard in this area is OCL, which is a standard from the OMG [4].

Object Constraint Language (OCL)

OCL is a standard that was developed by the OMG to query UML models. Apart from some basic types and operation, its distinguishing features are the support for multiple types of collections and set operations on them. Included in these set operations are existential and universal quantifiers with closure support, a very useful feature in this domain better known from SQL and dynamic programming languages.

XQuery

XQuery [3] is not commonly applied in the area of model transformations. It does however deserve a mention as the dominant standard in the area of document querying. While XQuery is a standard for querying XML documents that are essentially trees, it contains several features that could drive innovation in the area of model querying, the most notable of which being XPath.

3.7.8 Helpers Functions

Most MTL's have support for helper functions, often referred to as helper rules. These are totally different from the rules discussed earlier and hence the term function is preferred. While rules are always evaluated once and just once, helper functions can be invoked at will from these rules. Like any function they serve to contain functionality that is used in multiple places and that would otherwise have to be duplicated. The other important reason to have helper functions is to make a MTL Turing complete by allowing recursion.

3.8 Conclusions

While the research area of MDE is quite extensive, only several concepts are really well established. The well established commonalities in the area of model transformation are the concepts of source and target models, transformation and helper rules, rule ordering mechanisms, tracing mechanisms and declarative definition of transformations. Apart from the existence of these concepts in almost every MDE approach, there is a lot of variability in the interpretation of these concepts among different MDE approaches.

Chapter 4

Ruby and Domain Specific Embedded Languages

In this chapter we discuss the aspects of Ruby that make it great for the development of Domain Specific Embedded Languages (DSEL). We start by providing introductions into DSEL's and some of Ruby's structural and syntactical features. Since the most defining characteristic of Ruby is its marriage of different programming paradigms, most of the rest of this chapter is structured into the programming paradigms that Ruby borrows its language features from. These chapters focus specifically on how these language features can be applied to develop DSEL's.

Section 4.3 provides some introduction into the Ruby syntax. It focusses on syntactic aspects that make it great for DSEL development. For a broader and more thorough introduction we refer to the online documentation [25]. Sections 4.4, 4.5, 4.6 and 4.7 respectively describe object oriented, prototype oriented, functional and aspect oriented features of the Ruby language that can be used for DSEL creation.

4.1 Historical Context

Ruby [25] is a dynamic, reflective, general purpose, object oriented programming language. Ruby originated in Japan where it was developed by its creator Yukihiro 'Matz' Matsumoto during the mid 1990's. While Ruby's internals bear the most resemblance to Smalltalk [1], the Perl [14] programming language has served as a strong inspiration for the exterior features of Ruby. Matz had already picked the name Ruby over Coral before any code for the language was written. It was realized later that the Ayurvedic birthstone system designates the pearl as the birthstone for the month June and the ruby as the birthstone for the month July, implying Ruby as the successor of Perl. Ruby received a lot of attention in recent years because it is the foundation of the web development framework Ruby on Rails [10].

Ruby has never had an official specification. The original C implementation created by Matz is regarded as the authoritative specification for the language. Recently, third parties have started creating their own implementations of Ruby. Rubinius [28] was the first alternative implementation. It uses a custom virtual machine and is supposed to be faster and more reliable than the original implementation. However, it has yet to gain popularity. The RubySpec [22] project originated from the Rubinius project and is an executable specification of the Ruby language that was written in Ruby itself. RubySpec is now the standard for checking compatibility between different implementations. Sun is supporting the development of JRuby [13], which runs on the Java Virtual Machine (JVM) [24]. The main advantages of JRuby are its speed, its ability to integrate with Java libraries and its ability to deploy Ruby on Rails web applications to Java application servers. Microsoft also announced its own Ruby version for the .NET framework [8] called IronRuby [7], which is currently in alpha.

4.2 Domain Specific Languages

A Domain Specific Language is a computer language that is tailored to a specific application domain. It allows programs written in it to be more effective and concise within this application domain. This is accomplished by providing these programs with an implicit context that would have to be provided explicitly if the program was written in a general purpose computer language. While a DSL can be a language in the traditional sense with its own parser and compiler or interpreter, we are focussing on Domain Specific Embedded Languages (DSEL). A DSEL is a DSL that is embedded in a suitable general purpose computer language, borrowing common language features from it and adding language features that are lacking. A DSEL can be embedded in its host language at compile time [18] or at runtime [12]. We are focussing on the variant that is embedded at runtime, which is essentially an advanced form of the Application Programmer Interface (API). The distinction between API's that can be called DSEL's and API's that can't is very vague. We use the term DSEL very broadly to describe any API that has DSL-like properties. Embedding at runtime means that the DSEL has no control over its own compilation and thus that only language constructs of the host language can be used to express programs in the DSEL. Even though this approach seems constricting at first, this chapter will show that a well designed host language can require conformance from a DSEL while still being very flexible.

4.3 The Ruby Syntax

The Ruby syntax focusses on flexibility and conciseness. Many of the core constructs in Ruby can be expressed in several different ways, allowing it to mimic other computer languages. The code sample in figure 4.1 shows two pieces of code that do exactly the same thing using different programming styles, both are valid Ruby code but the first shows syntactical similarities to functional programming and linux shell scripting while the second one is similar to C and Java. This example shows two different syntaxes for both code blocks and string literals and also shows the optionality of both parenthesis for method invocation and the semicolon for statement termination. While these are only minor exterior provisions in the Ruby parser, it shows a design philosophy that prefers to allow rather than to require.

```ruby
# Print 'Hello World' three times
3.times do
  p 'Hello World'
end

# Exactly the same, but different
3.times {
  p("Hello World");
}
```

Figure 4.1: Example of Ruby syntax

Another example of this is illustrated by the code sample in figure 4.2. The first few lines define a method that takes one required argument and one optional argument that defaults to an empty hash. The last few lines show different ways in which this method can now be invoked. The important thing to note here is that wherever the hash literal character '=>' is used in a method invocation, Ruby will attempt to parse the rest of the argument list as a hash and pass it as a single argument, even though they are not grouped explicitly. While allowing to omit these hash grouping characters is again only a minor and rather quaint bit of syntactic sugar it allows concise syntax and flexibility in method definitions. The colon characters in this example are simply the literal syntax for creating a symbol. Symbols are similar to strings and are often used as the keys in a hash. Symbols don't contain whitespace and are more lightweight than real strings because they are represented internally by an integer. The last line shows an alternative syntax for hash literals introduced by Ruby 1.9.

The ability to express language constructs in different ways is especially useful for the creation of DSEL's using Ruby. It allows the user (programmer) of the DSEL to pick an alternative that is suitable in the context that the DSEL provides to improve readability. Please note how closely the last line of the code sample in figure 4.2 mimics SQL, a DSL for querying relational databases, while still being Ruby.

```ruby
# Define the find method
def select from, options = {}
  # Perform sql query
end

# Invoke method with different arguments
select 'products'
select 'products', :where => 'price > 100'
select 'products', :where => 'price > 100', :order_by => 'name'

# Using alternative syntax for hash literal (since Ruby 1.9)
select 'products', where: 'price > 100', order_by: 'name'
```

Figure 4.2: Example of using named arguments

4.4 Object Oriented Programming Features

Ruby is a purely object oriented language. Without exception, everything in Ruby is an object and there are no primitive types. This choice provides a lot of simplicity and consistency throughout the language and eliminates the need for hacks such as Java's auto-boxing. It means that all literals are objects as well and methods can be invoked on them. Some code examples of this that are commonly used to advocate the Ruby language can be found in figure 4.3. The ability to invoke methods on literals is directly derived from the ability to instantiate certain types of objects using a literal syntax. Besides the strings and integers, the Ruby language provides literal syntaxes for instantiation of floats, character codes, symbols, ranges, arrays, hashes, regular expressions and even shell commands.

```ruby
5.times do
  p "World! Hello".split.reverse.join(' ')
end
```

Figure 4.3: Examples of calling methods on literals

Another important thing to realize about Ruby is that every object has a class and these classes themselves are also objects. Classes are instances of the class called Class which in turn is an instance of itself, terminating the instantiation hierarchy. Ruby also has the concept of modules, which are basically classes that cannot be instantiated. Modules are often referred to as abstract classes in other languages. While Ruby does not support multiple inheritance and so every class has a single superclass, multiple modules can be included into the definition of a class. This behavior mimics multiple inheritance while eliminating some of the inconsistencies it causes. From here on we will refer to the class called Object, the class called Module and the class called Class as the meta-object, the meta-class and the meta-module respectively. We refer to the set of all three as the plural meta-objects because they all inherit from the meta-object and to avoid the overloaded term meta-classes. We will define meta-programming as programming using these meta-objects. The meta-module is located in the inheritance hierarchy between the meta-class and the meta-object, which signifies the root of inheritance hierarchy. All these relations are shown in figure 4.4 as the results of calls to the methods class and superclass, respectively defined by the meta-object and the meta-class.

Figure is 4.4 particularly useful for understanding the instantiation and inheritance hierarchies that where mentioned in the previous paragraph. The instantiation hierarchy is signified by the 'class' relations in this figure, while the inheritance is shown by the 'superclass' relations. Please note that the instantiation hierarchy exists throughout all Ruby objects while the inheritance hierarchy only applies to classes. This model allows Ruby to use a very simple mechanism for method dispatching: go one step up the instantiation hierarchy and then start looking for the method up the inheritance hierarchy. If the original method isn't found the same mechanism is used to look for a method named 'method_missing' which is implemented in the meta-object (the root of the inheritance hierarchy) to throw a runtime 'MethodNotFound' exception.

4.4.1 Open Imperative Class Definitions

Even though simple Ruby class definitions look a lot like declarative class definitions in for example Java, class definition in Ruby is actually an imperative process. This makes it possible to use control structures

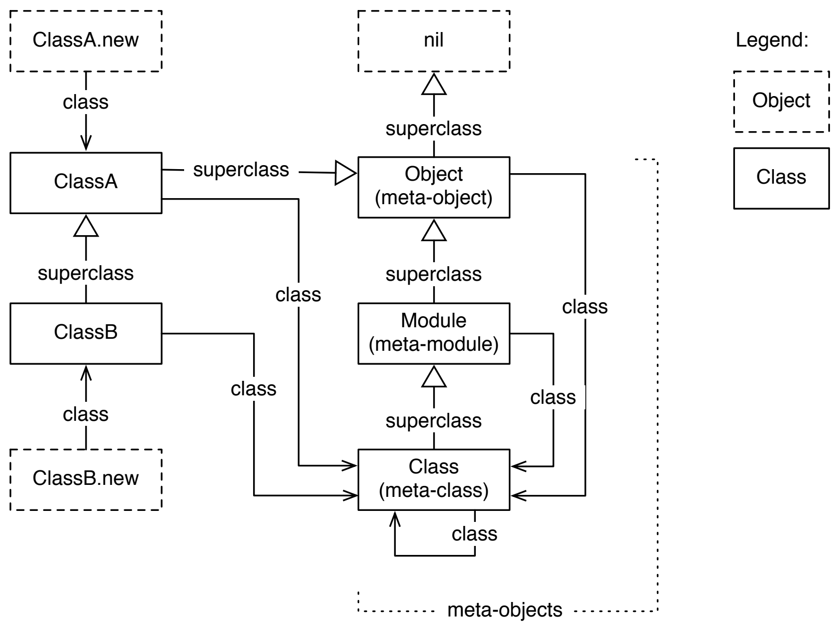


Figure 4.4: Meta-objects, regular objects and the relations between them

like loops and conditionals in class definitions and even call methods that define other methods for you. Figure 4.5 shows some sample code using 'ActiveRecord', a popular Object Relational Mapper (ORM) written in Ruby. Inheriting from the 'ActiveRecord::Base' class alone adds many methods to the 'Person' class and objects of that class by use of the 'Class.inherited' hook. Because the body of the method definition is just imperative Ruby code in the scope of the Person class, class methods for describing associations and validations added by 'ActiveRecord' can be called from it, this is done in the three lines after the first one. These class methods can in turn add new methods to both the 'Person' class and objects instantiated from it. This way, the definition of the 'Person' class becomes a program written in the 'ActiveRecord' DSEL for object relational mapping. The second part of the class definition shows an example of using control structures within a class definition, in this case defining a debug method if the code is run in development mode.

```
class Person < ActiveRecord::Base
  # Describing associations and validations using the ActiveRecord DSEL
  has_many :addresses                 # => Assumes addresses.person_id column
  belongs_to :group                   # => Assumes people.group_id column
  validates_presence_of :name, :email # => Assumes people.name and people.email

  # Using a conditional to optionally define a method
  if ENV['RAILS_ENV'] == 'development'
    def debug
      # TODO: Ouput debug information about this Person
    end
  end
end

Person.all :order => 'name'                    # => Return an array of all persons
Person.find_by_email 'example@cs.utwente.nl'  # => Finds a person by e-mail address
Person.first.addresses                         # => Returns first person's addresses
```

Figure 4.5: Examples of imperative class definition

The imperative nature of class definitions means the ordering of definition code matters. Defining a method twice for example would result in an error in any language with declarative class definitions like Java (Polymorphism set aside) while in Ruby the second method definition can just replace the first one without breaking with the paradigm. These semantics allow for partial class definitions; defining some attributes of a class in one place and defining others in another, possibly at a later time, with the class being fully functional (although partially defined) in between. This behavior is augmented by Ruby's dynamic method dispatch semantics; a call to a non-existent method is allowed at definition time and only throws an exception at method invocation time.

The imperative class definition semantics also enable Ruby to have open classes as seen in Smalltalk [1]. All classes in Ruby, including the core Ruby classes, can be subject to additional definition, including redefinition and removal of previously defined methods, from within anywhere in the Ruby program. This concept is often alarming to first time Ruby users and Object Oriented Programming (OOP) evangelists. They argue that Ruby programmers can never be entirely sure of the behavior of even the most basic Ruby objects. However, a good craftsman doesn't use blunt tools in order not to cut himself. Any programming language feature can lead to bad software if used improperly, and the proper use of open classes has many benefits, especially in the area of DSEL's. The code sample in figure 4.6 shows an example of reopening the Ruby 'String' class to add a method called 'titlecase'.

```ruby
class String
  def titlecase
    # Split on whitespace, collect capitalized words, join with a space
    self.split(/\s/).map(&:capitalize).join(' ')
  end
end

"hello world!".titlecase # => "Hello World!"
```

Figure 4.6: Opening the string class

Building on the principle shown in figure 4.6, we can easily build a DSEL for dealing with numeric values and different units of measurement. The code sample in figure 4.7 shows a simple DSEL for listing recipe ingredients that was developed by opening the 'Numeric' class. In this example, both the basic unit of measurement 'grams' and the additional unit of measurement 'cups' where defined directly on the 'Numeric' class. Also, we defined the method 'of' to return an array of both an ingredient name that was passed into it and the quantity of that ingredient. The last line shows an ingredient listing written in our DSEL.

```ruby
class Numeric
  def grams
    self
  end

  def cups
    self * 130.grams
  end

  def of ingredient
    [ingredient, self]
  end
end

[3.cups.of(:flour), 100.grams.of(:sugar)] # => [[:flour, 390], [:sugar, 100]]
```

Figure 4.7: A simple recipes DSEL

4.4.2 Method Chaining

Figure 4.7 is also an example of another common method for the creation of DSEL's: method chaining. This method is especially common in languages that lack the more sophisticated methods we describe in this chapter. Method chaining is an easy way of creating DSEL's wherein implicit context is provided and changed by chained method calls (the target (context) of each method is the result of the previous method). The resulting code can look a lot like natural language with dots instead of spaces. A common problem with method chaining DSEL's is the finishing problem. It arrises when there is no specific order to the methods that are chained together and a final operation or check needs to be applied to the result of the method chain. It can be solved by requiring a specific order in which the methods are chained (figure 4.7) or by requiring a terminating method to be added to the end of the chain.

4.4.3 Duck Typing

As can be seen in previous code samples, Ruby is not explicitly typed like Java or C. Ruby is dynamically and strongly typed, respectively meaning that type checking is done at runtime and that any type errors are regarded as defined and expected behavior. More specifically, it employs a type system called duck typing. In Ruby, every object has a specific type (class) and objects of any type can be assigned to all variables and passed to all methods. Type checking comes down to whether the object can or cannot accept certain messages (method calls), hence the quote from which the term duck typing was born: *'If it walks like a duck and quacks like a duck, I would call it a duck.'*. Additionally, a program can check at runtime whether an object is an instance of a certain type (class) using the 'class' method on any object. This functionality is called introspection. However, because of open classes and to not violate the principle of duck typing, it's usually better to use the 'respond_to?' method instead to check if an object can respond to a certain message. The following methods are also available for further introspection: 'Object.methods', 'Object.method', 'Method.arity' and 'Method.parameters'.

Thanks to duck typing, Ruby does not need automatic type casting at language level to be flexible and concise. Code can just accept any object that responds to a certain method. Methods like 'to_i' (integer), 'to_s' (string), 'to_a' (array), 'to_f' (float) and 'inspect' (pretty print) are implemented by most ruby classes and are called automatically by many standard methods. This way Ruby implements the expected (programmer/user oriented) behavior when for example passing any object directly into string concatenation without having any extra provisions for it at the language level. This is important for DSEL design because it eliminates boiler plate code that has no relation to the problem domain.

4.4.4 Operator Overloading

Like C++, Ruby supports operator overloading. This allows the programmer to define the behavior of operators applied to objects of a certain class the same way methods are defined. In fact, the way Ruby supports operator overloading is by allowing valid method names to consist of characters that are usually reserved by the compiler like '=' and '+'. Additionally Ruby allows the method invocation dot to be omitted for these kinds of methods, and allows the first argument to be passed between brackets for methods like '[]' and '[]='. An example of this in the context of a DSEL for set mathematics is given in figure 4.8.

```
def set *elements; Set.new elements; end

class Set
  attr_reader :elements

  def initialize elements; @elements = elements.uniq.sort; end

  def + other; Set.new self.elements + other.elements; end

  def * other; Set.new(self.elements.map do |e1|
    other.elements.map do |e2| Set.new [e1, e2] end
  end.flatten); end

  def [] element; elements.include? element; end

  # Implementing the spaceship operator enables sorting
  def <=> other; self.elements <=> other.elements; end

  # Implementing inspect enables pretty printing
  def inspect; "set(#{elements.map{|e| e.inspect}.join(', ')})"; end
end

# DSEL for set union, cartesian product and element inclusion checking:
set(1, 1, 4) + set(4, 5, 6)  # => set(1, 4, 5, 6)
set(1, 2) * set(3, 4)        # => set(set(1, 3), set(1, 4), set(2, 3), set(2, 4))
set(1, 2)[3]                 # => false
set(1, 2)[2]                 # => true
```

Figure 4.8: A simple DSEL for performing set mathematics

4.5 Prototype Oriented Programming Features

Ruby is not a prototype oriented programming language like Javascript because it knows the concept of classes. In a true prototype oriented programming language the concepts of class instantiation and specialization are unified into the single concept called prototyping, wherein one object functions as the prototype for the instantiation of a new object with the same methods and data. However, like prototype oriented languages, Ruby allows the definition of a method on a single object instead of a class. This functionality is implemented by inserting an invisible singleton class into the inheritance hierarchy between the object and its original class whenever such a method is defined. This way the simple method dispatch mechanism that was explained in Section 4.4 still applies. That's why we call methods that are defined on a single object singleton methods in Ruby. The code sample in figure 4.9 shows an example.

```
class Car; def method_missing name; "Unknown method: #{name}"; end; end

a = Car.new; b = Car.new

class << a; def honk; "Honk! Honk!"; end; end

a.class == b.class   # => true
a.honk               # => "Honk! Honk!"
b.honk               # => "Unknown method: honk"
```

Figure 4.9: Singleton method definition example

4.6 Functional Programming Features

In Ruby, methods and pieced of code can be first class entities. The 'Proc' class is used to hold a piece of code that can be passed around and be dynamically evaluated in the context of any object or any location in the programming code. Additionally, the 'Binding' class can be instantiated anywhere in the code using the 'binding' method to be passed around and used for evaluation of a 'Proc' object or inline code. Also, an object of the 'Method' class can be acquired from any object using the 'method' method and is essentially a 'Proc' that is bound to that specific object. A 'Method' object can dynamically be invoked on its object or rebound to another object. When a 'Proc' object is invoked without an explicit binding it is evaluated in its original context and is often referred to as a closure. Ruby also provides some syntactic sugar for passing a 'Proc' object to a method without explicitly instantiating it, in which case it is often referred to as block. Both the keywords 'do' and 'end' and accolades can be used to pass a block to a method. The block can then be invoked from the method it was passed to by using the 'yield' keyword. It can also be captured as an argument of the 'Proc' class by prefixing that argument with an ampersand, this 'Proc' object can than be introspected, evaluated in different contexts, saved in the object and passed to other methods further down the method invocation stack. Some example code samples of all this are shown in figure 4.10.

```
# Pass a block to the array map method using do .. end syntax
[1, 2, 3].map do |number|; number * 3; end  # => [3, 6, 9]

# Define a method that applies its block n times to subject
def apply subject, n; n.times{subject = yield subject}; subject; end

# Calculate 3 * 2 * 2 * 2, passing the block using accolade syntax
apply(3, 3){|s| s * 2}                     # => 24

# Assign a new Proc that multiplies by 2 to variable p
p = Proc.new {|s| s * 2}

# Calculate p(p(p(3))), passing in Proc p as the block
apply 3, 3, &p                             # => 24

# Calculate the same, calling p directly
p.call(p.call(p.call(3)))                  # => 24
```

Figure 4.10: Example of using procs

4.6.1 Ruby Blocks and DSEL's

The concept of blocks in Ruby opens up interesting possibilities in the area of DSEL's. In Section 4.2 we established that DSL's are more effective for writing programs in the application domain of the DSL because they provide implicit context to that program. Ruby's ability to pass around blocks of code as first class entities and evaluate them in the context of some object allows us to do exactly this. The object that is used for evaluation of the block can provide context to that block in the form of methods and instance data. Figure 4.11 shows a code sample that uses Ruby to define a DSEL with implicit context using blocks.

```ruby
# Create base class that allows initialization with block
class DSELElement; def initialize block; instance_eval(&block); end; end

# Create transformation that defines rule
class Transformation < DSELElement
  def rule &block; @rules ||= []; @rules << Rule.new(block); end
  def inspect; "transformation(#{@rules.map{|r| r.inspect}.join(', ')})"; end
end

# Create rule that defines source and target
class Rule < DSELElement
  def source; @source = yield; end
  def target; @target = yield; end
  def inspect; "rule(source=#{@source}, target=#{@target})"; end
end

# Define method on main object to instantiate transformation
def transformation &block; Transformation.new block; end

transformation{
  rule{ source{ 'a' }; target{ 'b' } }
  rule{ source{ 'c' }; target{ 'd' } }
} # => transformation(rule(source=a, target=b), rule(source=c, target=d))
```

Figure 4.11: Using blocks to create a DSEL with implicit context

4.7 Aspect Oriented Programming Features

Ruby is not a true aspect oriented programming language but it defines hooks for many system events that can be used to simulate aspect oriented behavior. The 'method_missing' hook was already mentioned several times in this chapter. It can be used to intercept calls to non-existent methods. The 'const_missing' method is called whenever a constant like a class name wasn't found, this hook can for example be used for autoloading of classes. When a class is defined that inherits properties from another class, the 'inherited' hook is called on the superclass. When modules are added to a class the 'included' or 'extended' hook is called on the module, depending on whether to module was added at class or at instance scope. Finally, 'method_removed' and 'method_added' are called on a class when methods are removed from it or added to it. The 'method_added' hook can for example be used to wrap the added method with additional behavior, effectively making Ruby an aspect oriented programming language. In addition to 'method_added' and 'method_removed', Ruby also provides 'singleton_method_added' and 'singleton_method_removed' that are called whenever singleton methods are added or removed.

Many of these hooks have application in MDEL design. The 'const_missing' hook can for example be used to catch keywords in the DSEL, removing the need for boiler plate characters like the colon for symbols and quotes for strings. The 'method_missing' hook can be used to create a DSEL like the one shown in figure 4.11 wherein the block names are not predefined. A good example of this is the XMLBuilder library for generating XML shown in figure 4.12.

```ruby
require 'rubygems'
require 'builder'

Builder::XmlMarkup.new(:indent => 0).date { x.year 1984; x.month 2; x.day 20 }
# => "<date><year>1984</year><month>2</month><day>20</day></date>"
```

Figure 4.12: Demonstration of the XMLBuilder gem

4.7.1 Catching missing constants to clean up a DSEL

Identifiers that start with a capital letter are considered constants by the Ruby language. Where regular missing identifiers can be caught by implementing the 'method_missing' hook, missing constants can be caught by implementing the 'const_missing' hook. This mechanism can be used to further clean up a DSEL that uses strings or symbols. Figure 4.13 shows how this principle can be applied to our recipe DSEL that was presented in figure 4.7. For simplicity's sake, the 'const_missing' hook is defined on 'Object' here, which means it will turn any missing constant anywhere in the program into an ingredient. In a real application the hook should be defined in a more specific context to avoid confusing errors.

```
class Ingredient
  def initialize name; @name = name; end
  def to_s; name end
end

def Object.const_missing name; Ingredient.new(name); end

2.cups.of Flour # => [#<Ingredient:0x21d89c @name=:Flour>, 260]
```

Figure 4.13: Using the const_missing method to clean up the recipe DSEL

4.8 Conclusions

DSEL's are API's in an existing programming language, the use of which resembles the use of a DSL. A DSL is a language that was specifically designed for writing programs in a particular application domain. A DSL is more effective for expressing problems in its domain because it provides implicit context to those programs. Ruby is a flexible general purpose programming language that integrates concepts from many different programming paradigms. These paradigms are object oriented, functional, prototype oriented and aspect oriented. This means the language supports many different ways of creating DSEL's which makes it, to the best of our knowledge, the most suitable language for this purpose. We identified and described the following facilities in the Ruby language that are useful for the creation of DSEL's.

- Flexible syntax that allows different notations for the same construct
- Meta-programming API for introspection and reflection
- Open imperative class definitions
- Method chaining
- Operator overloading
- Code blocks and the ability to evaluate them in a particular context

Chapter 5

TRans: A MTEL prototype

As we mentioned, the host language was already decided upon in Chapter 2, answering research question 1. We chose Ruby as our host language because of its concise and flexible syntax, extensive features for the creation of DSEL's, support for declarative and functional programming, reflective features and mature standard library. Its facilities for the creation of DSEL's are described in detail in Chapter 4, effectively answering research question 2. For our goal, the most important of these are blocks and operator overloading.

In this chapter we introduce and explain our MTEL (Model Transformation Embedded Language) called TRans. The name TRans was chosen first and foremost because it starts with the letters 'T' and 'R', respectively signifying transformations and Ruby. Additionally, the Latin prefix 'trans', meaning across or beyond, conveys the sense of opposite sides of the transformation process (source and target) and the abstracting or transcending nature of meta-modelling and code generation.

Because there can be no model transformation without models, we start of with introducing our basic modelling language in Section 5.1. This modelling language is not part of the TRans transformation language and can easily be replaced with another implementation. We will then introduce the basic features of the TRans language in Section 5.2. Section 5.3 shows how concerns can be added to transformations to augment or alter transformation behaviour. Section 5.4 goes deeper into the tracing features TRans provides. Section 5.2.1 shows a complete more complex transformation to provide context to the concepts that are introduced in this chapter. Lastly, Section 5.5 describes features for code generation.

The abbreviations MDE, MTL, DSL, DSEL, MTEL that are routinely used in chapters 5, 6 and 7 respectively stand for 'Model Driven Engineering', 'Model Transformation Language', 'Domain Specific Language', 'Domain Specific Embedded Language' and 'Model Transformation Embedded Language'. Figure 5.1 shows what the relations are between these concepts (in bold case) and how different model transformation languages (in regular case) fit into this picture.

5.1 A Basic Modelling Language

We decided to implement a basic modelling language to use with the TRans MTEL. We chose this approach because implementing a basic modelling solution actually doesn't require that much effort and the focus of this research is on model transformation and not on modelling itself. The use of an existing modelling infrastructure would have required a lot more effort because implementations of these existing infrastructures in Ruby are very immature.

The modelling language and TRans are very loosely coupled, the only coupling is a very minimal and very generic interface the modelling language needs to implement. The model transformations themselves can however be strongly coupled to both a specific modelling language implementation and the TRans MTEL. This makes it possible to wrap models from any existing modelling infrastructure in a Ruby object en perform transformations on them using TRans.

In our basic modelling language every model consists of a collection of elements, each consisting of a dictionary (a set of key/value pairs, also called a hash). Relations between model elements are established by allowing the values in the dictionary to reference other model elements. This way the modelling language only supports one-to-many relationships, many-to-many relationships can be modelled by having

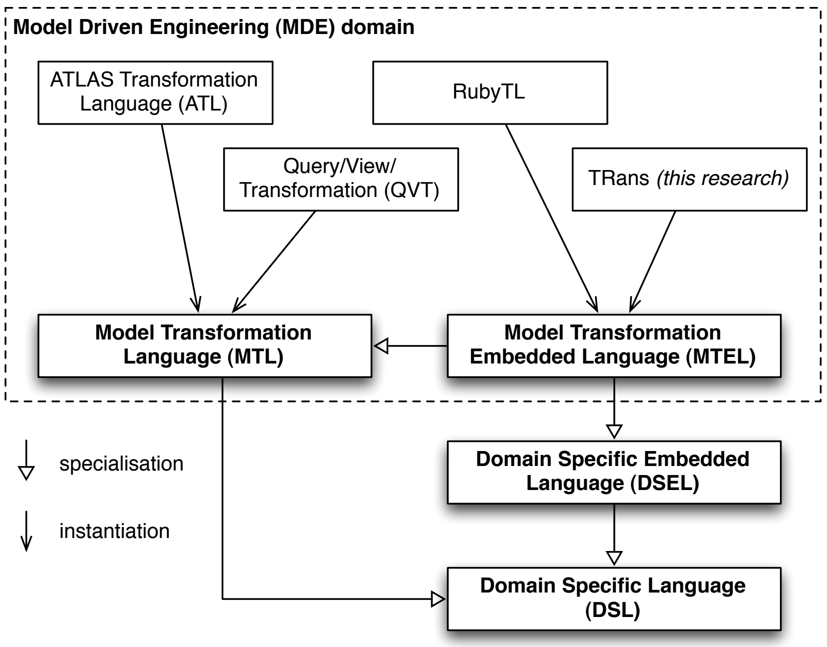

Model Driven Engineering (MDE) domain
ATLAS Transformation Language (ATL)
RubyTL
Query/View/Transformation (QVT)
TRans (this research)
Model Transformation Language (MTL)
Model Transformation Embedded Language (MTEL)
specialisation
instantiation
Domain Specific Embedded Language (DSEL)
Domain Specific Language (DSL)

Figure 5.1: Meaning of and relations between different concepts used in this report

explicit joining elements that refer to both ends of the relationship.

Our basic modelling language implementation supports a DSEL for model definition described in Section 5.1.1 and functionality for loading models from and saving models to the filesystem described in Section 5.1.2. It also implements a DSEL variant of OCL (OCEL if you will) in order to query models from within a transformation which is described in Section 5.1.3. Please note that this basic modelling language implementation does not contain any meta-modelling facilities.

The code sample in figure 5.2 shows some sample code for working with models. The first line shows that models can be instantiated just like any other Ruby class using the 'new' method. This produces a model without any elements. In the second line an element is added to this empty model using the 'element' method. While the availability of the methods 'new' and 'element' is required by the TRans MTEL, the arguments that are passed into the 'element' method can vary per modelling language implementation. In our basic implementation all arguments are stored in the element dictionary where the first (unnamed) argument is stored under the name 'meta' for meta-model type or meta-model class. The third line shows our implementation of the 'inspect' method for models that allows Ruby to pretty print models for inspection. The last line shows a list of all instance methods that are available on model objects.

```
require '../../../../final/trans/models/element.rb'
require '../../../../final/trans/models/model.rb'

model = Model.new
model.element 'Class', :name => 'Person'
model # => A Name: "Person", Meta: "Class"

Model.instance_methods false
# => ["element_class", "identifier", "any?", "supports_interface?",
#     "first", "all?", "select", "element", "elements", "save", "inspect"]
```

Figure 5.2: Sample code for dealing with models

5.1.1 Simple Model Definition DSEL

While models can be defined from code using the syntax that was presented in figure 5.2, we also provide a DSEL for this task that is a bit more concise. The code sample in figure 5.3 uses this DSEL to define a simple class model. It allows the receiver for the element method to be omitted and translates the use of constants (identifiers that start with a capital letters) to strings so all the quotes can be omitted as well. This techniques for creating DSEL's in Ruby are described thoroughly in Chapter 4.

```
Model.new {
  person = element Class, name: Person
  element Attribute, name: Age, type: Number, owner: person
  element Attribute, name: Name, type: String, owner: person
  teacher = element Class, name: Teacher
  element Attribute, name: Faculty, type: String, owner: teacher
  element Specialization, source: teacher, target: person
  professor = element Class, name: Professor
  element Attribute, name: Hobbies, type: String, owner: professor
  element Specialization, source: professor, target: teacher
  course = element Class, name: Course
  element Attribute, name: Name, type: String, owner: course
  element Association, source: course, target: teacher
}
```

Figure 5.3: DSEL for defining models in code

5.1.2 Loading and Saving Models

The code sample in figure 5.4 shows how models can be saved and loaded. The first line shows how a model object can be saved to a file by calling the 'save' method on it, passing in the file name. The second line shows how the model class can be used to instantiate a model object and load it from a file using the 'load' method and passing in the file name.

```
1   model.save "model.yml"
2   loaded_model = Model.load "model.yml"
```

Figure 5.4: Sample code for loading and saving models

Notice that the file names end in '.yml' which stands for YAML. YAML (YAML Ain't Markup Language) is a human friendly data serialization standard for all programming languages. We use Ruby's standard ability to (de)serialize any object to/from YAML without writing any serialization code. A sample model file with two elements can be seen in code sample 5.5. Like Ruby, YAML supports dictionaries and collections and references. This means the translation from a Ruby memory structure to a YAML serialization is very straight forward. The model has an attribute called elements which is a collection, as signified by the '-' (dash) symbols. Furthermore, the first model element is assigned an identifier using the '&' (ampersand) and the second model element references this identifier, using the '*' (asterisk) symbol.

5.1.3 Querying Models using OCL

While we have no desire to implement the actual OCL language, we implemented most of its querying functionality as a Ruby API. This includes collections, arithmetic, quantifiers and iterators. The main difference is that our implementation does not support the four different collection types that OCL does. Instead only the sequence is supported by use of the standard Ruby array. Duplicates and ordering can respectively be ignored by use of the 'unique' and 'order' methods, mimicking similar behaviour. All of these features if already in the Ruby language and just needs a thin wrapper to make it look more compatible with OCL. Not having to implement OCL while getting the functional equivalent practically for free is one of the big advantages of our approach.

The code sample in figure 5.6 shows an example OCL query using the Ruby syntax. It shows use of the 'select' iterator and the existential quantifier, in our implementation called 'any?'. Ruby closures make it easy to to implement the OCL iterators and quantifiers, most of them can just be forwarded to

```
--- !ruby/object:Model
elements:
- &id001 !ruby/object:Element
  attributes:
    :meta: Class
    :name: Person
- !ruby/object:Element
  attributes:
    :meta: Attribute
    :name: Age
    :type: Number
    :owner: *id001
```

Figure 5.5: Sample YAML model file with two elements

the model's internal array of elements that also supports these iterators and quantifiers. The only thing the model class adds to this behaviour is the ability to filter equivalence of the 'meta' attribute and also other attributes by passing in some regular arguments. In case of the universal quantifier, the semantics are that it returns true if all elements that correspond to the attribute equivalence arguments respond true to the block.

```
class_model.select(Class) { |c|
  !class_model.any?(Specialization) { |s| s.source == c }
}
```

Figure 5.6: Sample of our OCL variant, implemented as a DSEL

Table 5.1 shows OCL samples and their functional equivalent in our Ruby implementation. Please note that apart from several subtle differences, Ruby and OCL have mostly similar syntax. We could have opened up standard classes and created proxies to make our implementation a whole lot more similar to OCL still. We decided not to do this however because the purpose of a DSEL isn't to mimic the syntax of another language. Functionally the main difference between OCL and Ruby is that Ruby is a lot richer and supports a lot of functionality that isn't in the OCL specification.

OCL Sample	**Functional Equivalent**
true, false	true, false
1, 2, -1, 1.2	1, 2, -1, 1.2
'string'	'string'
$=, <>, not, and, or$	$==, !=, !, and, or$
$>, >=, <, <=$	$>, >=, <, <=$
+, -, *, /, mod()	+, -, *, /, %
concat(), size(), substring(), toInteger()	+, size, slice, to_i
Sequence{1,2}	[1,2]
Bag{1,2}	[1,2].sort
OrderedSet{1,2}	[1,2].unique
Set{1,2}	[1,2].unique.sort
if 3 > 2 then 'yes!' else 'what?' endif	if 3 > 2 then 'yes!' else 'what?' end
select(), collect(), forAll(), exists()	select, map, all?, any?

Table 5.1: Functional equivalents to OCL in our ruby implementation

5.2 TRans Introduction

We tried to make TRans as complete a language as possible, within limited time constraints, tackling the most essential problems first. We decided in advance that any viable MTL at least needs a querying language which functionally equivalent to OCL, tracing facilities and recursive helpers. We decided the querying language is the responsibility of the modelling language and implemented it there, as described in Section 5.1.3. Tracing facilities, (recursive) helpers, model declarations and rules, as well as many other features, are implemented by the TRans language itself and are described throughout the rest of this chapter.

5.2.1 A Complete Transformation

To provide some context to all the topics we are about to discuss, figure 5.7 shows a complete and realistic transformation. All aspects of this transformation are discussed throughout the rest of this chapter.

This particular transformation in our implementation of a standard case study in model transformation. The purpose of the transformation is to create a database model from a class model. Since most relational database don't support inheritance, the inheritance hierarchy needs to be flattened using a pattern using single table inheritance [15]. Basically, a table is created for each inheritance root, with all the attributes of all the classes in that inheritance hierarchy. The table also gets additional attributes 'Id' and 'Type' in respectively in order to be referred to by other tables and to store the type of the subclass that is actually being stored in each table row.

```
source :class_model, Base
target :database_model, Base

rule {
  from { class_model.select(Class) { |c|
    !class_model.any?(Specialization) { |s| s.source == c } } }
  build(:database_model, Table) { |c| name c.name }
  build(:database_model, Column) { |c|
    name Id
    owner trace(target: Table, source: c)
    type Integer }
  build(:database_model, Column) { |c|
    name Type
    owner trace(target: Table, source: c)
    type String } }

rule {
  from { class_model.select Attribute }
  build(:database_model, Column) { |a|
    name a.name
    owner trace(target: Table, source: root(a.owner))
    type a.type } }

rule {
  from { class_model.select Association }
  build(:database_model, Column) { |a|
    name a.target.name + Id
    owner trace(source: root(a.source), target: Table)
    type Integer }
  build(:database_model, ForeignKey) { |a|
    source trace(source: a, target: Column)
    target trace(source: root(a.target), target: [Column, name: Id]) } }

helper(:root) { |c|
  if c.nil? then nil else
    root class_model.first(Class) { |p|
      class_model.any?(Specialization) { |s| s.has? source: c, target: p }
    } or c
  end }

```

Figure 5.7: A complex transformation

5.2.2 Transformation Loading and Execution

Transformations written in TRans are usually saved in a text file. A single transformation is stored in a single text file. The code sample in figure 5.8 shows how to load a transformation from a Ruby file and execute it. The transformation is loaded by calling the 'load' method on the 'Transformation' class.

This is similar to the way models are loaded, which was shown in figure 5.4. The second line shows how the loaded transformation can be used to transform a number of source models (first argument) to a number of target models (second argument) using the 'transform' method. The keys in the hash are the names the models have in the transformation and the values are the actual models that are passed in. Empty models should be passed into the target models argument, otherwise they are just augmented. The transformation language cannot instantiate the target models itself because it is unaware of the different implementations of the model concept.

```
transformation = Transformation.load('transformations/class2database.rb')
transformation.transform({class_model: source}, {database_model: target})
```

Figure 5.8: Code sample of loading and executing a TRans transformation

5.2.3 Model Declarations

The TRans MTEL allows multiple source and target models. This enables transformations to combine multiple source models into a single target models and/or decompose a single source model into multiple target models. Especially the combination of models into a single target model is essential for achieving separation of concerns using model transformation because the concerns can be separated into those different source models. Branching a single source model into multiple target models can also be achieved by simply applying a different transformation for each desired target model, it is implemented mainly for convenience. Figure 5.9 shows the TRans transformation pattern with the transformation definition defining source and target modelling languages and the transformation execution taking instances of these types as input and output.

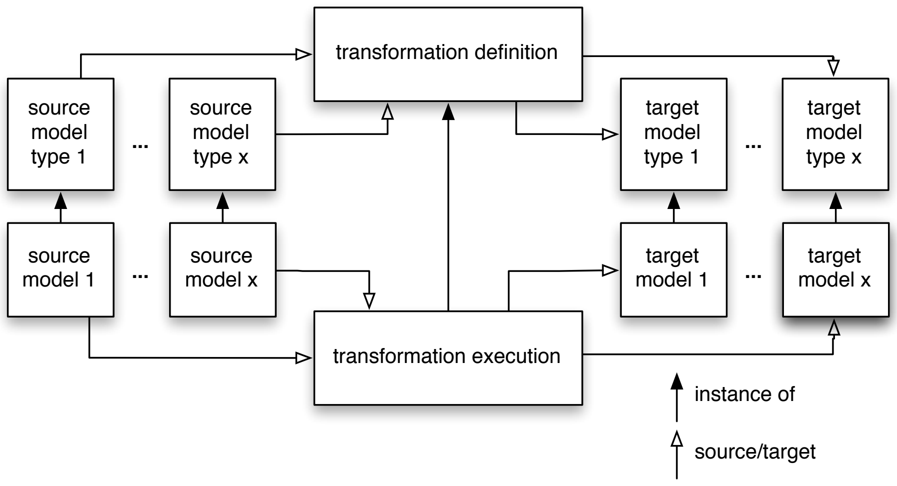


Figure 5.9: Diagram showing the model transformation pattern of TRans

All source and target models the transformation uses need to be declared within the transformation, using the keywords 'source' to declare a source model and using the keyword 'target' to declare a target model. The code sample in figure 5.10 shows the declaration of a single source model and a single target model in TRans. The symbols 'class_model' and 'database_model' are the names that are used to refer to the models from within the transformation rules. The 'Base' constant after each declaration is the modelling language interface the model should support. The string 'Base' is passed to the models' 'supports_interface?' method in order to make sure they they actually support the interface. When transforming models in our basic modelling language this constant can always just be 'Base'. When models that support a different interface such as querying using XPath or code generation are expected or required by the transformation, it can be declared here.

```
source :class_model, Base
target :database_model, Base
```

Figure 5.10: Code sample declaring a source and a target model

5.2.4 Requiring Libraries

One of the great things about TRans being a Ruby DSEL is that any Ruby library can be used from within transformations. The syntax for doing this is exactly the same as the Ruby syntax; by using the keyword ’require’. This keyword is caught by the DSEL context and forwarded directly to the standard implementation. A Ruby library that is especially useful in model transformations is ActiveSupport. ActiveSupport is the support library for the Rails web application framework and includes a lot of useful string inflections such as pluralization, singularization, camelcasing, etc. The code sample in figure 5.11 shows how to include ActiveSupport from a TRans transformation. Ruby Gems is a package distribution and dependency management tool and needs to be required first because ActiveSupport is distributed through this mechanism.

```
require 'rubygems'
require 'activesupport'
```

Figure 5.11: Code sample of requiring a Ruby Gem from a TRans transformation

5.2.5 Transformation Rules

The most important language construct in TRans is the (transformation) rule. A rule can have a source clause, which is evaluated when the model transformation is executed. For more information on the transformation execution algorithm, please consult Section 6.2.4. It can also have a name and has one or more target clauses which are evaluated once for every element that results from evaluation of the source clause. If no source clause is specified or if the result of its evaluated doesn’t respond to the map iterator, each target clause is evaluated just once. When target clauses are evaluated, the corresponding source element or value is passed into the block of each target clause as the argument, this way data from source models can be processed and/or copied into the target models.

The code sample in figure 5.12 shows a very basic TRans transformation rule. The argument passed to the ’rule’ keyword is its name. This name can be used for tracing purposes and may be omitted. Within the rule code block the ’from’ keyword may be used once and the ’build’ keyword should be used once or more. The ’from’ keyword specifies the source of the transformation rule and only takes a block which is evaluated in a context in which all source models are available. Any querying language the source models support can be used here but in this example the OCL-like querying DSEL described in Section 5.1.3 is used.

The target clause is specified by the ’build’ keyword and takes one argument into which the source element or value is passed. In this case the argument is called ’c’ for class because the source clause results in a collection of classes. The two arguments to the build are the target model to which the resulting element should be added and optionally a bunch of arguments to ’element’ method of the model, in this case only the value for the ’meta’ value of the element. Which arguments are allowed here and their semantics is, like the code in the source clauses, entirely up to the modelling language. The same goes for the code in the target clause code block; ’name’ is a method that is called on the element. In this case it takes a single parameter and specifies the value for the ’name’ key in the element’s dictionary.

The keywords ’from’ and ’build’ where chosen over ’source’ and ’target’ because the sentence ’from ..., build ...’ reads like natural language and the terms source and target a somewhat overloaded within the model transformation domain. Like with the natural language sentence, the source and target clauses can be switched around within the rule code block. Please also note that the code between the curly braces after the ’from’ or ’build’ keywords is not evaluated until transformation execution time. The sample code in figure 5.12 and accompanying description in this section show why there is hardly any coupling between the modelling language and the transformation language but an actual transformation is strongly coupled with both. Any coupling that does exist between the transformation language and the modelling language is there to make transformation writing more convenient.

```
rule(Tables) {
  from { class_model.select Class }
  build(:database_model, Table) { |c|
    name c.name
  }
}
```

Figure 5.12: Code sample of a simple rule in TRans

5.2.6 Helpers

Helper rules are functions that are defined in the context of the transformation. Like regular functions, they can take arguments and have the ability to recursively call themselves. They are used to prevent duplication and encapsulate functionality. In many other transformation languages the recursiveness of helpers are required to make the language Turing complete. Since TRans is not a purely declarative language (all imperative control structures in Ruby are available) it doesn't need recursion to be Turing complete. Using recursion and declarative programming is however considered good practice.

The code sample in figure 5.13 shows a helper definition. The keyword 'helper' is used, passing the name of the helper as the first parameter. The implementation of this particular helper is somewhat complex but it is implemented recursively and declaratively (as a single expression). The goal of this helper is to determine the root of the inheritance hierarchy of a class. The first line of the helper implementation defines the base case; a call to the helper where the class is nil also returns nil. The recursive case calls itself on the parent of the class, or on nil of none exists, resulting in nil like we discussed. If this call to itself results in nil the class is itself the root of the hierarchy and is chosen over nil by the or operator. Some more characteristics of our querying DSEL where inadvertently demonstrated in this helper as well; the 'first' iterator is like the 'select' iterator but only returns the first element and the 'has?' method on model elements can be used to concisely check equivalence of multiple element attributes.

```
helper(:inheritance_root) { |c|
  if c.nil? then nil else
    inheritance_root class_model.first(Class) { |p|
      class_model.any?(Specialization) { |s| s.has? source: c, target: p }
    } or c
  end
}
```

Figure 5.13: Code sample of a TRans helper

5.3 TRans AOP Features

AOP features are implemented using metaphors from the Aspect Oriented Programming (AOP) field. A join point model is implemented for the transformation execution algorithm and concerns that can include before, after and around advice for join points can be registered at transformation definition and execution time. The join point model is implemented by wrapping certain parts of the execution algorithm in the block of a Ruby method that evaluates it's block along with any advice that needs to be evaluated as well. Around filters have full control over the return value and any exception of the wrapped part of the execution algorithm, this way its behaviour cannot only be augmented but also be changed. Possible applications of this kind of AOP in TRans are saving trace data, logging, realtime debugging, omnipresent debugging, modifying the executions algorithm and model element filtering.

To define a concern, the 'Concern' class must be specialized to implemented methods with names that start in either 'before_', 'after_' or '_around' and end with the name of a join point. Alternatively, instead of the name of a join point, 'all' can be specified as a wild card. One or more concerns can be passed to the call to the 'transform' method of a loaded transformation as extra arguments. The code sample in figure 5.14 shows how a simple join point logger can be defined by specializing 'Concern' and adding a 'around_all' method. It also shows how this concern can then be passed along when performing the transformation. The first line of the 'around_all' method body is only a bit of Ruby magic to do pretty printing while the second line shows how the 'yield' keyword is used to invoke the original join point code. The local variable result is used to save the result of the join point code and also return as the result of the '_around' method in other not to influence any behaviour. Figure 5.15 shows sample output generated by the concern for a small transformation.

```
class JoinPointLogger < Concern
  def initialize; @indent = 0; end

  def around_all name, *arguments
    puts "#{'|  ' * @indent}\\_ #{name.to_s.gsub(/_/, ' ').capitalize}"
    @indent += 1; result = yield; @indent -= 1; return result
  end
end

transformation.transform(source_models, target_models, JoinPointLogger.new)
```

Figure 5.14: Code sample of a TRans join point logger

```
\_ Transformation
|  \_ Source model check
|  \_ Target model check
|  \_ Transformation element generation
|  \_ Attribute generation
|  |  \_ Element attributes generation
|  |  |  \_ Trace
|  |  |  |  \_ Element attributes generation
|  |  |  |  \_ Element attributes generation
|  |  \_ Element attributes generation
|  |  |  \_ Trace
|  |  \_ Element attributes generation
|  |  |  \_ Trace
|  |  \_ Element attributes generation
|  |  |  \_ Trace
```

Figure 5.15: Sample output for the code sample in figure 5.14

When implementing concerns that do not need to influence the behaviour of the component that implements the join point model it is often a bad idea to use '_around' methods because it is to easy to accidentally influence behaviour. Most of the applications of AOP in TRans in model transformations we discussed earlier just extract information and don't need to influence behaviour. Examples of this are logging, simple realtime debugging and omnipresent debugging. In these cases it is better to use '_before' and '_after' methods instead and eliminate the need to call yield and return a value. The code sample in figure 5.16 shows how we can rewrite the single '_around' method in figure 5.14 this way.

```
def before_all name, *arguments
  puts "#{'|  ' * @indent}\\_ #{name.to_s.gsub(/_/, ' ').capitalize}"
  @indent += 1
end

def after_all name, *arguments; @indent -= 1; end
```

Figure 5.16: Changing the join point logger to use '_before' and '_after' methods

Concerns can also be defined directly from within a TRans transformation specification using the keyword 'concern'. The concern is then woven into all executions of the transformation, as well as any concerns specified for the execution itself. The code sample in figure 5.17 shows how a concern can be specified in a transformation definition that checks that all target models adhere to their meta-model. Please note that our modelling language doesn't actually support meta-modelling features and consequently this is hypothetical example.

```
concern {
  def after_transformation source_models, target_models
    target_models.values.each do |target_model|
      target_model.check_meta_model
    end
  end
}
```

Figure 5.17: Sample code for defining a concern from the transformation

5.4 Tracing in TRans

In Section 5.2 we saw how the language can be used to generate elements in a target model from elements in a source model. This, however, is only the first step. A model transformation language needs to establish links between generated elements in order to perform realistic transformations. This ability is provided by tracing. When each element is being generated it may perform traces to establish links to other elements that where already generated or possible need to be generated first. This way, traces dictate the order in which target elements are generated while not shifting from the declarative paradigm.

The most obvious way to implement automatic transformation ordering based on traces is by ordering the rules of a transformation. The execution algorithm can start out by evaluating an arbitrary rule and evaluate rules this initial rule is dependent on as traces are encountered. When a rule is directly or indirectly dependent on itself being evaluated, a cycle is detected and the execution algorithm should report an error. If the granularity of these dependencies is increased from rules to rule targets or even individual target elements, more transformations execute without these cyclic dependency errors, which is desirable behaviour. TRans manages these dependencies on a per-element basis which guarantees that no unnecessary cyclic dependency errors occur.

The code sample in figure 5.18 shows a simple rule for generating target database columns from source class attributes. A trace is executed to determine the owner of the newly created column element. The trace method takes a hash of options specifying what target elements are required. The rest of this section goes into what options are supported by TRans.

```
rule {
  from { class_model.select Attribute }
  build(:database_model, Column) { |a|
    name a.name
    owner trace(target: Table, source: a.owner)
    type a.type
  }
}
```

Figure 5.18: An example of a simple trace in TRans

5.4.1 Tracing Constraints

There are three ways of specifying which elements should be matched by the trace. Constraints can be placed on the element itself, on the source element it is created from and what rule it is created by, respectively specified with the options 'target', 'source' and 'rule'. Values for the options 'target' or 'source' are passed directly to respectively the source element or target element. This means that their interpretation is entirely up to the modelling language implementation.

Target and Source Element Constraints

For our basic modelling language that was discussed in Section 5.1, model elements support the passing of arrays, strings, hashes and elements as constraints. When an array is passed all elements of the array are interpreted as constraints recursively. When a string is passed, it must equal the 'meta' value for the element. When a hash is passed as a constrain all keys must be present as attributes of the element and their values must equal the values in the hash. When another element is passed it must be the same element in order for the constrain to accept the element, this is obviously only applicable to source elements.

The code sample in figure 5.19 shows these element constraints in practice. It is the code for a rule target that takes a foreign key constraint in a database model from a one to many association in a class model. The elements 'source' and 'target' for the foreign key element are both determined by a trace. The first trace matches an element that was also created from the association element but is of the meta type 'Column', it matches the referring column that was also created. The second trace matches an element that was created from the target of the association, is of meta type 'Column' and has the name 'Id'. This is the primary key of the column the referring column is referring to.

```
build(:database_model, ForeignKey) { |association|
  source trace(source: association, target: Column)
  target trace(source: association.target, target: [Column, name: Id])
}
```

Figure 5.19: An example of element constraints for our basic modelling language

Rule Constraint

The 'rule' constraint option can only accept a string that much match the name of the rule that created the matching element. This option is especially useful for code generation which we will discuss in Section 5.5.

5.4.2 Return Types

The option 'returns' can be used to specify how the trace method should deal with different numbers of matching elements. The possible values for this option are shown in table 5.2, along with the result of the trace function when respectively zero, one or multiple matches are found. No actual value is returned from the trace wherever it says exception, instead an exception is thrown. When the 'returns' option isn't specified, the trace method defaults to the behaviour specified for the 'Only' value.

Value	**No match**	**Single match**	**Multiple matches**
Only (default)	exception	only match	exception
One	exception	only match	first match
Any	nil	only match	first match
All	empty array	matches array	matches array
OneOrMore	exception	matches array	matches array
Count	0	1	number of matches

Table 5.2: Possible values for the 'returns' option for the trace method

5.4.3 Result Count Conditions

When 'All' or 'OneOrMore' is specified for the 'returns' option, the options 'minimum' and 'maximum' can specified to impose additional constraints on the number of matches that must be found. When specifying the 'minimum' option with a number the trace function throws an exception when less matches than that number are found. Analogously, the 'maximum' option causes an exception to be raised when more matches are found than the number specified for the option.

5.5 Code Generation

The code sample in figure 5.20 shows how TRans can be used with a different modelling language to perform code generation. It is in many ways the same as our basic modelling language but its elements support three special attributes that can be used to generate code. The 'text' and 'nest' attributes can both be used multiple times to add to the code content for an element. Both strings and traces to other elements can be used to add to the code content of the element. The difference is that the 'text' attribute keeps the same indentation and the 'nest' attribute increases the indentation. After the transformation is performed, all elements that include a attribute 'file' are written to those files.

```
require 'rubygems'
require 'activesupport'

source :class_model, Base
target :rails_app, CodeGeneration

rule {
  from { class_model.select Class }
  build(:rails_app) { |c|
    file "app/models/#{c.name.underscore}.rb"
    text "class #{c.name} < #{model_parent(c)}"
    nest  trace(return: All, rule: Associations, target: {source_class: c})
    text "end"
  }
}

helper :model_parent do |c|
  specialization = class_model.select(Specialization, source: c).first
  if specialization.nil? then 'ActiveRecord::Base' else specialization.target.name end
end

rule(Associations) {
  from { class_model.select Association }
  build(:rails_app) { |a|
    text "belongs_to :#{a.target.name.underscore}"
    text "validates_presence_of :#{a.target.name.underscore}_id"
    source_class a.source
  }
  build(:rails_app) { |a|
    text "has_many :#{a.source.name.pluralize.underscore}"
    source_class a.target
  }
}
```

Figure 5.20: A code generation sample

Figure 5.21 shows a possible output model for the transformation in figure 5.20. Only the model elements that have a 'file' attribute are shown because only those elements will be written to output files.

```
-- Models code model:
  A File: "app/models/person.rb", Content:
    class Person < ActiveRecord::Base
    end
  B File: "app/models/teacher.rb", Content:
    class Teacher < Person
      has_many :courses
    end
  C File: "app/models/professor.rb", Content:
    class Professor < Teacher
    end
  D File: "app/models/course.rb", Content:
    class Course < ActiveRecord::Base
      belongs_to :teacher
      validates_presence_of :teacher_id
    end
```

Figure 5.21: A code generation sample

5.6 Conclusions

In this Chapter we discussed our MTEL called TRans and our implementation of a simple modelling language to be used with it. Even though TRans is an academic prototype, it is a pretty usable MTL. It is a very rich language because it is actually Ruby, made to look like a MTL, and all features and libraries that are available in Ruby are available in TRans. It adds to Ruby the ability to write declarative model transformation programs that consist of model declarations, rules and helpers. TRans provides features for AOP, tracing and code generation to these programs, as well as many extension points where TRans can be extended. We will use the TRans case study to answer the research questions that where raised in Chapter 2.

Chapter 6

TRans: Architecture, Design and Implementation

This chapter describes the architecture, design and implementation of our MDE environment. We start by providing a logical overview of the components that make up our environment in Section 6.1. This section also provides a short description of each component. Chapter 6.2 describes the design of the important components more thoroughly using class diagrams. This section also explains the algorithms we use for model transformation, DSEL evaluation and concern weaving using sequence diagrams. Finally, Section 6.3 contains some implementation guidelines.

Since we implemented our MDE environment in Ruby and Ruby is such a dynamic language it is hard to describe every aspect of it using traditional UML diagrams. While this chapter uses UML diagrams where possible, the dynamic aspects of our implementation are described using natural language.

6.1 Architecture

The overview in figure 6.1 shows the main components of the TRans language and how it is positioned among other components, such as concerns and models. The boxes in the diagram represent implementations and components. Boxes with a dashed line are alternative implementations of the enclosing component, which is an interface or abstract class. Components (with a solid line) are integral to the enclosing component.

The relations between figures 3.3 (Model Transformation Architecture) and 6.1 (TRans Architecture) can be a little bit hard to see since the first is a process oriented view and the latter is a purely logical view. Additionally, TRans does not support any meta-modelling, whereas figure 3.3 describes mostly meta-modelling architecture. However, the concepts of transformation definition, transformation execution and model are shown in both figures.

The first important thing to note is that TRans is not directly coupled to a specific implementation of the model concept, the coupling is achieved by the transformations themselves. Model transformations can use specific API's a model implementation provides like OCL, UML, XQuery or any other language. Additionally whether or not to implement a meta-modeling architecture and/or whether or not to enforce it during transformation is entirely up to the model implementation as well. All TRans does is provide an inversion of control mechanism for working with these models that is driven by the topological sort algorithm.

Two libraries (AOP Tools and DSEL Tools) where extracted from the TRans language because they are excellent candidates for reuse. The DSEL Tools component provides the Ruby context for the evaluation of an arbitrary DSEL by applying a chain of handlers to every encountered keyword, it is described in detail in Section 6.2.1. The AOP Tools component provides join point model specification for arbitrary pieces of Ruby code along with concern weaving and very crude point cut specification. It can be used as a platform to implement reflective features in a Ruby system like TRans. It is described in detail in Section 6.2.2.

The TRans language itself consists of two parts, the transformation representation and its execution. The representation is responsible for loading the file en evaluating most of the DSEL. It stores the

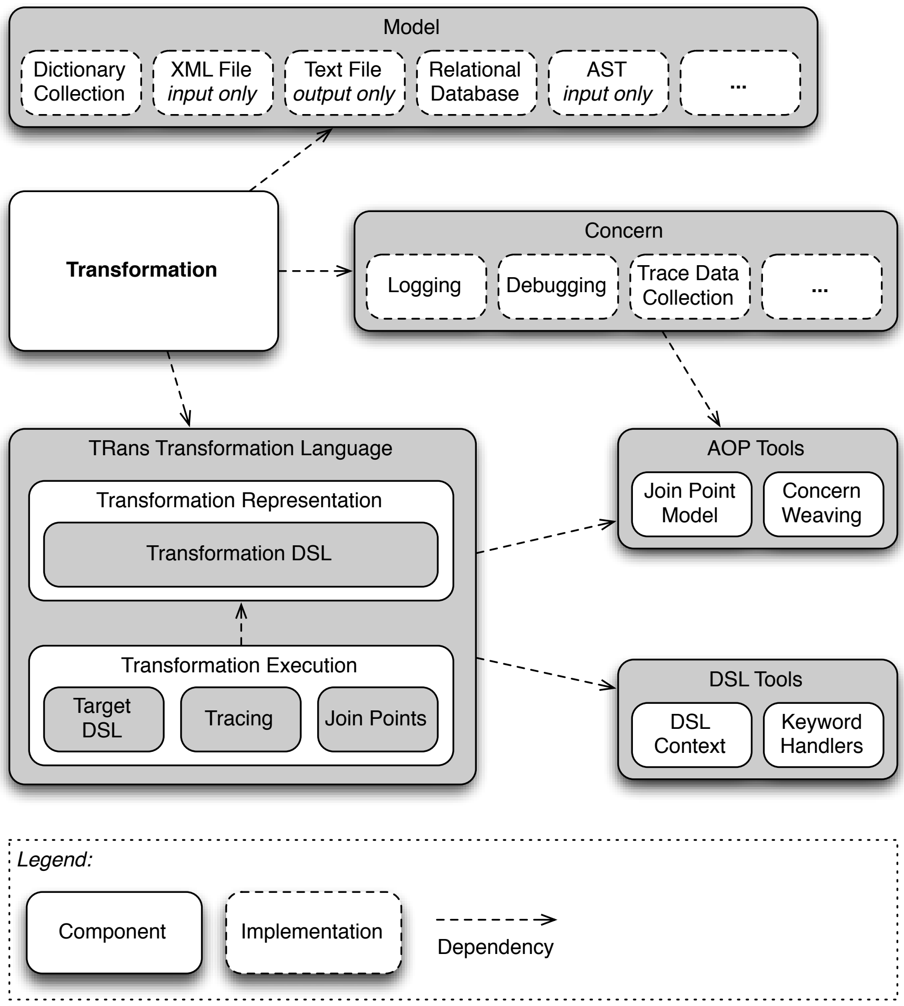


Figure 6.1: Architectural logical overview diagram

sources, targets, helpers, rules and concerns that make up the transformation. The parts of the DSEL that specify the target elements are not evaluated by the representation but stored for evaluation by the transformation execution. The transformation execution also implements tracing (including topological sort) and specifies join points.

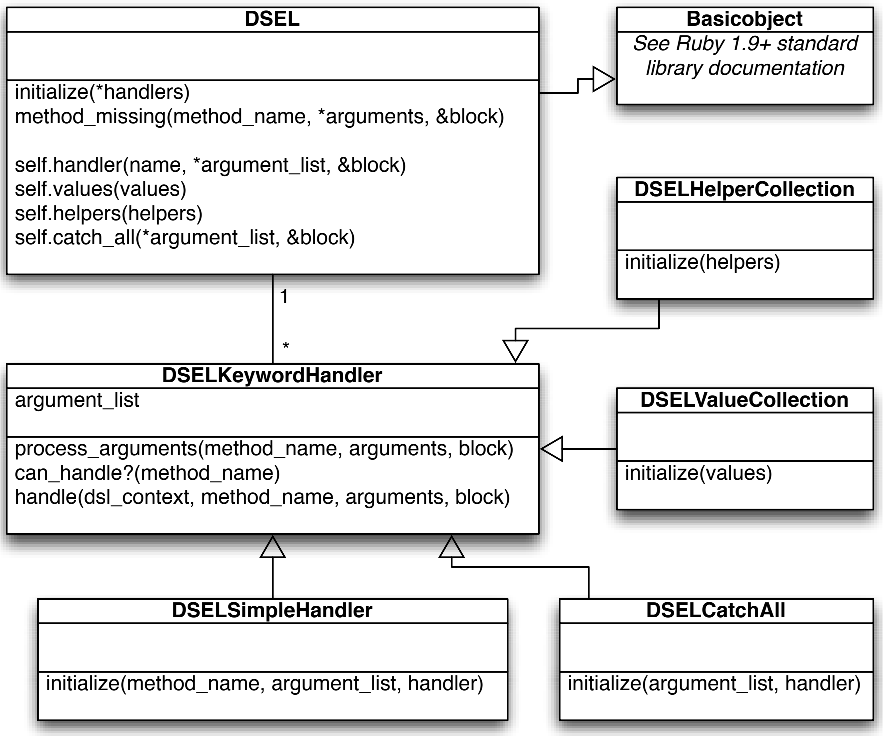
6.2 Design

This section describes the design of the DSEL tools, the AOP tools, the transformation representation, the transformation execution en the model components. Class diagrams are provided for each component and a sequence diagram is provided for each algorithm.

6.2.1 DSEL Tools Component

The DSEL Tools component is responsible for providing a context in which a DSEL can be evaluated and handling the use of methods that are defined in this context. Figure 6.2 shows a class diagram of this component. The DSEL class is responsible for providing the context. It inherits from BasicObject in order to have a very clean namespace. It is initialised with a collection of keyword handlers that support the methods 'can_handle?' and 'handle'. Since the DSEL object has a clean namespace and doesn't define any methods, all methods invoked in its context are handled by its 'method_missing' implementation.

The 'method_missing' implementation invokes the 'handle' method on the first keyword handler that responds positively to a call to the 'can_handle?' method. The handler then, is responsible for checking and parsing the arguments that where passed to the keyword and doing the actual work.



DSEL
initialize(*handlers)
method_missing(method_name, *arguments, &block)
self.handler(name, *argument_list, &block)
self.values(values)
self.helpers(helpers)
self.catch_all(*argument_list, &block)
Basicobject
See Ruby 1.9+ standard library documentation
DSELHelperCollection
initialize(helpers)
1
*
DSELKeywordHandler
argument_list
process_arguments(method_name, arguments, block)
can_handle?(method_name)
handle(dsl_context, method_name, arguments, block)
DSELValueCollection
initialize(values)
DSELSimpleHandler
initialize(method_name, argument_list, handler)
DSELCatchAll
initialize(argument_list, handler)

Figure 6.2: Class diagram of the DSEL Tools component

Specifying A DSEL

Currently, four types of keyword handlers are implemented. The simple handler calls a block of code when its keyword is encountered. The block of code is executed in the context where the DSEL is defined and can so be used to manipulate variables from that context and extract information from the DSEL code that is being evaluated. The catch-all handler does very much the same thing but always responds true to 'can_handle?' and passes the keyword as an extra argument into its block of code. The catch-all handler should always be inserted at the end of the handling chain because handlers after it will never be invoked. The values handler simply returns a value to the DSEL for each keywords it supports and can be used to pass a collection of values into the DSEL. The helpers handler is like the values handlers but passes a collection of methods into the DSEL that are invoked in the context of DSEL itself.

The best way to understand the DSEL Tools component is to have a look at a DSEL definition. The code in figure 6.3 was taken from the 'generate_target_element' method in the 'TransformationElement' class and shows how the DSEL Tools component is applied to create part of the the TRans DSEL. The context that is being defined here is for generating the attributes of the target elements and uses all four types of handler. The simple handler is used to define the trace method, the values handler to make source models available, the helpers handler to make transformation helper rules available and the catch-all helper to catch all other keywords as defining attributes of the target element. Please note that the simple handler and the catch-all handler support the passing of an argument list on creation that is checked automatically when the keyword is used in the DSEL.

```
DSEL.new(
  DSEL.handler(:trace, Hash) do |options|
    execution.trace options
  end,
  DSEL.values(source_models),
  DSEL.helpers(helpers),
  DSEL.catch_all(:any) do |method_name, value|
    @target_element[method_name] = value
  end
).instance_exec @source_element, &@target.generator
```

Figure 6.3: DSEL definition code taken from 'TransformationElement' class

```
build(:database_model, 'Column') { |o|
  name o.target.name + 'Id'
  owner trace(source: inheritance_root(o.source), target: 'Table')
  type 'Integer'
}
```

Figure 6.4: Sample DSEL code for DSEL definition in figure 6.3

Writing The DSEL

Figure 6.4 shows some code written in the DSEL defined by the DSEL definition code in figure 6.3. The block of code passed to the build method is stored in the '@target.generator' and evaluated in the context of the DSEL using the 'instance_exec' method, passing in the source element as an argument to the block. It shows the use of the trace method, into which a hash of options is passed, as declared by the argument list in the DSEL definition. The 'name', 'owner' and 'type' methods are caught by the catch-all handler and respectively define the 'name', 'owner' and 'type' attributes of the target element. 'inheritance_root' is a helper defined by the transformation and was made available using the helpers handler. Since the helper is evaluated in the context of the DSEL, is has access to itself and to the source models in order to recursively determine the inheritance root of the class that is passed into it.

DSEL Evaluation

Figure 6.5 shows a somewhat simplified sequence diagram of the evaluation of DSEL's in our example case. The 'generate_target_element' method first uses the class (static) methods of the DSEL class to define the transformation, this part is omitted in the diagram. After the DSEL is constructed, 'instance_exec' is called on it to evaluate the DSEL code in its context. This causes control flow to be passed to the DSEL code in the transformation. Whenever a method is used in the DSEL, 'method_missing' gets invoked on the DSEL object, which in turn calls handle on its first handler that is able to handle the keyword. This handler processes the arguments and optionally calls a block in the original DSEL definition in 'TransformationElement' when it can manipulate local variables and instance variables.

6.2.2 AOP Tools Component

The AOP Tools component is responsible for weaving additional concerns into the transformation execution code. The 'TransformationExecution' class inherits from 'JoinPointModel' and any concerns that need to be weaved into the transformation execution have to inherit from 'Concern'. The enables the transformation execution to internally define join points by calling the 'join_point' method and enclosing the code that constitutes the join point in the block passed to the method. All the concerns have to do is define methods that start with either 'before_', 'after_' or 'around_', following by the name of the join point and these methods will be called automatically when the join point is passed. Additionally, concerns can define 'before_all', 'after_all' and 'around_all' to be invoked for every join point. Figure 6.6 shows a class diagram of the AOP Tools component.

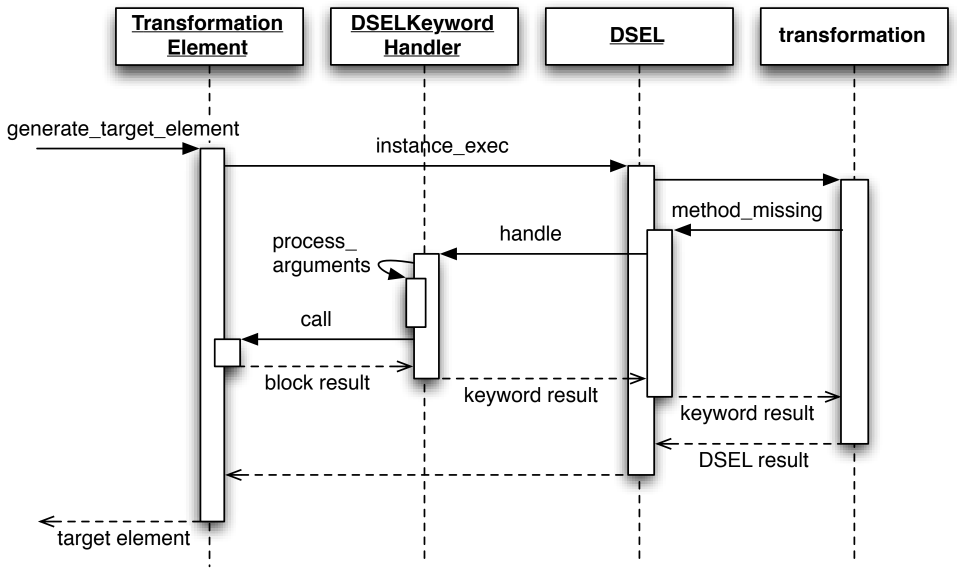


Figure 6.5: Sequence diagram of handling DSEL keywords

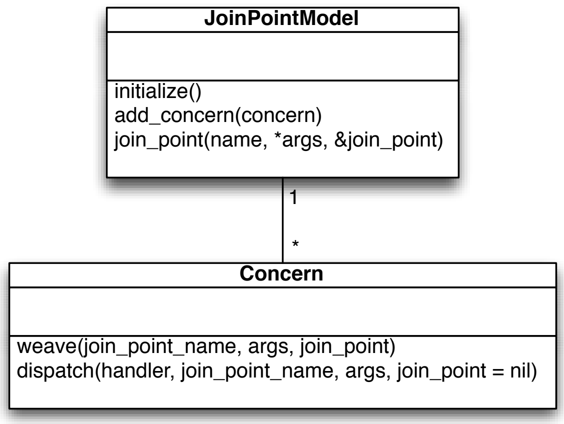

JoinPointModel
initialize()
add_concern(concern)
join_point(name, *args, &join_point)
1
*
Concern
weave(join_point_name, args, join_point)
dispatch(handler, join_point_name, args, join_point = nil)

Figure 6.6: Class diagram of the AOP Tools component

Figure 6.7 shows the simplified sequence diagram of a join point called 'trace' in a case where two concerns where added to the join point model, one of which defined 'before_trace' and the other of which defines 'around_trace'. Firstly, the weave method is called on both concerns. The weave method takes a block as its argument and returns a new block containing the extra functionality defined by the concern and a call to the original block. The resulting block is then invoked in its original context by the 'join_point' method, resulting in calls to the relevant callback methods in the concerns.

6.2.3 Transformation Representation Component

The Transformation Representation component is responsible for evaluating all of the transformation DSEL code, except for source expressions and target generators. It stores all source model declarations, target model declarations, rules, helpers and concerns. The transformation representation and execution are decoupled like this to enable consecutive transformations using a single representation. Also, in theory, it allows another transformation execution algorithm be used to execute the same transformation. The 'Transformation' class' 'self.load' methods uses the DSEL Tools component to evaluate the transformation code in a way similar to the code sample in figure 6.3. Figure 6.8 shows the class diagram for

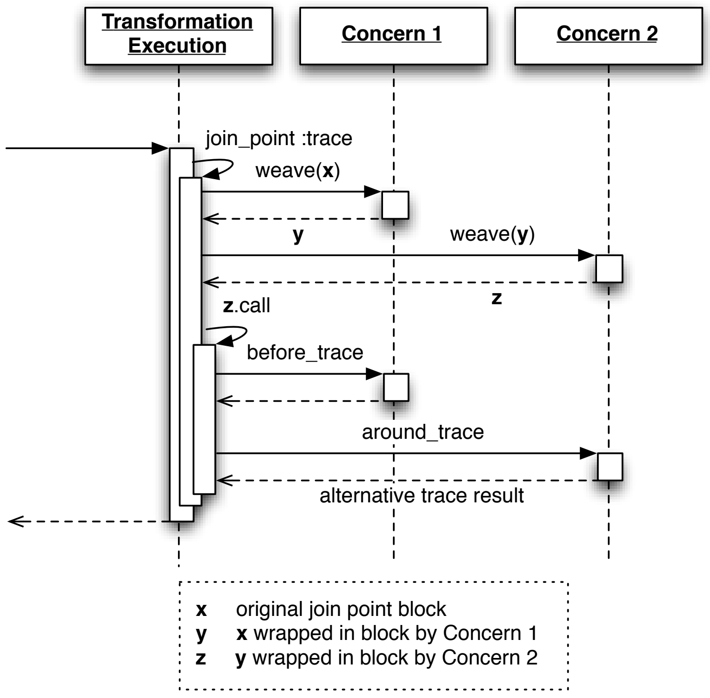


Figure 6.7: Sequence diagram showing the weaving of concerns for a join point

the Transformation Representation component.

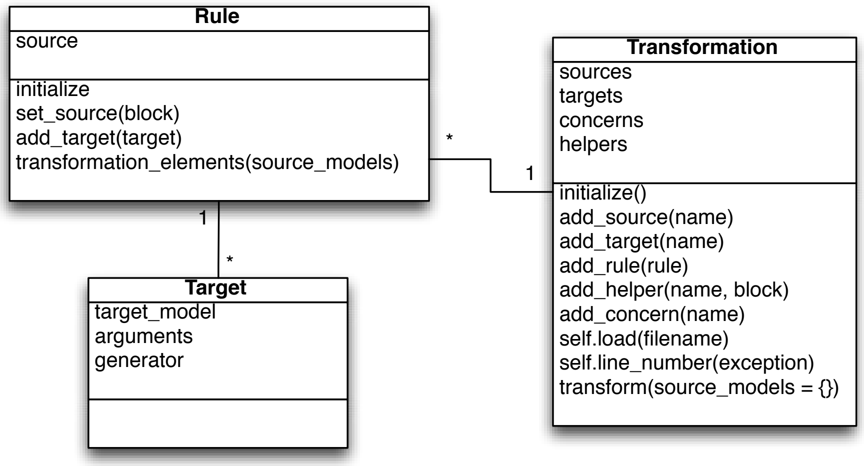
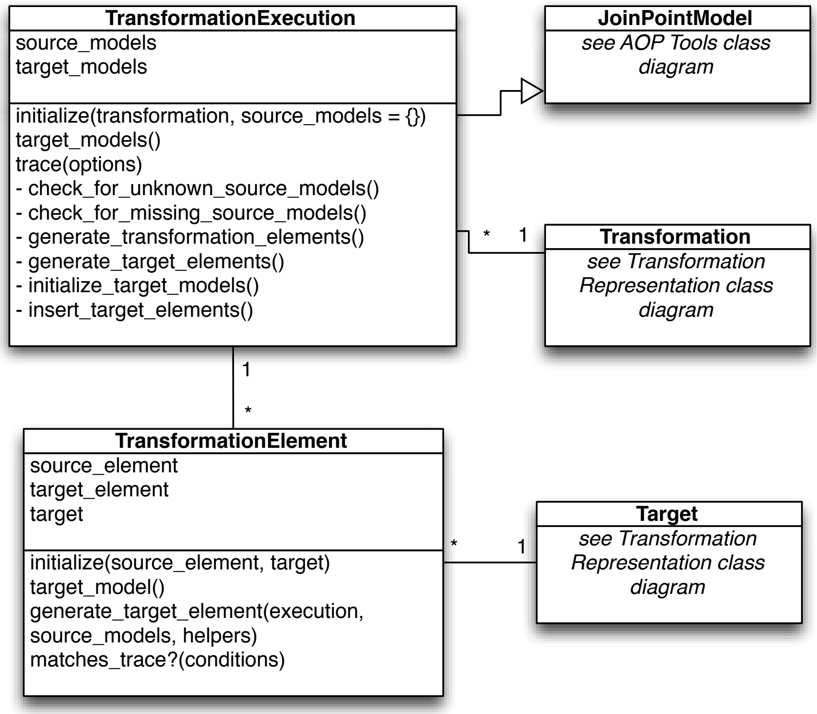


Figure 6.8: Class diagram of the Transformation Representation component

6.2.4 Transformation Execution Component

The Transformation Execution component is responsible for implementing the model transformation algorithm which includes the topological sort algorithm and tracing functionality. The class diagram for the component is shown in figure 6.9. The 'TransformationExecution' class implements the general transformation steps. The 'TransformationElement' class represents the transformation from a single source elements to a single target element. It implements the evaluation of the target element generator DSEL that can be seen in figures 6.3 and 6.4.



TransformationExecution
source_models
target_models
initialize(transformation, source_models = {})
target_models()
trace(options)
- check_for_unknown_source_models()
- check_for_missing_source_models()
- generate_transformation_elements()
- generate_target_elements()
- initialize_target_models()
- insert_target_elements()
JoinPointModel
see AOP Tools class diagram
Transformation
see Transformation Representation class diagram
*
1
1
*
TransformationElement
source_element
target_element
target
initialize(source_element, target)
target_model()
generate_target_element(execution, source_models, helpers)
matches_trace?(conditions)
Target
see Transformation Representation class diagram
*
1

Figure 6.9: Class diagram of the Transformation Execution component

Figure 6.10 shows details about the transformation algorithm. Whenever 'transform' is called on a transformation, it creates an instance of 'TransformationExecution'. It then requests the target models (result of the transformation) from the instance, setting in motion the transformation process. The transformation process consists of six steps, as can be seen in the diagram from the six consecutive methods the transformation execution calls on itself from within the 'target_models' method. While most of these steps are pretty self-explanatory, the creation of transformation elements and the creation of target elements require some explanation. Each of transformation elements that is created signifies a single target element in any of the target models. They are created by evaluating the source expression for every rule and creating a transformation element for each of the resulting source elements, for each of the rule's targets.

The creation of target elements is done by the transformation elements, exactly one target element per transformation element. The catch here is that the generation of one target element can lead to the creation of an arbitrary number of other target elements by tracing. When a target element generator executes a trace, this trace is passed on to the execution object en matched to each of transformation elements. Each transformation element that actually matches the trace is then generated before the original element is, possibly executing more traces and in turn possibly generating more target elements. Automatically ordering the generation of target elements this way is called a topological sort algorithm and it is why TRans can be used as an entirely declarative language.

6.2.5 Model Component

The model component is responsible for storing models. For a model to be transformed using TRans it has to provide some kind of querying language to the transformation. The transformation can query any source model from the source clauses of its rules using the querying language provided by that source model. TRans automatically iterates over the result of such a query and hands control back to the target clauses of the transformation rules where it uses the instantiation mechanism provided by the model to generate target model elements. As was mentioned in the introduction of this chapter, TRans is completely unconcerned with the querying and instantiation mechanisms of the model.

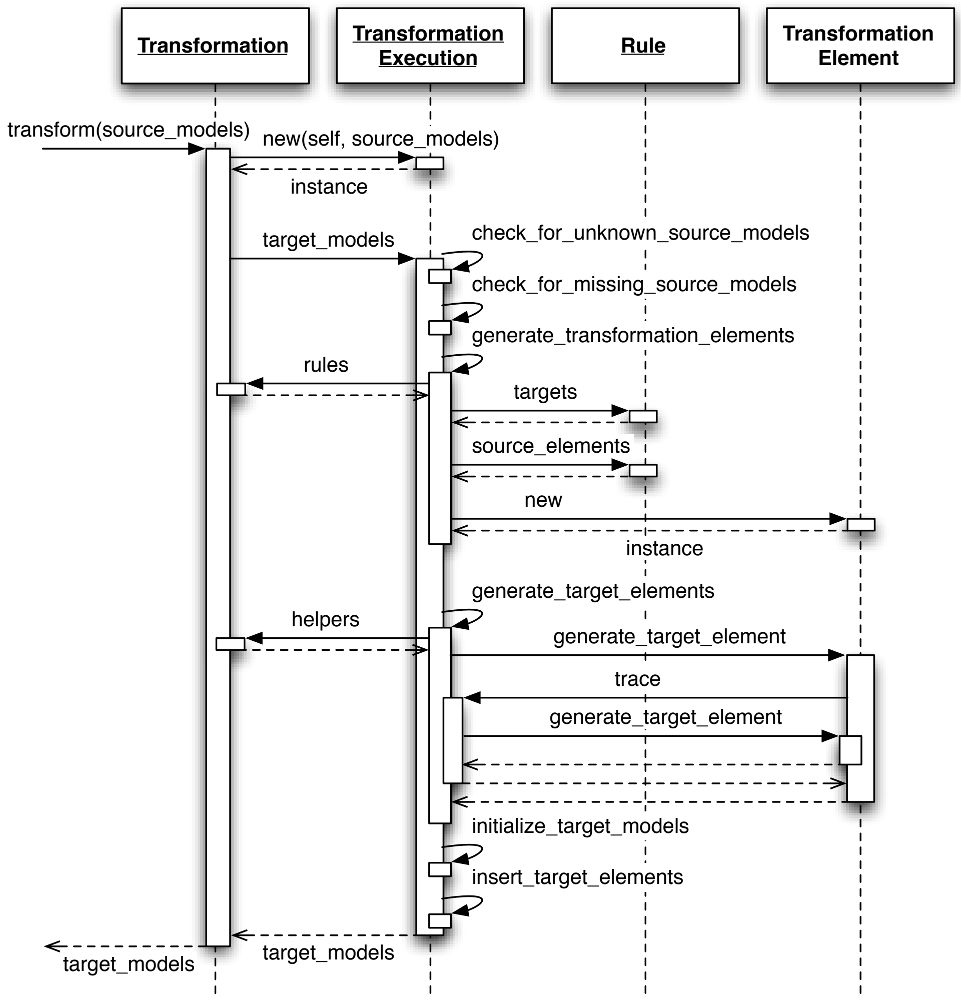


Figure 6.10: Squence diagram of the Transformation Execution algorithm

We implemented a model which is internally a collection of dictionaries (elements) and can be queried using a subset of the OCL language, using Ruby syntax. Additionally we implemented a different kind of model that supports an instantiation mechanism for code generation.

6.3 Implementation

This section describes some best practice that influenced the implementation of TRans. We provide these because they are the motivation for many of the design decisions this chapter presents.

- **Write sample transformation first**

  For every feature we add to the TRans transformation language, we write sample transformations that utilise the feature before actually implementing the feature itself. The first advantage to this method is that is constitutes a user-centric approach, minimising the negative effect implementation as an DSEL has on the syntax of the language. Secondly, these transformations can act as test cases to test the feature.

- **Convention over configuration**

  Many arguments and options that are passed into methods should be optional, allowing for a lot of flexibility where required while providing sensible defaults where this flexibility isn't required.

This excellent principle is applied extensively throughout most Ruby code and TRans should be no exception.

- **Don't repeat yourself (DRY)**

  Wherever duplication is encountered it should be removed, even when meta-programming is required to do so. This leads to a concise and adaptable implementation. This is another principle that is followed by most Ruby programmers.

- **Ruby 1.9 features**

  TRans can use features of the new Ruby 1.9 version where needed because it is a academic prototype project and none of its code could go into production before 1.9 is the stable version of Ruby.

6.4 Conclusions

In this chapter we discussed the architecture and design of our MDE environment. It shows how the TRans transformation language is positioned amongst the other components of the environment en how each component works internally.

The two most important components of our implementation are the TRans transformation language and the implementation of model. These two components are not directly coupled, except for the fact that TRans makes a few minor assumptions about models that are all optional. Any particular transformation is however strongly coupled with both its source and target model types and the TRans transformation language. This makes it possible to transform any kind of model implementation using trans and also to transform those model implementations using another transformation language. This loosely coupled architecture is enabled by the dynamic nature of the Ruby language.

Chapter 7

Evaluation

In this chapter we will evaluate the TRans model transformation language and its development. We observed a number of advantages and disadvantages to our approach during the development and testing of TRans. The advantages and disadvantages are respectively enumerated and explained in sections 7.1.2 and 7.1.3. In 7.2 we compare TRans to RubyTL [19] since that is the MTL most similar to TRans. RubyTL represents the current state of the art for doing model transformations using DSEL's.

7.1 Comparison to Classic Language Design Approach

This section describes the advantages and disadvantages of implementing a MTL as a DSEL compared by implementation using a classic language design approach. These advantages and disadvantages where observed during the development and testing of the TRans MTEL.

7.1.1 Achieved Reuse

As stated in Chapter 2, the advantages and disadvantages of our approach are all related to the central concept of reuse. Our goal of reusing existing language features in Ruby as part of the implementation of TRans worked out really well. As can be seen in Chapter 5, TRans is a fully functional and feature rich MTL, even though it was implemented in only about 600 lines of Ruby code. The Ruby language and standard library provide TRans with a parser, interpreter, debugger, arithmetic, expressions, imperative constructs, exception handling, recursive functions/methods and many other things. Implementing these features for a traditional language implementation would have required many times the effort it took to implement all of TRans. By developing TRans as a runtime DSEL we achieved maximum functionality with a minimal amount of effort.

The rest of this section will explore whether the other characteristics changed the way we predicted in Chapter 2. We also try to bring across the extent to which certain characteristics where impacted. Unfortunately we where unable to formalise the extent of impact by proper measurement. Impacted characteristics are grouped by wether they are an advantage or a disadvantage. An advantages can be either a increased desirable characteristic or a decreased undesirable characteristic. Disadvantages are regarded analogously.

7.1.2 Advantages

In Chapter 2 we predicted that predicted that the desirable characteristics of familiarity, maturity, reliability, library availability, maintainability and adaptability would all increase. Additionally we predicted that the undesirable characteristic of implementation effort would decrease. This section describes to what extend they actually did.

Implementation Effort

TRans was developed over a period of about one accumulative month of programming by a single person. The codebase if about 600 lines of code. According to our experience, creating a language from scratch, including writing a parser, runtime, and all the other language features requires many times this effort. Therefore implementation effort was significantly reduced.

Familiarity

Another big advantage of the approach is familiarity. Using a subset of the Ruby syntax provides TRans with instant familiarity to thousands of programmers that are already familiar with Ruby. Since Ruby mimics a lot of language features from C, PHP and Java, some familiarity is provided to users of these languages as well. This same effect could be achieved by mimicking an existing language but that would take extra effort, while for TRans it comes for free. Additionally, if Ruby changes, TRans changes with it automatically if the user of TRans decides to use this new version of Ruby.

Maturity and Reliability

Since Ruby is a relatively mature programming language (it has been in development as long as Java), all features that TRans borrows from it are tried and tested. Every major programming language in history has taken the better part of a decade to mature and become adopted on a large scale. Most features in TRans are mature and production ready right from the off, since they are simply features of Ruby.

Library Availability

Apart from features that TRans borrows from the Ruby language and its standard library, any of the libraries that where developed for the Ruby language are also available in TRans. Ruby is a relatively mature language and libraries exist for pretty much any technology.

Maintainability and adaptability

The implementation of TRans is much smaller than an implementation of a similar language using the traditional approach would be. This leads to increased maintainability and adaptability because a small codebase has less regression problems and is easier to get familiar with. This makes it an ideal tool for research on model transformation languages. It is relatively easy to add features and try out new things because you don't need to go through extra layers of implementation like parsing.

7.1.3 Disadvantages

In Chapter 2 we predicted that predicted that the desirable characteristic of flexibility would decrease. Additionally we predicted that the undesirable characteristic of superfluous functionality would increase. This section describes to what extend they actually did.

Flexibility

The usage of the Ruby syntax can be interpreted as an advantage because it provides familiarity. However, it also places a lot of restrictions on what can be done with the language. It can be hard to come up with syntactical constructs that work well with both Ruby and the model transformation domain. Even within the syntactics Ruby allows, a balance must be found between what is desirable from a model transformation perspective and the amount of effort it will take to implement in Ruby.

Superfluous Functionality

The reuse of language features that was mentioned as an advantage also has downside. You don't get to cherry-pick the language features that are desirable in a MTL, instead you just get everything. While this adds to familiarity for Ruby users, it also means that at a higher level of abstraction there is no telling what a TRans transformation does or doesn't involve. It is very well possible to embed anything, including malicious code in any TRans transformation. This makes TRans, like Ruby less preferable for formal and process oriented software development.

Coupling to Host Language

A DSEL is strongly coupled to its host language because it relies so heavily on it, therefore the TRans MTEL is strongly coupled to the Ruby language. In the traditional approach to language design a new implementation of the language could be written in a different language. With the DSEL approach this is only possible if most of the host language is re-implemented as well.

7.1.4 Comparison Table

Table 7.1 summarises this section by comparing the classic approach to the DSEL approach we followed based on the criteria that where introduced in chapter 2. Wherever one approach has a definitive advantage over the other, that approach's characteristic is *emphasised*.

Criterium	**Classic Approach**	**DSEL Approach**
Implementation effort	Large effort	*Much smaller effort*
Flexibility	*Fully flexible*	Restricted flexibility
Familiarity	By mimicking a language	*Comes for free*
Maturity	None	*Reused language features are mature*
Available Libraries	None	*All host language libraries*
Maintainability and adaptability	Lower	*Higher*
Superfluous functionality	*None*	Unwanted host language functionality

Table 7.1: Comparison table for the classic approach to language design versus the DSEL approach

7.2 Comparison to RubyTL

The current state of the art in model transformation using a DSEL is RubyTL [19], another MTEL based on the Ruby language. This section examines the differences between RubyTL and TRans by looking at the advantages and disadvantages of TRans compared to Ruby TL.

7.2.1 Advantages

The main advantage of TRans is that conceptually it is cleaner than RubyTL because it implements only a single model transformation paradigm and is decoupled from a specific notion of a modelling language. Additionally, the practice of creating DSEL's in Ruby has advanced since implementation started on RubyTL which means TRans uses some more advanced methods that lead to a cleaner syntax.

AOP Support

TRans supports AOP features where RubyTL does not. AOP features allow the transformation execution algorithm to be augmented or changed by applying advice. Advice can be supplied from within a transformation definition, but can also be added at execution time. Because TRans is a DSEL that uses Ruby, a general purpose programming language is available to write advice, which makes this an especially powerful concept. Example applications are tracing, logging, debugging and execution algorithm adjustment. Since concerns containing advice can be imposed at execution time, there is great potential for reuse of these functionalities.

Independent of Modelling Language

The TRans language itself assumes almost nothing about a specific modelling language implementation. This means it can be used with different standards, your own implementation or even with languages that are not modelling languages in the traditional sense. This was done by moving the model querying and model synthesis parts out of TRans and into the modelling languages themselves. TRans simply delegates these parts of the model transformation to the modelling language for evaluation. The advantage of this approach is that the TRans language is easily reusable for the transformation of models expressed in different languages/standards. Additionally, it is possible to transform models expressed in one language into models expressed in another.

Comprehensive Approach

TRans's indifference to different modelling languages allows it to be a comprehensive approach to MDE. Everything is a model transformation, including incorporating different kinds of data into the model driven workflow and code generation. So, where RubyTL has an entirely different DSEL for code generation, TRans just uses a regular model transformation with a code generation specific modelling language for the target model.

More Advanced DSEL

Since RubyTL has been around for a while, it doesn't employ the techniques for DSEL creation that have recently become popular in the Ruby community. This allows the TRans syntax to be a bit more concise and suffer less from being a subset of Ruby.

Single Paradigm

Mainly due to time constraints, TRans has made some firm decisions about the model transformation paradigms it supports. The two decisions that are most influential on the design of the language are the clear separation of source and target models and transformation ordering based solely on the topological sort algorithm. The separation of source and target models means that transformations cannot update or refine models. The topological sort means that transformation don't have to be divided into steps or phases but trace statements often have to be written differently because selecting on target attributes can more easily result in trace cycle errors. While these choices may be interpreted as restrictive they also provide a very clear structure and concept to every model transformation.

Cleaner Implementation

As we mentioned, TRans implements a very specific subset of model transformation features and doesn't worry about things that (according to us) belong in a separate modelling language component. This means that the implementation of the TRans language is very small and clean, making it much more adaptable and maintainable than RubyTL. This adaptability makes it an ideal tool for research into model transformation language features.

7.2.2 Disadvantages

This section describes the disadvantages of TRans when compared to RubyTL. The main disadvantage of TRans is the lack of formal extension points.

No Formal Extension Points

One thing RubyTL does exceptionally well is the definition of extension points. New types of transformation rules and other language features can easily be implemented by inheriting from certain classes in the implementation. While adding language features to TRans is also pretty trivial, it does not provide formal documented extension points for doing is, which is a serious shortcoming.

Less Feature Rich

Whereas TRans chose to implement a very specific model transformation paradigm, RubyTL combines features from different approaches. This makes RubyTL a more flexible language that probably appeals to a wider audience.

7.2.3 Comparison Table

Table 7.2 summarises this section by comparing RubyTL to the TRans MTEL based on several criteria. Wherever one implementation has a definitive advantage over the other, that implementation's characteristic is printed in italic text.

Criterium	**RubyTL**	**TRans**
Modelling language dependency	Dependent	*Independent*
Code generation	Different DSEL	Integrated
DSEL creation techniques	Simple	*More advanced*
Transformation paradigm	Mixed paradigm, Feature rich	Single paradigm, Opinionated
Implementation	Large	*Smaller, Simpler*
Formal extension points	*Available*	None
AOP support	None	*Substantial*

Table 7.2: Comparison table for RubyTL versus TRans

7.3 Conclusions

In this chapter we evaluated advantages and disadvantages of the MTEL approach in relation to the more traditional way of language development. Additionally, we evaluated the TRans language in relation to RubyTL, which is the current state of the art in the MTEL domain. We think that in both cases the advantages to our approach outweigh the disadvantages, particularly in the DSEL versus traditional language design debate in relation to MDE.

Chapter 8

Conclusions

This chapter describes conclusions that can be drawn from our research. Section 8.1 provides a short summary of our research. Section 8.2 provides the answers we have found to the research questions that where presented in Chapter 2. Sections 8.3 and 8.4 respectively present the contributions of our research and our recommendations for future work in this field of study.

8.1 Summary of Concepts

In this report we presented our research into MTEL's. The term MTEL (Model Transformation Embedded Language) was chosen by us to describe a MTL (Model Transformation Language) developed as a DSEL (Domain Specific Embedded Language). This approach is just one of several possible approaches that aim at the reuse of language features of a general purpose programming language for the development of a MTL. The DSEL approach provides the most potential for reuse of language features with the least amount of effort. This means that if this method is viable (any disadvantages can be accepted or controlled), it is the most desirable alternative approach. We established this in Chapter 2 and scoped our research accordingly. The main goal of this research became to develop a MTEL that constitutes a valid prove of concept and to evaluate its architecture, advantages and disadvantages.

In order to establish what constitutes a valid prove of concept in the MTL domain, we had to perform some literature research. The resulting survey of the MDE (Model Driven Engineering) and more specific MTL domain was presented in Chapter 3. A subset of the observed features was chosen in such a way as to minimise development effort but provide a valid prove of concept. The features that where chosen are declarative transformation definition, multiple source and target models, recursive helper rules, a subset of OCL semantics, some aspect oriented features, tracing functionality and automatic execution ordering based on encountered traces.

As the host language for our DSEL we chose Ruby. It is to the best of our knowledge the only language that offers extensive facilities for the creation of DSEL's. In order to gain a full and thorough understanding of these facilities we performed literature research on this subject as well. The resulting survey of the Ruby programming language and in particular the aspects that can be used for the creation of a DSEL where presented in Chapter 4. The features that proved most important for the creation of our prototype are code blocks (closures) and the ability to evaluate them in a specific context, flexible syntax, hash literals and undefined constant and method definition.

Based on the framework of concepts described in our literature research we designed and implemented our MTEL, called TRans. Its features and usage where described in Chapter 5. TRans is a fully functional MTEL that implements all the MTL features we chose as a Ruby DSEL. Chapter 6 describes the architecture, design and implementation of the language. Since the TRans interpreter is a highly dynamic Ruby program, it is not always possible to describe its workings using traditional methods such as class and sequence diagrams. While we did or best to describe TRans's internals using natural language, we also encourage anyone interested to examine the source code.

Based on our experiences developing in and using TRans, we can conclude that implementing a MTL as an DSEL is an approach that yields many advantages, the most notable of which are reuse of mature host language features, familiarity to host language users, library availability, adaptability and maintainability. Adaptability and maintainability are observed as the result of having a much smaller codebase due to the large amount of reuse. These advantages make the approach especially well suited for situations where resources are limited.

The main disadvantages that where observed are restrictions imposed by usage of the host language syntax and the inability to hide unneeded functionality from transformation implementations. These disadvantages make the approach less suited for a situation in which the MTL needs to have a specific syntax or when restrictions need to be placed on what operations a transformation is allowed to perform.

8.2 Research Questions

This sections provides the answers the last three of our six research questions from Chapter 2, based on the evaluative observations that where presented in this chapter.

Q1 What host language should we use and because of which requirements?

Q2 Which facilities in the host language can be used to create a DSEL?

Q3 What features do we want our MTL to support in order to make it a valid proof of concept?

Q4 To what degree where we able to achieve the reuse we where aiming for?

Q5 What advantages and disadvantages did the achieved reuse yield?

Q6 How does our implementation compare practically to implementations using the same approach?

8.2.1 Research Question 1

After deciding we wanted to develop our MTL as an DSEL, the most important criterium for choosing the host language became it's ability to support runtime DSEL creation. This made Ruby a fairly obvious choice. To the best of our knowledge, no other language has such extensive facilities for the creation of DSEL's. Additionally, the host language should not have a syntax that would seem exotic and foreign to the domain of model transformation. Since most work in the area has been done in Java, Ruby is a perfect fit because it allows programs to be written in a way very similar to Java or C.

8.2.2 Research Question 2

In Chapter 4 we did an extensive study of the features that Ruby offers for the creation of DSEL's. Many of these features where used for the development of TRans, as can be seen in Chapter 5. The most important feature for us was the ability to capture blocks of code in variables and evaluate them in a specific context. Additionally, we used hooks to capture the use of undefined keywords and constants. Some more common language features that also proved extremely useful where dynamic typing and easy introspection, hash literals and operator overloading.

8.2.3 Research Question 3

We presented a survey of MTL features in Chapter 3. We decided to implement a MTL based around the principle of tracing and the topological sort algorithm because this seems to be to most modern and clean approach. Additionally we support modelling language agnosticism and AOP features for model transformation, upon which reflective features could be implemented. Because TRans is a prove of concept, we decided not to implement the somewhat more trivial features such as phasing, rule inheritance, and different types of rules.

8.2.4 Research Question 4

We established that a very large degree of reuse is possible by building a prototype. Our prototype, TRans, is a fully functional and feature rich MTL, even though it was implemented in only about 600 lines of Ruby code. The Ruby language and standard library provide TRans with a parser, interpreter, debugger, arithmetic, expressions, imperative constructs, exception handling, recursive functions/methods and many other things.

8.2.5 Research Question 5

This section describes what quality characteristics where changed in what ways by having this high degree of reuse.

Disadvantages

As we mentioned in Section 7.1.3, the most notable disadvantages to the chosen approach are restrictions in the host language syntax and superfluous host language functionality.

The biggest challenge in the development of TRans was choosing syntactical constructs that are possible in Ruby that map well to the model transformation domain but also allow a clean implementation of the DSEL. When implementing a MTL using the traditional approach to language development is would not be an issue and as such me note it as a serious disadvantage to our approach.

Another disadvantage is our inability to limit of functionality that is exposed to the author of a model transformation. While TRans does its best to keep the contexts in which the different parts of a model transformation are evaluation as clean as possible, it is always possible to gain access to the full spectrum of Ruby functionality. This includes the ability to create malware. This problem can be partially overcome by running model transformation in a sandbox Ruby environment, which is relatively easy to do.

Advantages

We managed to reuse Ruby language features for almost all parts of the TRans language. There are no features that where already available in Ruby that we had to re-implement for TRans. This level of reuse allowed us to really focus on the model transformation specific language features of TRans. For this reason we believe that implementation as a DSEL is definitely the way forward for MTL's, in particular in an academic context. For a process oriented enterprise environment the approach may be less suitable because of the lack of control that can be exercised on the functionality of model transformations written in a MTEL.

As we mentioned in Section 7.1.2, apart from our goal of language feature reuse, several other advantages of the approach where observed. The most notable of which are familiarity, maturity, library usage and maintainability.

Because so many of the features of TRans are borrowed from the Ruby language they are instantly familiar to programmers that are familiar with Ruby. This softens the learning curve for TRans for these programmers. Borrowing these features from the Ruby language also means they are more mature, simple because they have been tried and tested for many years.

An unforeseen advantage of TRans is it's ability to directly use Ruby libraries. Ruby provides libraries for every notable technology out there and also for other things that are desirable in model transformations like natural language inflections.

Lastly, the approach leads to a very clear separation of general purpose and model transformation specific language features. TRans, containing the model transformation specific features, being dependant on Ruby but not the other way around. This allows TRans to be a small and cleanly implemented language that easy to adapt and maintain.

8.2.6 Research Question 6

We where able to introduce some significant advantages over RubyTL; AOP support, independence of modelling language, a comprehensive approach, a more advanced DSEL, single paradigm and a cleaner implementation. The most important of which are described in section 8.3.

8.3 Contributions

This section enumerates the parts of this research that are new in the field of DSEL's for model transformations.

8.3.1 Architectural Improvements

TRans's architecture offers a very clean and extensible approach to model transformation. It allows the model querying and instantiation mechanisms to be a separate DSEL that is all but completely decoupled from the actual model transformation DSEL. This new architecture allows a comprehensive approach to model transformation (mentioned separately) and allows research into new model transformation techniques to be much more focussed on specific parts of the approach.

8.3.2 Comprehensive Approach

Because TRans is so indifferent to the interpretation of the model concept everything can be a model whit its own mechanisms for instantiation and querying. This means that any type of computer data, like for instance a code repository can be wrapped and used as a source or target model. TRans supplies the model declaration, declarative transformation definition using rules, the automatic rule ordering using the topological sort algorithm and tracing facilities to transformation of any type of model. This means these facilities can be used for code generation and the incorporation of arbitrary data into de MDE workflow as well as for model transformation in the classic sense.

8.3.3 AOP Functionality for Model Transformation

TRans features the ability to apply Aspect Oriented Programming in its model transformations. This ability is especially powerful since the full featured Ruby language can be used to write advice. The declarative model transformations written in the TRans MTL can be augmented with imperatively written functionality in this way.

8.3.4 Complete Survey of DSEL Techniques

This report presents a very complete survey of DSEL creation techniques in the context of the Ruby language. Many of these techniques have been applied for the creation of the TRans language which leads to it being a more concise DSEL for model transformations than RubyTL.

8.4 Future Work

This section describes the future work we propose in continuation of research into this field. The possibilities for future can be categorised in finding ways to overcome some of the disadvantages that where observed and the development of additional functionality using the same approach.

8.4.1 Disadvantages

The section describes possible ways to overcome the disadvantages of developing a MTL as a DSEL that where described in Chapter 7. The disadvantages are restrictive host language syntax and superfluous host language functionality.

Host Language Syntax

By far the biggest disadvantage of the DSEL approach is restrictions on the MTL syntax imposed by the syntax of the host language. One possible way to get around this is to use a different approach that allows less intense reuse such as compiling to the host language. However, we think that it is possible to design a general purpose programming language that supports the creation of DSEL's even better than Ruby. Ruby was not designed for the development of DSEL's but rather it is a byproduct of language features that where included for other reasons. A general purpose programming language that is specifically designed for the purpose of supporting DSEL's can potentially lift a lot of the syntax restrictions Ruby imposes.

Superfluous Functionality

Another disadvantage is the availability of unwanted host language functionality to model transformations. This disadvantage could also be overcome by having a host language that is designed specifically to support DSEL's. Such a language could potentially provide far more restricted or sandboxed contexts for he evaluation of model transformation code. Ruby does provide sandboxing, but only for an entire virtual machine, not for the evaluation of certain code.

Additional Functionality

While TRans is a quite full featured MTL, model transformation is only one part of model driven development. It would be a good idea to evaluate the viability of developing other MDE type languages in the same fashion. Possible candidates are a meta-modelling language and a workflow/dependency language in which the different artefacts in the MDE workflow can be described with their meta-models and their interconnecting transformations.

Bibliography

[1] Smalltalk.org. Retrieved December 25, 2008, from http://www.smalltalk.org/.

[2] *Review of "Software reuse: Architecture, process and organization for business success" by Ivar Jacobson, Martin Griss, and Patrik Jonsson, Addison-Wesley publishing Co., Reading, MA, 1997*, volume 37. IBM Corp., Riverton, NJ, USA, 1998. Reviewer-Pfleeger, Shari Lawrence.

[3] Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. 2005.

[4] Thouraya Bouabana-Tebibel and Mounira Belmesk. An object-oriented approach to formally analyze the uml 2.0 activity partitions. *Inf. Softw. Technol.*, 49(9-10):999–1016, 2007.

[5] Jean-Michel Bruel, editor. *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*. Springer, 2006.

[6] J. Clark. XSL transformations (XSLT) version 1.0 W3C recommendation 16 november 1999. Technical report, W3C - World Wide Web Consortium, 1999.

[7] Microsoft Corporation. Microsoft ironruby. Retrieved December 11, 2008, from http://www.ironruby.net.

[8] Microsoft Corporation. Microsoft .net. Retrieved December 15, 2008, from http://www.microsoft.com/net/.

[9] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45:621–645, 2006.

[10] et. al David Heinemeier Hansson. Ruby on rails, web development that doesn't hurt. Retrieved December 21, 2008, from http://www.rubyonrails.org/.

[11] Michel Ezran, Maurizio Morisio, and Colin Tully. *Practical software reuse*. Springer-Verlag, London, UK, 2002.

[12] Neil Ford. Advanced dsls in ruby. Presentation at RubyConf, 2008.

[13] Codehaus Foundation. Jruby, java powered ruby implementation. Retrieved December 25, 2008, from http://jruby.codehaus.org/.

[14] The Perl Foundation. The perl directory. Retrieved December 21, 2008, from http://www.perl.org/.

[15] Martin Fowler. Single table inheritance. Retrieved February 20, 2010, from http://martinfowler.com/eaaCatalog/singleTableInheritance.html.

[16] Jean Bézivin Ivan Kurtev Patrick Valduriez Frédéric Jouault, Freddy Allilaire. Atl: a qvt-like transformation language. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, 2006.

[17] Object Management Group. Unified modeling language specification. Retrieved February 14, 2010, from http://www.omg.org/technology/documents/formal/uml.htm, September.

[18] Paul Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[19] Jess Molina Jess Cuadrado and Marcos Tortosa. Rubytl: A practical, extensible transformation language. *ECMDA-FA*, LNCS 4088:158–172, 2006.

[20] S. Kent. Model driven engineering. *Lecture Notes In Computer Science*, 2335:286–298, 2002.

[21] Ivan Kurtev. *Adaptability of model transformations*. PhD thesis, Enschede, 2005.

[22] Jean-Philippe Lang. Rubyspec - the standard you trust. Retrieved December 20, 2008, from http://rubyspec.org/.

[23] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. *Satellite Events at the MoDELS 2005 Conference*, pages 139–150, 2006.

[24] Tim Lindholm and Frank Yellin. *Java(TM) Virtual Machine Specification, The (2nd Edition)*. Prentice Hall PTR, 2 edition, April 1999.

[25] Y. Matsumoto. Ruby programming language. Retrieved November 20, 2008, from http://www.ruby-lang.org/.

[26] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[27] OMG. Mof qvt final adopted specification. (OMG document formal/2005-11-01), June 2005.

[28] Evan Phoenix. Rubinius. ruby, the way it was meant to be. Retrieved November 19, 2008, from http://rubini.us/.

[29] Martin Thiede. Rgen, ruby modeling and generator framework. Retrieved February 20, 2009, from http://ruby-gen.org/.