Cláudio Ângelo Gonçalves Gomes

Bachelor

A Framework for Efficient Model Transformations

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador :	Vasco Amaral, Prof. Auxiliar, Universidade Nova de Lisboa
Co-orientador :	Bruno Barroca, Doutorado, Universidade Nova de Lisboa

	Júri:
Presidente:	[Nome do presidente do júri]
Arguentes:	[Nome do primeiro arguente] [Nome do segundo arguente]
Vogais:	[Nome do primeiro vogal] [Nome do segundo vogal] [Nome do terceiro vogal] [Nome do quarto vogal]

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

09, 2013

A Framework for Efficient Model Transformations

I dedicate this thesis to my grandparents and to my lifemate.

Acknowledgements

I want to thank my supervisor Vasco Amaral for sharing his experience and knowledge throughout this thesis and long before that, in lectures, conferences, etc...

I owe many of the ideas presented in this thesis to the discussions I had with Bruno Barroca and for that, thank you Bruno.

In order to be able to complete this thesis on time, I had to postpone my regular visit to my parents, grandparents and girlfriend. I thank them for being comprehensive and supporting me. My girlfriend took care of almost all the household tasks so that I had more time to write this thesis. I would have been really hard to accomplish this without her. Instead, it was fun.

Abstract

The reported productivity gains while using models and model transformations to develop entire systems, after almost a decade of experience applying model-driven approaches for system development, are already undeniable benefits of this approach. However, the slowness of higher-level, rule based model transformation languages hinders the applicability of this approach to industrial scales. Lower-level, and efficient, languages can be used but productivity and easy maintenance seize to exist.

The abstraction penalty problem is not new, it also exists for high-level, object oriented languages but everyone is using them now. Why is not everyone using rule based model transformation languages then?

In this thesis, we propose a framework, comprised of a language and its respective environment, designed to tackle the most performance critical operation of high-level model transformation languages: the pattern matching. This framework shows that it is possible to mitigate the performance penalty while still using high-level model transformation languages.

Keywords: Model Transformations, DSL, Language Design, Pattern Matching, Model Transformation Optimization, Model-Driven Development

Resumo

Os aumentos de produtividade reportados ao longo quase uma década de utilização de modelos e transformações entre modelos para desenvolver sistemas complexos constituem uma prova irrefutável dos benefícios desta abordagem. Conduto, a lentidão na execução de transformações expressas em linguagens de alto nível, baseadas em regras, prejudica muito a applicabilidade da abordagem. As linguagens de baixo nível, que são muito rápidas, podem ser usadas mas nesse caso não se consegue produtividade e fácil manutensão das transformações.

Este problema da abstração não é novo. Também as linguagens orientadas por objectos passaram pelo mesmo mas hoje em dia toda a gente as usa. Então porque é que não acontece o mesmo com as linguagens de transformação? O que falta fazer?

Nesta tese, propomos uma linguagem de transformação e respectivo ambiente de suporte, concebida para contornar o maior obstáculo à aplicabilidade industrial das linguagens de transformação de alto nível: a captura de padrões. Com a nossa abordagem demonstramos que é possível mitigar o problema da abstração.

Palavras-chave: Transformações de Modelos, Linguagens de Domínio Específico, Desenho de Linguagens, Captura de Padrões, Optimização de Transformações, Desenvolvimento baseado em Modelos

Contents

List of Figures

List of Tables

Listings

1

Introduction

The immersion of computer technology in a wide range of domains leads to a situation where the users' needs become demanding and increasingly complex. Consequently, the development of successful software systems also becomes increasingly complex.

Models are an important mechanism to deal with system complexity [**Schmidt:2006tg**; **Kurtev:2006uy**]. In the context of software engineering, models can be used to describe and prescribe entire systems. Therefore, a promising *divide-and-conquer* idea to break down this increasing complexity in software engineering, is to intensively use models during all stages of software development, as in the Model Driven Development (MDD) approach.

MDD is a software engineering approach that uses models and model transformations as first class citizens to create and evolve software systems [**Hailpern:2006vd**]. Typically, several different models of the same system are combined across multiple levels of abstraction resulting in the implementation of that system. In MDD, both the design and development of new software systems is done by having multiple levels of abstraction, where each level deals only with a particular aspect of the system (therefore decreasing its complexity), and assuring the consistency between them (e.g., translations, synchronisations, etc.). In practice, each level of abstraction can be formalised by means of a domain specific modelling language (DSML), and materialised by its respective supporting tools (i.e., editors, simulators, interpreters, analysers and compilers).

Model-Driven Development (MDD) has already been applied to the development of web applications (e.g.,[**Fraternali:2000tm**] and [**Tai:2004tu**]), real-time systems (e.g., [**Burmester:2005wz**]), role-playing games (e.g., [**Marques:2012um**]), and many more domains but its wide adoption depends on how easily transformations between models can be specified and how fast those transformations can be executed. Notice that the execution of model transformations is typically slow.

1.1 Problem Statement

In order to effectively enable MDD, the consistency between models has to be ensured by model transformations. These transformations have to be easy to specify, maintain and quick to execute. However, as we will see in the following chapters, designing a model transformation language that satisfies those three requirements is really difficult.

There are model transformation languages that are quick to execute, for instance, any general purpose programming language equipped with a proper library such as Java+EMF [**Gronback2009**]; and languages that promote productivity and maintainability, such as AGG [**Taentzer2004**], Atom3 [**Lara2004**] or Viatra2 [**Varro2007a**].

The former languages are typically imperative in the sense that the user of the language specifies how the transformation is supposed to execute. The latter ones are declarative, where the user specifies a set of rules that relate the input models to the output models. The model transformation engine handles the other details. The rules are comprised of a left hand side graph-like pattern and right hand side graph-like pattern. During the transformation execution, the engine must search the input model for occurrences of the left hand side pattern and, when a match is found, an instance of the right hand side pattern is produced in the output model. Due to the graph-like representation of models, searching for occurrences of some graph-like pattern is known to be an NP-Complete problem [**Zundorf:1996vd**].

This shows that there is a price to pay for increased abstraction in model transformation languages. Taking that in consideration, our research question can be stated as:

How can we avoid the abstraction penalty in model transformation languages?

1.2 Expected Contributions

In this thesis we explore how to mitigate the abstraction penalty in model transformation languages. We propose a framework, comprised of a language and its respective environment, designed to tackle the most performance critical operation of high-level model transformation languages: the pattern matching.

The language proposed, instead of following a rule based approach, like most of the state of the art, represents the transformation as a network with explicit structures to control the flow of information from the input model to the output model. This explicit representation of the transformation allows for the application of several analysis and optimizations.

Upon successful completion, we will show that it is possible to mitigate the abstraction penalty in high-level model transformation languages.

1.3 Document Structure

This thesis' structure reflects the main development steps of our solution. Each chapter is a step in achieving that solution.

The first four chapters represent our state of the art investigation. Then, Chapter 5 clarifies the problem we are trying to solve. Chapter 6 plans and explores our approach to solve that problem and Chapters 7, 8 and 9 implements the solution to the problem. Finally, Chapter 10 evaluates the implemented solution and Chapter 11 presents our conclusions.

2

Model Driven Development

This chapter is intended to make the thesis as self-contained as possible. We present all the necessary concepts to understand the context of this thesis. We start by conveying the definition of the word model and introducing the example that will be used throughout the chapter. We explain how models can be used to describe complex systems and the role of model transformations in that process. We conclude with an overview of the most used pattern matching and optimization techniques.

2.1 Models

It is by no means reasonable to build a map the same size as the city it represents. Abstraction, along with problem decomposition and separation of concerns is one of the best ways to deal with complexity [**Schmidt:2006tg**; **Kurtev:2006uy**]. A map of a city is an abstraction of that area that addresses a particular concern. There are several types of maps: road maps (the most widely used), economical maps, political maps, etc... A map is a model.

Given a system M_0, a model M_1 is (i) a representation based on the system M_0 (ii) reflecting only the relevant parts of M_0 (iii) to serve a given purpose [**Kuhne:2006dw**].

In the context of software engineering, models are often represented as graphs[1] because graphs can express complex structures and relations; they are intuitive, expressive and suited for automatic manipulation [**Ehrig:2006tp**]. We will be using the terms graphs and models interchangeably. UML Class Diagrams are widely used to create models of the static information in a software system.

[1] More precisely, models are often represented as typed and attributed graphs. For more details please refer to [**Ehrig:2006tp**].

Using models like the one shown in Figure 2.1, social scientists, without any programming expertise, are able to deploy full featured web applications to conduct their studies. In order to accomplish this, these scientists use applications such as *eSurveysPro*[2] or *LimeService*[3] to read and translate questionnaire models into web applications [**Azanza:2010wq**].

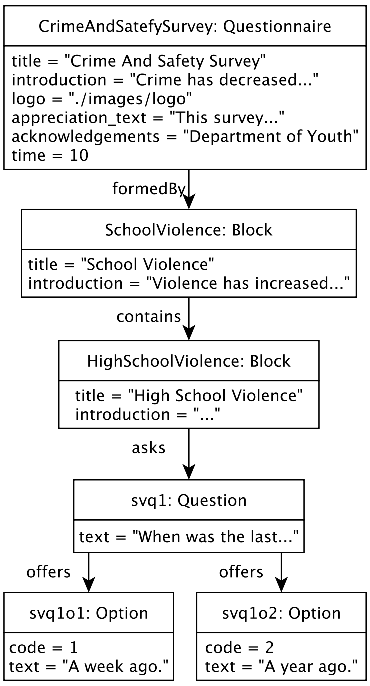

CrimeAndSatefySurvey: Questionnaire
title = "Crime And Safety Survey"
introduction = "Crime has decreased..."
logo = "./images/logo"
appreciation_text = "This survey..."
acknowledgements = "Department of Youth"
time = 10
formedBy
SchoolViolence: Block
title = "School Violence"
introduction = "Violence has increased..."
contains
HighSchoolViolence: Block
title = "High School Violence"
introduction = "..."
asks
svq1: Question
text = "When was the last..."
offers
offers
svq1o1: Option
code = 1
text = "A week ago."
svq1o2: Option
code = 2
text = "A year ago."

Figure 2.1: Questionnaire model. Taken from [**Azanza:2010wq**].

In technical terms, one of the ways to serialize a model is through the XML Metadata Interchange (XMI) format [**OMG2007**]. XMI is a standard XML format created to enable both human and machine readable representation of models.

How does one guarantee that social scientists will always build syntactically correct questionnaire models? By syntactically correct we mean that they are suited for being parsed by some application. Note that a successfully parsed questionnaire does not have to make any sense. We will discuss syntax and semantics in the next section.

2.2 Syntax and Semantics

In order to better understand syntax and semantics consider the following example:

> The expression "$\frac{5}{0}$" is syntactically correct with respect to integer arithmetic because "5" and "0" are valid integers and the division is a binary operator; but semantically incorrect since "$\frac{5}{0}$" does not mean anything or has

[2] http://www.esurveyspro.com

[3] http://www.limeservice.org

> the wrong meaning. The expression "$5 * 3$" is syntactically and semantically correct: it means "15". The expression "$5+$" is syntactically incorrect because the plus operator needs two operands.

A syntactically correct model has to respect some well-formedness rules. A semantically correct model can be interpreted to produce something with meaning.

Since models are to be automatically parsed and interpreted, there must be a way to unambiguously specify well-formedness rules. That is were a metamodel comes in.

2.3 Metamodels

A metamodel is a set of rules that describes all possible syntactically correct models, i. e., given a model M_1 and a metamodel M_2 of M_1, it is easy to check if M_1 is syntactically correct with respect to M_2 [**Kurtev:2006uy**]. When this happens, we say that M_1 *conforms to* M_2. Moreover, given M_2, a computer is able to parse any M_1, that conforms to M_2, and further do something useful with that information. This is analogue to what happens between a program and its programming language. The program has to be parsed by some compiler (or interpreter), hence the need for a set of rules dictating syntactically correct programs.

Figure 2.2 shows a simplified metamodel for questionnaire models. This metamodel can be used by applications that parse questionnaire models to check if they are syntactically correct. As the questionnaire metamodel shows, a questionnaire is formed by a forest of blocks. Each block contains a set of questions with zero or more options. Questions with zero options expect any textual answer. A questionnaire has attributes such as title, introduction, logo, appreciation text, acknowledgements and an expected time to complete. The information about the each particular metamodel element lies in its attributes. Comparing the model shown in Figure 2.1 and the metamodel of Figure 2.2 one is able to determine to determine the correspondence between each model element and metamodel element. For instance, the *SchoolViolence* element corresponds to a *Block*. This relation between a model element and its corresponding metamodel elements is the *instance of* relation. In our example, the *SchoolViolence* is an instance of the *Block* element.

Formally, a model M_1 conforms to a metamodel M_2 iif it is possible to build a $instanceof$ relation between the set of nodes and associations N_1 and A_1 of model M_1 and the set of nodes and associations N_2 and A_2 of M_2 such that the following conditions hold:

$$\forall_{a \in A_1} instanceof(src(a)) = src(instanceof(a))$$
$$\forall_{a \in A_1} instanceof(trg(a)) = trg(instanceof(a))$$

where $src(a)$ and $trg(a)$ denote the source and target elements of the association a. For more details, Hartmut Ehrig presents a formalization on type and attributed graphs in [**Ehrig:2006tp**].

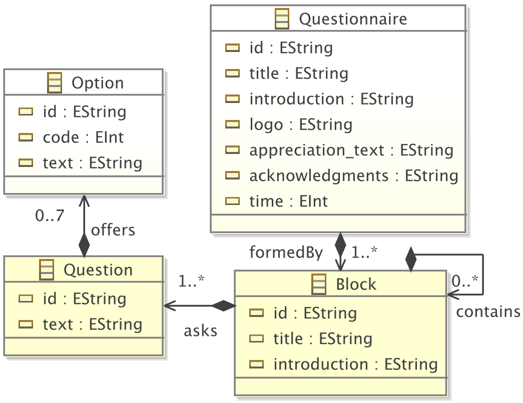


Figure 2.2: Questionnaire metamodel. Taken from [**Azanza:2010wq**].

Figure 2.3 shows the *instanceof* relation between the model of Figure 2.1 and the metamodel of Figure 2.2.

There are tools called modelling environments that offer support to the creation of metamodels and tools to manipulate models. For instance, a user can use a modelling environment to create a questionnaire metamodel like the one shown in figure 2.2 and automatically generate a set of editors that ease the manipulation of questionnaire models. Eclipse Modeling Framework (EMF) [**Gronback2009**] is an example of a modelling environment but there are others such as Generic Modelling Environment (GME) [**Ledeczi:2001va**] or MetaEdit+ [**Tolvanen:2003jj**].

How does the modelling environment guarantees that the created metamodels are syntactically correct? Where are the set of rules that prescribe well formed metamodels?

2.4 Meta-metamodels

A metamodel M_2 conforms to a meta-metamodel M_3. A meta-metamodel represents the set of all well formed metamodels. Note that this makes the metamodel a model M_2 conforming to M_3. Figure 2.4 shows a simplification of a meta-metamodel: the Ecore meta-metamodel. The Ecore is the meta-metamodel used in the Eclipse Modelling Environment (EMF).

The relation between a metamodel M_2 and a meta-metamodel M_3 is the same as the one between a model M_1 and M_2: M_2 is an *instanceof* M_3. The *instanceof* relation is (partially) shown in Figure 2.5.

The meta-metamodel must also obey to some well formedness rules. It turns out that the meta-metamodel is supposed to conform to itself, i.e., it is reflexive. This means we don't need a meta-meta-metamodel. Unfortunately, there are some issues about interpreting a reflexive meta-metamodel from a pragmatic point of view as identified in [**Seidewitz:2003jw**]. Fortunately, in practice this limitation is negligible since most tools operate at the level of metamodels and below [**Favre:2004va**]. Figure 2.6 summarizes the

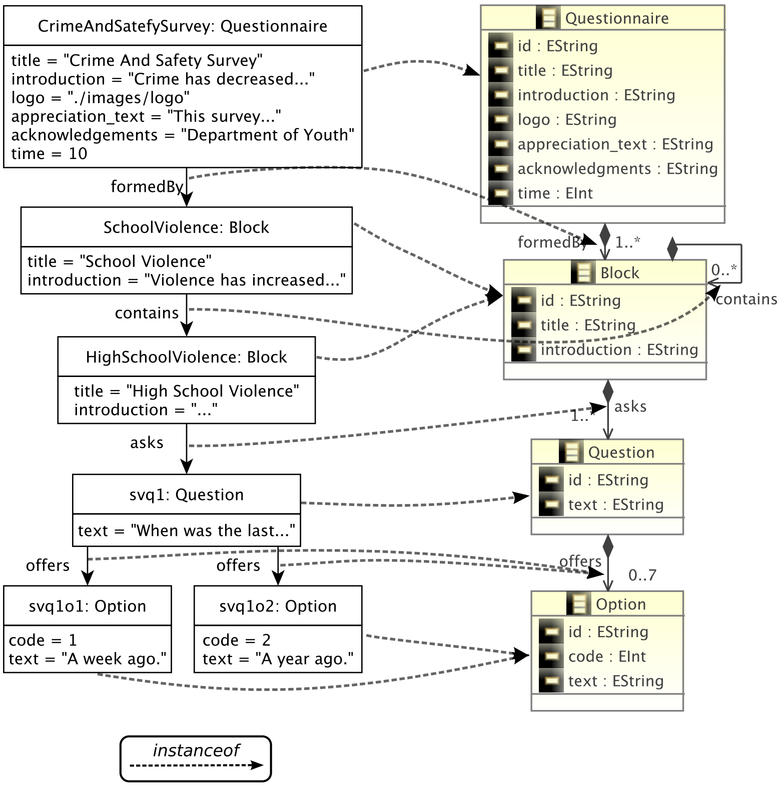


Figure 2.3: Questionnaire model and metamodel. Taken from [**Azanza:2010wq**].

four levels of abstraction described until now. The M_0 denotes the system under study that needs and abstraction; M_1 denotes the model of M_0; M_2 is the metamodel to which M_1 conforms to; and M_3 is the meta-metamodel.

2.5 Model Driven Development

Model-Driven Development (MDD) is a software engineering approach that uses models as first class citizens to create and evolve software systems [**Hailpern:2006vd**]. The software code is generated from a set of models thus enabling massive reuse and "correct-by-construction" software [**Schmidt:2006tg**].

The Model-Driven Architecture (MDA) is a set of guidelines proposed by the Object Management Group[4] (OMG) on how to produce applications using MDD [**Soley2000**]. These guidelines suggest that when building a complex system, Platform Independent Models (PIMs) should be used to describe that system. Then, PIMs are translated to Platform Specific Models (PSMs). Furthermore, MDA proposes a meta-metamodel (M_3) called Meta Object Facility (MOF) and the four abstraction levels shown in Figure 2.6.

Platform Independent Models (PIMs) denote those models that don't keep specific

[4] http://www.omg.org/

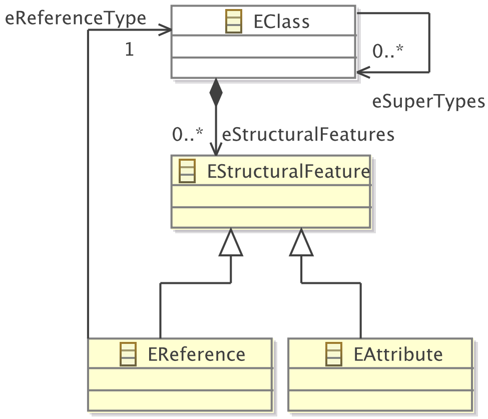


Figure 2.4: (Highly) Simplified Ecore Metamodel.

information about a system's ultimate execution environment (operating system, programming language, etc. . .) [**Atkinson:2005ux**]. The questionnaire model of Figure 2.1 does not keep information about its execution environment, i.e., it is a PIM. Because of this, an application like eSurveyPro can choose to generate a web application or a desktop application, or even a mobile version, to evaluate the survey.

After a total or partial specification of the necessary PIMs to describe a system, a set of Platform Specific Models (PSMs) is generated from those PIMs. PSMs carry information about the system's execution environment and are well suited for being used to generate (automatically or semi-automatically) that system.

An application like eSurveyPro, capable of processing a questionnaire model to produce a web application, would operate in the following way:

1. A questionnaire model is given as input to the application;
2. The application checks if the model conforms to the questionnaire metamodel.
3. Assuming the model is syntactically correct, there are three equivalent alternatives:
   (a) The application interprets the model and presents the user with a web page containing the questions and choices declared in the model.
   (b) The application translates the model to a set of Java classes that, when run, present the user with a web page containing the questions and choices declared in the model.
   (c) the application translates the questionnaire model to a Enterprise JavaBeans (EJB) model[5] and then the EJB model is translated to a set of classes implementing the questionnaire. The classes generated include support for transactions, remote and distributed execution, persistence, etc. . . This EJB-to-Java translation is done by other application (e.g., *AndroMDA*[6] is a tool capable of

[5] http://www.oracle.com/technetwork/java/javaee/ejb/index.html

[6] http://www.andromda.org

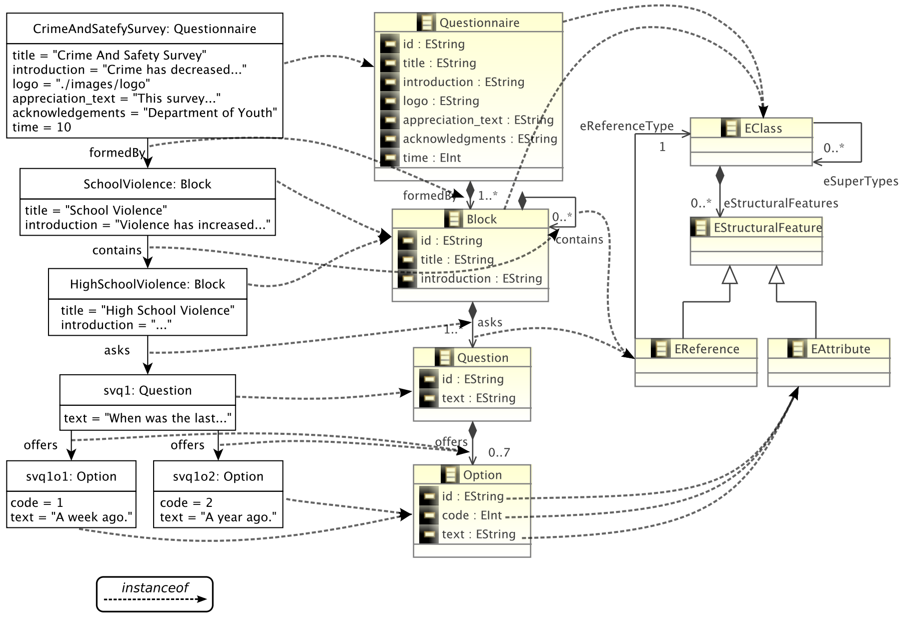


Figure 2.5: (Partial) Relations of conformance between model, metamodel and meta-metamodel.

performing the translation), specifically devised to translate EJB models to executable Java classes.

The alternatives 3a, 3b and 3c have the same purpose: they provide semantics (a meaning) to the questionnaire model. The first alternative gives semantics in an operational manner. The second and third ones do this through a transformation into a model that already has defined semantics (EJB is a model and code can also be seen as a model [**Bezivin:2005wv**]).

The alternative 3c is the *easiest* to realize because the transformation we have to build is one between models conforming to similar metamodels (Questionnaire and EJB) as to building an interpreter or a compiler (options 3a and 3b). Option 3c is also the *safest* alternative because the probability of introducing bug in the transformation process is far smaller than in the other options. Figure 2.7 shows a possible (simplified) EJB model produced with such a transformation applied to the questionnaire model of Figure 2.1. By observing the EJB model, we can immediately come up with a set of heuristics on how to perform the transformation: 1. Each questionnaire item is translated into a *Session Bean* and a persisted *Entity Bean*. The *Session Bean* objects provide the necessary controls to allow the user to answer the questionnaire. 2. Blocks, Questions and Options are all translated into respective *Entities* as they need to be persisted after each session.

The keen reader will observe that the heuristics presented are applicable to any possible questionnaire model, not just the one shown in Figure 2.1.

here are at least two possible ways to implement the translation between questionnaire models and EJB models. One is to manually code a visitor that parses the XMI file

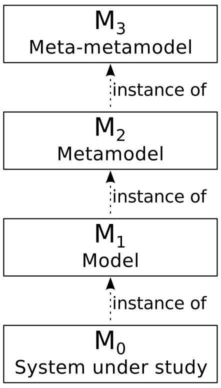


Figure 2.6: System, Model, metamodel and meta-metamodel.

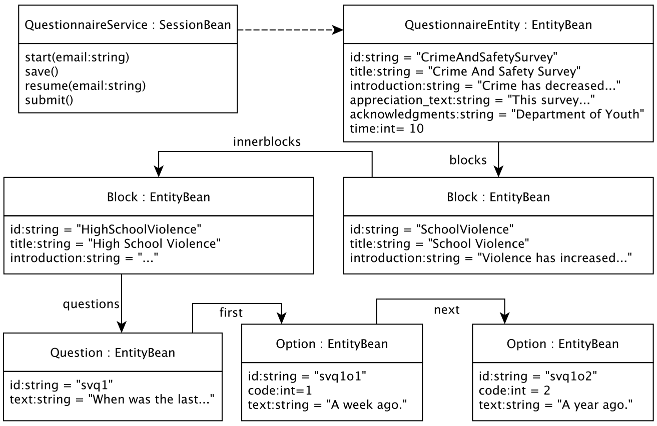

QuestionnaireService : SessionBean
start(email:string)
save()
resume(email:string)
submit()
QuestionnaireEntity : EntityBean
id:string = "CrimeAndSafetySurvey"
title:string = "Crime And Safety Survey"
introduction:string = "Crime has decreased..."
appreciation_text:string = "This survey..."
acknowledgments:string = "Department of Youth"
time:int= 10
innerblocks
blocks
Block : EntityBean
id:string = "HighSchoolViolence"
title:string = "High School Violence"
introduction:string = "..."
Block : EntityBean
id:string = "SchoolViolence"
title:string = "School Violence"
introduction:string = "Violence has increased..."
questions
first
next
Question : EntityBean
id:string = "svq1"
text:string = "When was the last..."
Option : EntityBean
id:string = "svq1o1"
code:int=1
text:string = "A week ago."
Option : EntityBean
id:string = "svq1o2"
code:int = 2
text:string = "A year ago."

Figure 2.7: Simplified EJB model generated from a questionnaire model.

of a questionnaire model and generates a XMI file representing the EJB model. This is the visitor based approach [**Jilani:2010wr**]. It is also possible to use templates to orchestrate the generation of the EJB model contents, i.e., following a template based approach. However, these two alternative are more suited to perform model-to-code transformations as they allow for easy code generation. In this case we are interested in performing a model-to-model transformation. Model Transformations are the subject of the next chapter.

3

Model Transformations

A model transformation is "the automatic generation of a target model from a source model, according to a transformation definition" [**Kleppe:2003:MEM:829557**].

The transformation definition is expressed in some language. The language can be a general purpose language like Java or a more specific model transformation language.

Model transformation languages, together with their supporting model transformation tools provide an high-level and highly productive environment where transformation specifications consist of rules like the one shown in Figure 3.1.

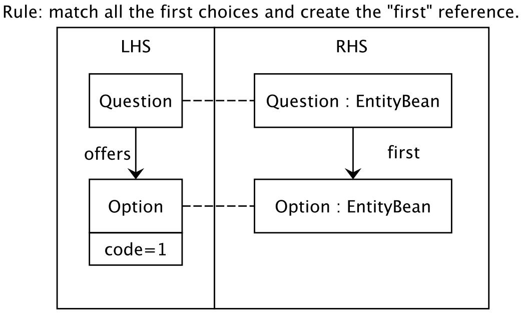


Figure 3.1: Example rule used to specify a model transformation.

3.1 Environment

Model transformation tools interpret a transformation specification the relates some input model, conforming to metamodel, to some output model, conforming to some other metamodel. Typically, even the transformation itself is a model conforming to a metamodel [**Bezivin:2004us**] as is illustrated in Figure 3.2.

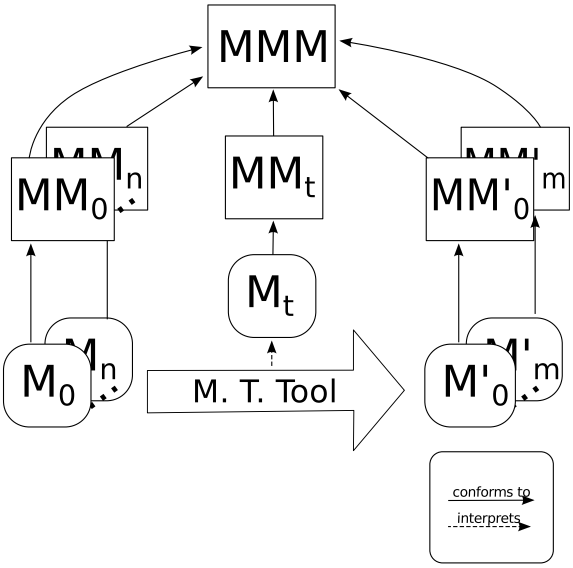


Figure 3.2: Overview of the model transformation process.

The MMM denotes the meta-metamodel (e.g. Ecore or MOF) and the MM$_t$ is the metamodel of the transformation specification. This metamodel, MM$_t$, is often called the transformation language. Note that if $n = m = 0$ and MM$_0$ = MM$'_0$ we have a transformation between models conforming to the same metamodel, i. e., an endogenous transformation [**Meszaros:2006ds**]. When the input and output metamodels are different we have an exogenous transformation [**Meszaros:2006ds**].

There is a great variety of MTTs, each unique in features provided, language used, and approach followed to solve the model transformation problem [**Grunske:2005uv**]. Some operate with just a set of transformation rules applying them in any order (declarative approaches); others allow the user to control rule scheduling (or rule selection) (hybrid approaches); others take this further by providing an imperative language with loop constructs, branching instructions, composition mechanisms, etc... that allow the user to program all the transformation process (imperative approaches). Czarnecki and Helsen provided a classification of model transformation approaches in [**Czarnecki2003**] that captures most of the MTTs' features from a usability point of view.

3.2 DSLTrans - A Model Transformation Language

DSLTrans [**Barroca:2011wr**] is a visual rule-based language for model transformations. It uses layers to define an order to apply the rules to the input model. Figure 3.3 shows an excerpt of a transformation that translates questionnaire models into EJB models as is described in section 2.5.

A transformation in DSLTrans is formed by a set of input model sources called file-ports (“inputQuestionnaire” in Figure 3.3) and a list of layers. Input model sources are typed by the input metamodel and layers are typed by the output metamodel. Each layer is a set of rules that are executed in a non deterministic fashion. The top part of each rule is called the “match” and the bottom is “apply”.

In the example presented in Figure 3.3, in the first layer, left rule, for each Question instance found in the input model, a new EntityBean instance is created in the output model, with the name “Question” and id equal to the id of the found Question instance. In addition, a trace link is created internally identified by the “Q2E_Trace” string. These trace links can be used in the subsequent layers to retrieve a Question instance and the corresponding EntityBean instance created in the rule. In the right rule, a similar operation is performed to all the Option instances found in the input metamodel. In the rule of the second layer, all the question and respective offered questions are being matched in the input model, together with the corresponding previously created EntityBeans and a new association called “first" is being created.

For more details about DSLTrans, please refer to section 6.2 in chapter 6.

3.3 State of Art

Off course DSLTrans is not the only model transformation language. There are many others, each with it’s particular set of features, advantages and limitations. In our study, we tried to cover as much languages as possible, namely: (i) imperative tools such as ATC [**estevez2006atc**] and T-Core [**Syriani:2010tq**]; (ii) declarative tools such as AGG [**Taentzer2004**], Atom3 [**Lara2004**] and Epsilon Flock [**Rose:2010:MME:1875847.1875862**]; (iii) programmed graph rewriting approaches such as GReAT [**Balasubramanian:2007td**], GrGen.NET [**Kroll2007**], PROGReS [**Schurr1994**], VMTS [**Levendovszky2005**] and MoTif [**Syriani:2011kf**]; (iv) incremental approaches such as Beanbag [**Xiong2009**], Viatra2 [**Varro2007a**] and Tefkat [**Lawley2006**]; (v) and bidirectional approaches such as BOTL [**Braun2003**].

3.4 The Model Transformation Process

Based on our study of the state of the art tools, we built a general process that identifies the main stages occurring in most model transformation executions.

Figure 3.4 identifies the main stages in two typical transformation execution modes: interpretation (left) and compilation (right). The only difference between these two modes is that, in the compilation, the execution of the transformation is separated from the transformation load, parse and compile tasks and off course, the performance, as we will see in chapter 10.

We stress the fact that the presented diagrams are not supposed to describe exactly how model transformation tools operate, but to provide instead a clear overview of the main stages in most model transformation executions. However, these diagrams are general enough to describe even imperative tools (e.g., ATC [**estevez2006atc**]), where most of the transformation execution stages (in the Execute Transformation State) are manually coded by the transformation programmer. We also assume that a transformation is comprised of a set of *rules*, each containing an Left-Hand Side (LHS) pattern, that needs to be found in the input model, and a Right-Hand Side (RHS) pattern which represents the output model. There is no loss of generality, since these *rules* (with the mentioned patterns) do not need to be explicitly represented in the transformation language. They can be implicit in the transformation programmer's mind when coding the transformation. For simplicity's sake, we only consider one input model and one output model but it is easy to see how the process can be adapted to multiple input/output models.

As is illustrated in Figure 3.4, an engine always starts by loading the transformation and, in the case of interpretation, the input model. At this point, some existing engines perform global optimizations, which will be explained and categorized in section 4.2. Next, the engine executes the transformation by selecting each rule and optionally performing some local optimizations (see section 4.2). After those optimizations, a search in the input model must be performed in order to find where to apply the rule. In this task, the engine has to find occurrences of the rule's left-hand side pattern in the input model. From a performance point of view, this operation is the most expensive, and usually all the optimizations target the reduction of its cost (see chapter 4). The application of the rule's right hand side is performed for each occurrence found and, if there are more rules to be applied, the transformation continues. Else, the transformation execution ends. After that, the output model is stored.

3.5 Performance

In order to better understand how much the pattern matching operation (the *Match LHS* State in Figure 3.4) costs, we picked a benchmark created for the Tool Transformation Contest (TTC) 2010. For all details about the benchmark case study, input models and tools used, please refer to chapter 10.

From the available submissions we selected transformations expressed in Epsilon Flock, ATL and GrGen.NET tools. We also coded a transformation entirely in Java, using the Eclipse Modelling Framework (EMF) library to load and store the models.

Running each transformation with increasingly larger input models yielded the results shown in Figure 3.5. Note the logarithmic scale in both axes. The vertical axis denote the total transformation time, i.e., the input model load, transformation parse, execution and output model storage tasks. The horizontal axis denotes the input model size.

It is clear that, in terms of performance, there is a big gap between model transformation languages and a general purpose programming language like Java. In a scenario where performance is very important, Java, or any other low-level general purpose language, might be the only tool for the job. Off course, creating and maintaining a transformation entirely written is Java is, at least compared with model transformation languages, a tiring, unproductive and error-prone task. Not to mention that the code is tightly coupled to the EMF library to load and store models.

The three tools used have a very intuitive, compact and declarative syntax to specify the transformation. Compared to Java, it is really easy and quick to create the transformation in all three languages. Also it should be straightforward to cope with any change to the representation of models. The problem is that at industrial scales, any transformation written in these languages becomes useless.

Only after high-level transformations are shown to run fast enough, people will use them at industrial scales. Optimization techniques play an important role for that purpose.

In the next chapter, we will see why the pattern matching process is the most critical in terms of performance and a categorization of the state of the art optimization techniques to mitigate this problem.

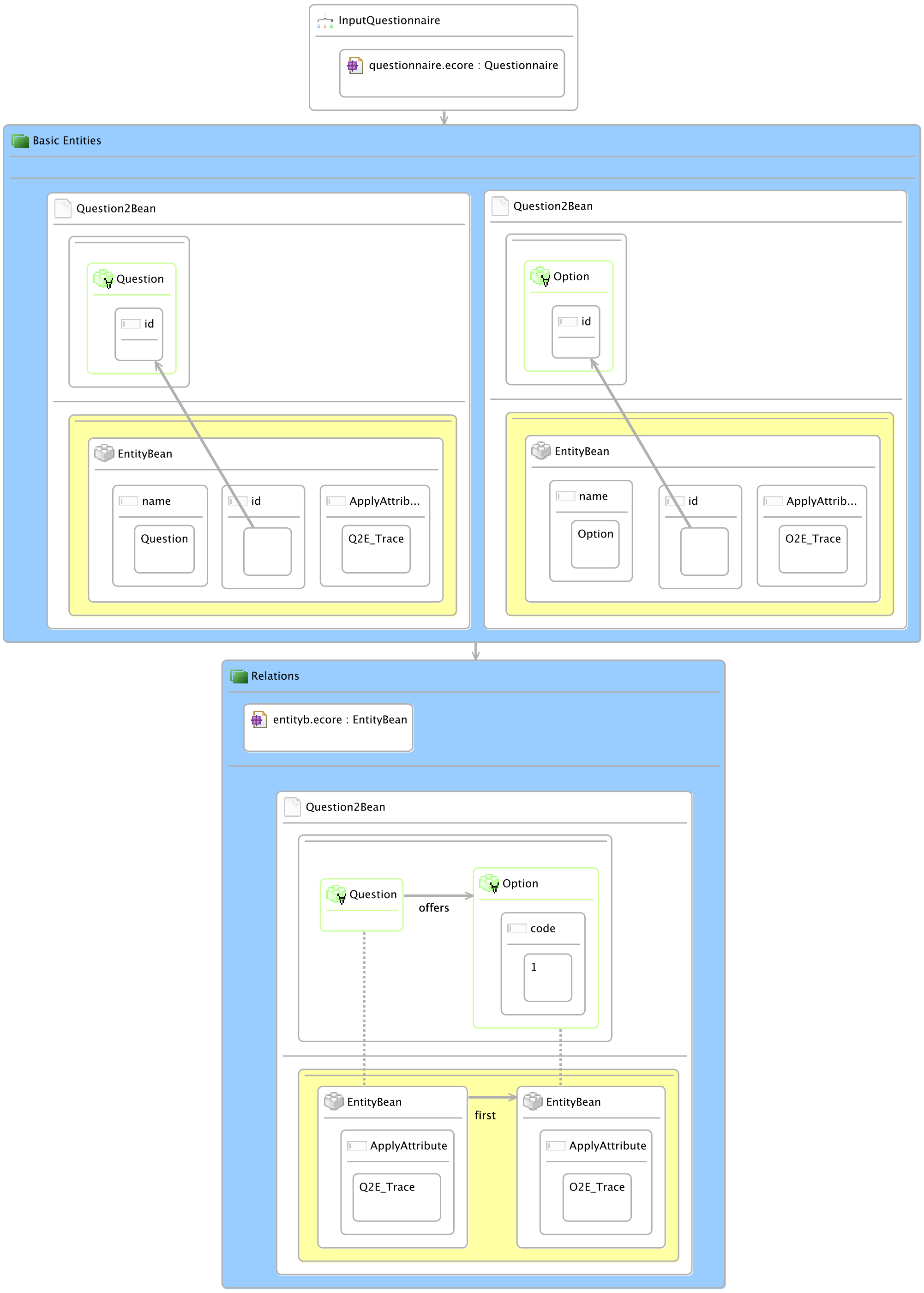
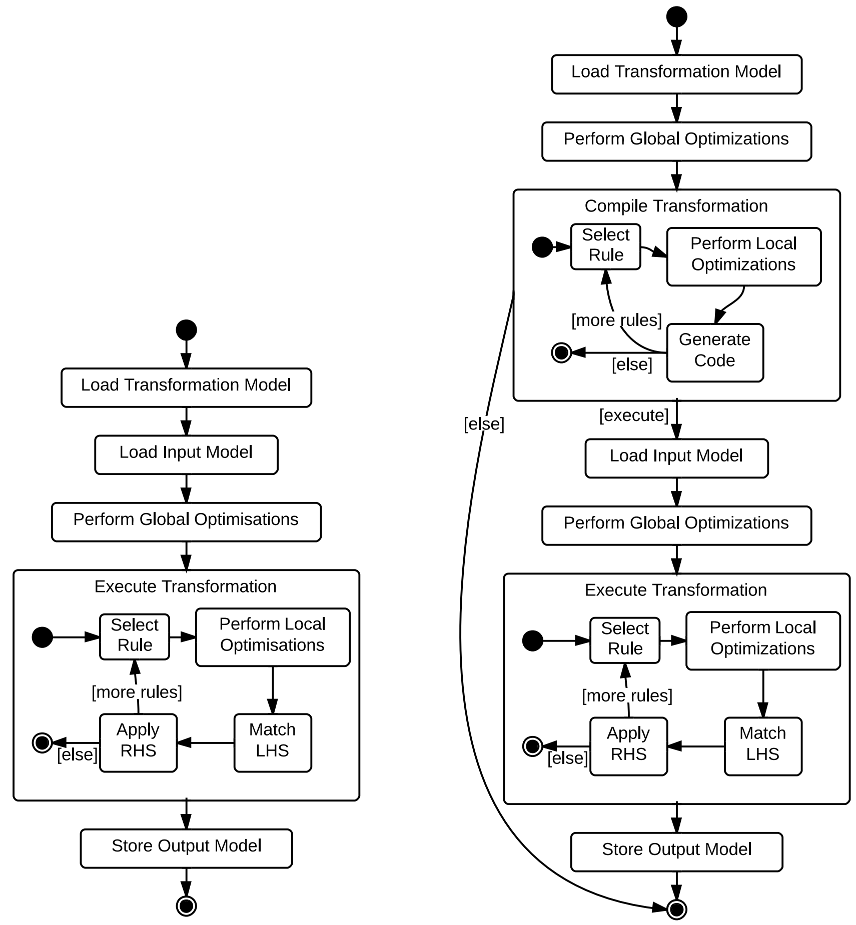


Figure 3.3: Excerpt of a transformation expressed in DSLTrans.



(a) Interpretation process of a transformation model overview.

(b) Compilation process of a transformation model overview.

Figure 3.4: Overview of the execution process of a transformation model.

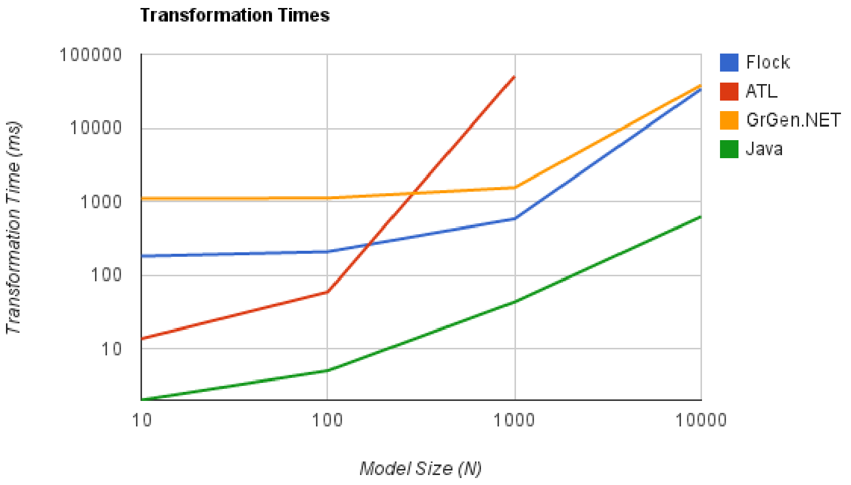


Figure 3.5: Transformation running times of some state of art tools.

4

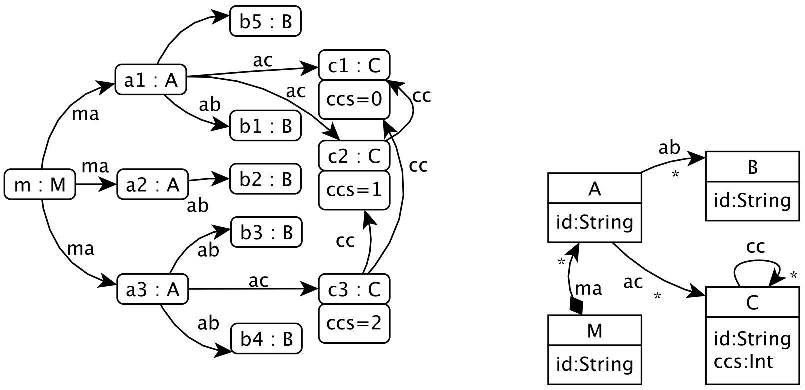
Pattern Matching Optimization Techniques

In this chapter, we study the pattern matching process. Optimizing this NP-Complete [**Zundorf:1996vd**] problem is one of the most effective ways to reduce execution times and achieve industrial applicability for model transformations. We identify and classify several pattern matching optimization techniques and we present the categorization of the state of the art tools.

In the following sections we provide a simple and general explanation for each optimization technique with the help of some examples. The input model used, and corresponding metamodel, are presented in Figure 4.1. The metamodel states that instances of *M* are comprised of zero or more instances of *A*. *A* elements refer to zero or more elements of type *B* or *C*. *C* elements may reference multiple *C* instances. All elements have an *id* string attribute and *C* elements have an extra *ccs* integer attribute. Note that containment associations (with the black diamond in the source) state that a child element (association target) may only exist inside one parent element (association source). Later we will see why this fact has an impact on the performance of the pattern matching operation.

4.1 The Pattern Matching Problem

Consider the pattern shown in Figure 4.2. The search for occurrences of that pattern in some model is called the pattern matching. More specifically, for an occurrence to be found, each element in the pattern, including the associations, has to be mapped to an element of the input model. Figure 4.3 shows one such mapping for the input model of figure 4.1a but there can be thousands of possible matches in large models. In this



(a) Model used as input for the illustration of pattern matching techniques.

(b) Metamodel.

Figure 4.1: Sample input model (left) and corresponding metamodel (right).

example, there are two: the first one is shown in the figure and the second one has the mapping $\{(x, a3), (z, c3), (y, b3)\}$. Had we omitted the restriction *ccs=2* and there would be 6 possible mappings.

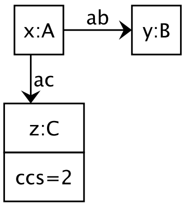


Figure 4.2: Sample pattern.

In order to find such mapping in any given input model, a transformation engine has to follow an algorithm similar to Algorithm 1. A few remarks about the notation: 1. The *GetAllInstances(Type)* function returns the set of all elements from the input model that are instances of *Type*; 2. *element.association* returns, all the model elements that are connected to *element* by the association *association* as targets; 3. *element.attribute* returns the value of the attribute *attribute* of the element *element*; 4. It should be clear by the context whether we are navigating an association or accessing an attribute with these two last operations; 5. The *BAC* unique identifies the pattern in Figure 4.2;

The algorithm starts by finding all the instances of *A* and then, for each instance *a*, navigates along the edges *a.ab* and *a.ac* to find the remaining elements.

For the worst case scenario, assume that *GetAllInstances(T)* has to search through the entire model to find all elements of type *T* and assume that *A* elements are connected to every *B* and every *C* elements in the model. The time complexity of the Algorithm 1, in this scenario, is $O(((|M|+|A|+|B|+|C|)+(|A|\times|B|\times|C|))$ where $|T|$ denotes the size of the *GetAllInstances(T)* set. Note that this is just the cost of one particular pattern matching

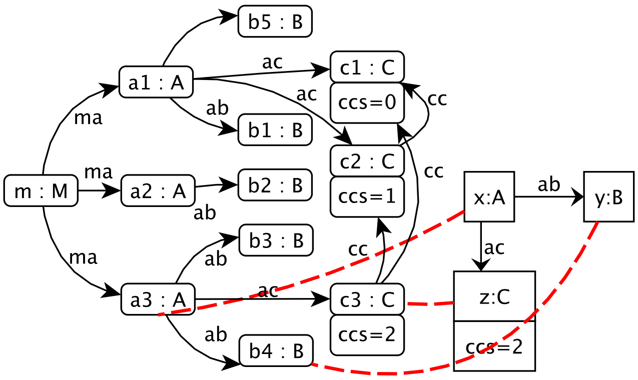


Figure 4.3: Example occurrence of the Left-Hand-Side (LHS) pattern of the rule in Figure 3.1.

Algorithm 1 An algorithm to find occurrences of the pattern shown in Figure 4.2.

```
function FindAllOccurrencesBAC
    $occurrences \leftarrow \{\}$
    for $a \in GetAllInstances(A)$ do
        for $b \in a.ab$ do
            for $c \in a.ac$ do
                if $c.ccs = 2$ then
                    $occurrences \leftarrow occurrences \cup \{(x, a), (y, b), (z, c)\}$
                end if
            end for
        end for
    end for
    return $occurrences$
end function
```

operation. In a whole transformation execution, several patterns need to be matched.

Most model transformation tools, when loading the input model (see the process in figure 3.4), create an index to store elements organized by their type. Using this feature the tool does not have to search the whole input model to find an element of a given type[1]. This causes the *GetAllInstances(T)* function to return immediately with the set of all instances of T. So, the worst case, for most model transformation tools, of Algorithm 1 is $O(|A| \times |B| \times |C|)$.

In general, given an arbitrary pattern with $\{n_1, n_2, \ldots, n_m\}$ matching elements, each being instance of types $\{T_1, T_2, \ldots, T_m\}$, respectively, an algorithm to match those elements has a worst case time complexity of $O(|T_1| \times |T_2| \times \ldots \times |T_m|)$. Note that if $T_i = T_j$ for all i and j, we have $O(T^m)$ meaning that the algorithm is exponential in the size of the pattern.

[1] Recall that most pattern elements are syntactically typed

4.2 Optimization Techniques

The good news is that, in the model transformation context, models are typically sparsely connected, model elements are indexed by their type and patterns in rules are generally small. Because of these facts, the average pattern matching process has a complexity that can be "(...) overapproximated by a linear or quadratic function of the model size" [**Varro2008**]. Also, there is lots of room for improvement in Algorithm 1.

4.2.1 Indexing Techniques

For instance, the tool could be improved to, when loading the input model, create an index with the inverse associations of the existing associations. This feature is so useful that it is usually supported by the model management frameworks used by tools, such as UDM [**Magyari:2003vy**] or EMF [**Gronback2009**]. It also means that, for each instance of an association $T_1 \xrightarrow{assoc} T_2$, there would be an instance of the association $T_2 \xrightarrow{assocInv} T_1$. The implications are clear: If there are only associations $T_1 \xrightarrow{assoc} T_2$ in the model and we are given an element t_2 instance of T_2, it is still possible to obtain all the instances of T_1 that are connected to t_2 without having to search all instances of T_1 in the model. We do that with $t_2.assocInv$. If an engine supports this indexing technique, then the Algorithm 1 could be improved to Algorithm 2.

Algorithm 2 An algorithm to find occurrences of the pattern shown in Figure 4.2 taking advantage of inverse associations.

```
function FINDALLOCCURRENCESBAC
    occurrences ← {}
    for c ∈ GetAllInstances(C) do
        if c.ccs = 2 then
            for a ∈ c.acInv do
                for b ∈ a.ab do
                    occurrences ← occurrences ∪ {(x, a), (y, b), (z, c)}
                end for
            end for
        end if
    end for
    return occurrences
end function
```

In Algorithm 2 we switched the order of the loops because, since we can only have an occurrence if the z element satisfies the restriction *z.ccs* = 2, we might as well start by looking for such z elements. Notice that we can only take advantage of this heuristic if the engine has built inverse relations.

In the worst case, all the z elements satisfy the *z.ccs* = 2 restriction the complexity remains at $O(|C| \times |A| \times |B|)$. But, in the average case, it will be $O((|C| \times P(z.ccs = 2)) \times |A| \times |B|)$ where $P(z.ccs = 2)$ is the probability of picking one instance z of C

that satisfies the restriction *z.ccs* = 2. In Chapter 8 we will show how this probability can be estimated. For now, if the input model shown in figure 4.1a is representative, then $P(z.ccs = 2) = \frac{1}{3}$.

We can improve the indexing techniques of the tool to provide attribute indexes. This way, if an element T with an attribute $T.attr$ of type P is frequently accessed throughout a transformation, then building an index $T_{attr} : P \rightarrow T$ can yield a major speed up. Algorithm 3 shows how this index can be used to fetch immediately all the instances c of C satisfying the restriction $c.ccs = 2$. Note that since the index fetches the correct instances there is no need for the verification of the restriction. Normally, the creation of these indexes is controlled by the user so that only the most relevant attributes are indexed but the tool can analyse all the transformation and determine which attributes deserve to be indexed. In order to fill the index with the necessary instances, the tool, when loading the input model, scans all C instances and organize them in the index.

Algorithm 3 An algorithm to find occurrences of the pattern shown in Figure 4.2 taking advantage of attribute indexes.

```
function FINDALLOCCURRENCESBAC
    occurrences ← {}
    for $c \in C_{ccs}(2)$ do
        for $a \in c.acInv$ do
            for $b \in a.ab$ do
                occurrences ← occurrences ∪ $\{(x, a), (y, b), (z, c)\}$
            end for
        end for
    end for
    return occurrences
end function
```

We can take these indexing techniques one step further and implement structural indexes. These allow for the storage of instances of whole patterns. For example, assume that the pattern $B \xleftarrow{ab} A \xrightarrow{ac} C$, identified by *BAC′* is frequently matched throughout the transformation. By detecting this fact, the engine creates an index $I_{BAC'} : B \times A \times C$ that, when accessed, returns all the instances of that pattern. Algorithm 4 shows an example match operation that uses the structural index that we gave as example. Similarly to the previous indexing techniques, the engine has to fill the index after loading the model, and particularly for this indexing technique, this operation can be costly. But the average cost of Algorithm 4 lowers to $O(|I_{BAC'}| \times P(z.ccs = 2))$. Normally, the creation of structural indexes is controlled by the user (as in PROGRES [**Zundorf:1996vd**; **Schurr1994**]) but there are tools that create indexes for all patterns in the transformation (such as Viatra2 [**Bergmann2008**; **Varro:2006vo**]). These tools keep the transformation running and update the indexes whenever the input model changes. In this way, retrieving the output model is extremely fast and there is no need to perform pattern matching for all rules when a minor change occurs in the input model [**Varro2006**]. This technique is called incremental pattern matching.

Algorithm 4 An algorithm to find occurrences of the pattern shown in Figure 4.2 taking advantage of structural indexes.

```
function FINDALLOCCURRENCESBAC
    occurrences ← {}
    for (b,a,c) ∈ $I_{BAC'}$ do
        if c.ccs = 2 then
            occurrences ← occurrences ∪ $\{(x,a),(y,b),(z,c)\}$
        end if
    end for
    return occurrences
end function
```

Up until know we have seen three indexing techniques: Type, Attribute and Structural. These techniques are applied automatically by the engine, all the necessary structures are initialized in the “Perform Global Optimizations” stage of the transformation process of Figure 3.4, and they impact more than one pattern matching operation. That is why we call them global optimizations as opposed to local ones, in which the impact is on the current pattern matching operation.

4.2.2 Caching

Yet another global optimization technique, similar to indexing, is to cache pattern matching operations. This can be performed automatically by detecting which match operations' results can be reused in other operations. The *Cache* act as a map that returns all instances of a given pattern identifier. For instance, consider Algorithm 5 that implements the look up for the pattern of Figure 4.4. Note how the cache is being accessed and updated. If all the pattern matching algorithms used in the transformation behave the same way with respect to the cache, then the matching of the pattern of Figure 4.4 could be matched by Algorithm 6. If the engine matches the pattern of Figure 4.4 before matching the one of Figure 4.2, then Algorithm 6 will always hit the cache and only iterates the C instances. Note that *AB* is a unique identifier for the pattern in Figure 4.4 and that *BAC* is a unique identifier for the pattern in Figure 4.2.

Caching techniques serve not only the purpose of storing patterns. For instance, they can be used to store derived attributes[2] as is done in ATL [**Jouault2008**]. Derived attributes can be computed and stored when loading the input model.

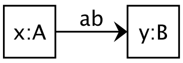


Figure 4.4: Sample pattern.

[2] Derived attributes are attributes that are computed when they are accessed.

Algorithm 5 An algorithm to find occurrences of the pattern shown in Figure 4.4 using cache.

```
function FindAllOccurrencesAB
    if Cache[AB] $= \varnothing$ then
        occurrences $\leftarrow \{\}$
        for $a \in GetAllInstances(A)$ do
            for $b \in a.ab$ do
                occurrences $\leftarrow$ occurrences $\cup \{(x, a), (y, b)\}$
            end for
        end for
        Cache[AB] $\leftarrow$ occurrences
    end if
    return Cache[AB]
end function
```

Algorithm 6 An algorithm to find occurrences of the pattern shown in Figure 4.2 using cache.

```
function FindAllOccurrencesBAC
    if Cache[BAC] $= \varnothing$ then
        occurrences $\leftarrow \{\}$
        for $\{(x, a), (y, b)\} \in$ Cache[AB] do
            for $c \in a.ac$ do
                occurrences $\leftarrow$ occurrences $\cup \{(x, a), (y, b), (z, c)\}$
            end for
        end for
        Cache[BAC] $\leftarrow$ occurrences
    end if
    return Cache[BAC]
end function
```

4.2.3 Search Plan Optimization

Algorithm 2 presents a different loop order to take advantage of the *z.ccs=2* restriction in pattern of Figure 4.2. The argument is that the average cost of the whole pattern matching operation is less than the cost of Algorithm 1. This is true but for an engine to reason about such things there must be a notion of plan and cost.

Generically, a search plan [**Varro2008**; **Zundorf:1996vd**] for a pattern is a representation of the algorithm to be performed to find occurrences for that pattern. Changes to the search plan imply changes to the algorithm. A search plan with a lower cost, implies that the algorithm, in the average case, should have a lower cost. Search plan optimizations are local because, each time they are applied, they attempt to reduce the cost of each individual pattern matching optimization.

Each model transformation tool that implements search plan optimizations has its own representation of a plan, and cost. But in the tools that we studied we were able to identify three kinds of cost models, which we named according to the kind of information they require. There are cost models that take into account a sample of representative input models, the current (under transformation) input model, the input metamodel structure and even the cost of the underlying structures such as index look ups, disc access, etc.

A cost model that only requires information about the metamodel is called metamodel sensitive. It employs a set of heuristics that take into account the kind of restrictions in the pattern, the type of associations between metamodel elements, etc... We have already presented an example of one of these heuristics, which is also the most used one, the *first-fail* principle: a good search plan should start the search in the most restricted pattern element since it will have the fewest possible occurrences. The Algorithm 2 starts by iterating all the C instances because of the attribute constrain. Algorithm 1 may have to iterate several A and B instances before discovering that none of those form a pattern occurrence because the few connected C instances don't satisfy the restriction. Other heuristics, such as taking into account multiplicities in associations or the existence of indexes, are presented in [**Zundorf:1996vd**] and used in the PROGRES tool.

Cost models that, in addition to using information from the metamodel, also use statistics and other relevant data from the model are called model-sensitive. As an example, consider the cost model used in the Viatra2 [**Varro2006a**]. According to Varró et al. [**Varro2008**], the cost of a search plan is given by the potential size of the search tree formed by its execution. To estimate that cost, probabilities are calculated using statistics that were collected from a sample of representative models. This is all performed at compile time in Viatra2, so multiple alternative algorithms are generated to perform the same pattern match. At run-time, the best alternative is selected taking into account the current input model's statistics.

An implementation sensitive cost model such as the one presented in [**Batz2008**] and implemented in the GrGen.NET [**Kroll2007**] tool takes not only the size of the search tree into account but also the cost of each individual operation such as the search for all the elements given some type. This allows the tool to seamlessly consider the existence of indexes and other characteristics of its own implementation in the cost model. This is similar to the cost model used in database systems since they typically take the indexes, hard-disc access and other implementation features into account [**Silberschatz:2010uw**].

4.2.4 Pivoting

Contrarily to the previous techniques, that can be performed automatically by the tool, Pivoting requires the user to interfere. In order to apply this technique, the tool has to support rule parametrization and a way to instantiate those parameters with concrete model elements; and the user has to identify which rules are suited to be parametrized and forward previously matched model elements to those rules. As an example, assume that the pattern shown in Figure 4.4 is matched before the pattern of Figure 4.2. A keen transformation engineer parametrizes the second pattern with elements that are to be matched in the first pattern. In that case, the Algorithm 7, that performs the match for the pattern in Figure 4.2, could be invoked with all the occurrences of the sub pattern *AB*.

Algorithm 7 An algorithm to find occurrences of the pattern shown in Figure 4.2 using pivoting.

```
function FINDALLOCCURRENCESBAC(occurrencesAB)
    occurrences ← {}
    for $\{(x, a), (y, b)\} \in$ occurrencesAB do
        for $c \in a.ac$ do
            occurrences ← occurrences $\cup \{(x, a), (y, b), (z, c)\}$
        end for
    end for
    return occurrences
end function
```

The specific mechanism that transports matched elements from one match operation to other vary greatly with each tool. GReAT [**Balasubramanian:2007td**; **Vizhanyo2004**] and MoTif [**Syriani:2008tz**] allow for pivoting. A transformation specification expressed in these languages consists of a network of rules with well defined input parameters (or input interface) and output parameters (output interface). The input parameters declare the rule's incoming occurrences that serve as a starting point for the pattern matching (just as in Algorithm 7. The output interface represents those occurrences that will be transported to the following rules in the network.

4.2.5 Overlapped Pattern Matching

Overlapped Pattern Matching is a technique where two or more patterns are factorized in order to identify a common pattern that can be matched before them. The common pattern occurrences are then passed as parameters to match the remaining patterns of the two rules [**Meszaros:2010wn**]. This is very similar to pivoting but it is performed without user intervention. The impact is that the overall number of pattern matching operations is greatly reduced. For a example of application of this optimization and impact analysis see section 9.6 in page 116.

As an example, consider the patterns shown in Figures 4.2 and 4.4. A tool supporting this technique computes the intersection between the two patterns and obtains the common pattern $B \xleftarrow{ab} A$, i.e., the pattern that is in Figure 4.4. The common pattern is matched before the other two patterns and then its occurrences are passed as parameters to both algorithms. VMTS [**Levendovszky2005**] applies this technique in pairs of similar rules.

There is a wide array of other pattern matching optimization techniques such as the usage of lazy rules in ATL [**Jouault2008**] or the user-specified strategies to solve systems of equations in BOTL [**Braun2003a**] involving several attributes, etc... The ones the we presented were are the most prominent and well documented.

4.3 State of the Art

In this section, we present a summary of the state of the art tools and optimization techniques that they use. The tools we considered are: PROGRES [**Zundorf:1996vd**; **Schurr1994**], BOTL [**Braun2003**; **Braun2003a**], AGG [**Taentzer2004**; **Rudolf2000**], Atom3 [**Lara2004**], Great [**Balasubramanian:2007td**; **Vizhanyo2004**], ATC [**estevez2006atc**], GrGen.NET [**Batz2008**; **Kroll2007**; **Jakumeit2010**], Motif [**Syriani:2008tz**; **Syriani:2008wm**], BeanBag [**Xiong2009**], VMTS [**Meszaros:2010wn**; **Lengyel:2006wa**] and T-Core [**Syriani:2010tq**].

Table 4.1 shows the results of our study. Since model transformation tools evolve very rapidly we have included the year next to the tool in which a paper was published concerning the tool's internal mechanisms to perform pattern matching.

It is important to note that there are tools, such as AGG and GrGen(PSQL) that use a constraint satisfaction solver or a database management system as underlying pattern matching engine. In this sense, a pattern matching process relying on a CSP solver or a DBMS adopts the techniques employed in the underlying engine. CSP solvers perform backtracking search, use heuristics such as the first-fail principle, leverage information about the input model to determine the variables' domain, perform forward checking and other optimization techniques [**Russell:2003tj**]. That is why AGG is characterized as shown in Table 4.1. DBMSs use query evaluation plans with sophisticated cost models that take into account statistics about the relations, indexes on their columns and the individual costs of operations [**Silberschatz:2010uw**].

4.3.1 Discussion

There is a wide variety of approaches to the pattern matching optimization problem. However, each approach is independent from the execution mode (interpretation or compilation) so the optimization techniques identified can be applied in both modes. Also, as said previously, techniques employed by tools that reduce the pattern matching problem to the CSP or DB domain also fit in the presented categorization. These facts allows us to compare the different pattern matching approaches and the tools that support them without having to consider other aspects such as their execution modes or if they perform a reduction to other domain.

In terms of performance, the imperative languages (see Czarnecki's categorization [**Czarnecki2003**]) such as ATC [**estevez2006atc**] or T-Core [**Syriani:2010tq**] or Java, despite not supporting many optimizations techniques, can be extremely fast. This is because the user can choose to directly code the optimizations. So it can apply virtually any optimization technique, as long as the language is expressive enough, which is generally true.

Naturally, those optimization techniques that depend on user intervention are the ones that contribute more to the performance but with an impact in the productivity and maintenance. On the other hand, those techniques that can be applied automatically, require less knowledge from the user and ease the creation and maintenance of the optimized transformation specifications. As shown in Table 4.1 tools that invest more in optimization tend to combine manual and automatic techniques.

All the studied techniques, without exception, target the pattern matching operation. This strengthens the fact that pattern matching is the most critical operation in a model transformation.

Table 4.1: State of art tools and the pattern matching techniques in use.

	Manual	Semi-Automatic		Automatic									
		Local	Global	Local					Global				
			Pivoting	Backtracking					Caching	Indexing			Overlapped P.M.
				Planned				Unplanned		Syntactic			
				Model Sensitive		Metamodel Sensitive	Implementation Sensitive			Type	Attribute	Structural	
Tools				Multiple	Single								
PROGRES (1996)			X			X	X		X		X	X	
BOTL (2003)		X						X					
AGG (2004)			X		X	X	?			X			
Atom3 (2004)	X							X					
Great (2004)			X					X					
ATC (2006)	X												
Viatra2 (2006)	X		X	X	X	X				X		X	
Tefkat (2006)		X			X	X	?			X		X	
ATL (2008)	X	X						X	X				
GrGen.NET (2008)			X		X		X						
Motif (2008)			X					X		X			
BeanBag (2009)			X					X		X		X	
VMTS (2010)	X		X					X		X			X
T-Core	X												

5

Problem Definition

It is very clear by now that, in terms of performance, lower-level and imperative languages are much better than high-level, declarative languages. The problem is that, at a lower level of abstraction, transformations written in imperative languages are hard to create, read and maintain. However, only after transformations written at an high level of abstraction are shown to run fast enough, people will use them at industrial scales. Optimizations, and in particular those that target the pattern matching operation, play a key role.

This trade-off between abstraction and performance is not new, it is at least as old as the first high-level languages appeared.

5.1 Abstraction Penalty

There are plenty of sites, for example [**benchmarksgame**] and [**attractivechaos**], that compare different programming languages with respect to their performance and, by observing one of the results taken from [**attractivechaos**] and shown in figure 5.1, it is clear that higher-level languages are typically much slower than lower level ones. Another trivial conclusion is that compiled code, shown in red, runs a lot faster than interpreted code, in black.

But if higher level languages are so slow when compared to lower level ones, why are they so used? That is because, by applying well know optimization techniques, their performance improved enough to be superseded by the productivity and maintainability advantages. This is what must be done for model transformation languages.

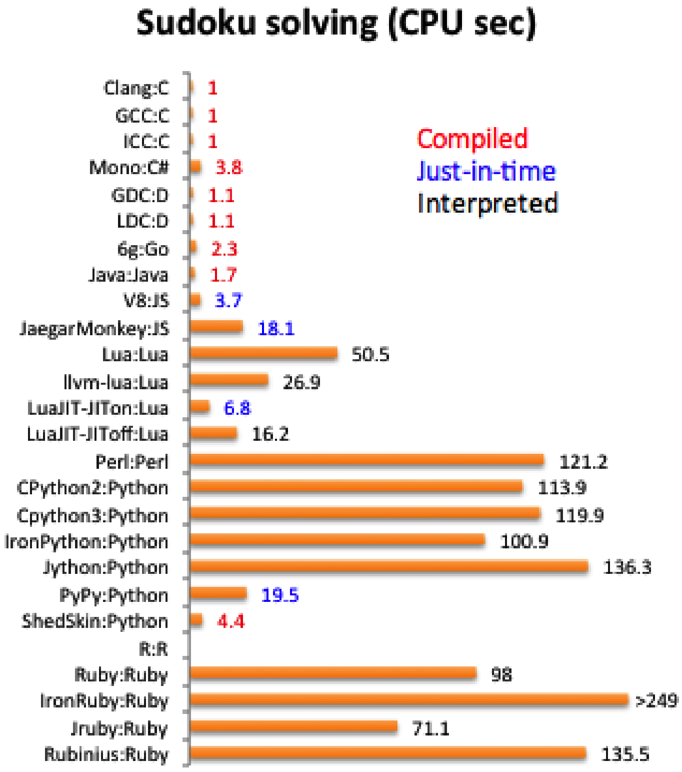


Figure 5.1: Programming languages benchmark on sudoku solving. Taken from [**attractivechaos**].

5.2 How to Avoid the Penalty

In order to understand what is the best way to mitigate the abstraction penalty, we have the study how General Purpose Programming Languages (GPLs) tackled that problem.

Assembly is a lot faster than C. Despite this, only a few people today code entirely in assembly. This is mainly because of three things: the raising complexity of software systems; the raising variety of processor architectures and, last but not least, performance optimization in the C compiler. There has been a lot of research in compiler optimization techniques and they have been applied to the C compiler [**Aho:2008wl**; **Srikant:2008us**]. As a result, the C compiler has evolved to produce assembly code that is efficient enough to make coding directly in assembly a bad investment, i.e., it mitigated, for the most cases, the abstraction penalty.

As is shown in Figure 5.2, the optimizing C compiler does not produce assembly code directly from C, it uses multiple intermediate representations of a program that serve specific optimization techniques. The C programs are translated first through these representations and after that, to assembly code. For the sake of simplicity we are skipping several steps in the compilation process.

Figure 5.2 illustrates one particular representation that is used for many optimizations: the Control Flow Graph (CFG). This representation supports many data-flow analyses that, once performed, result in optimizations such as reaching definitions, live variables, available expressions [**Aho:2008wl**], etc. . . Note that we focused only on data flow analyses because they are machine independent optimization techniques. There are other, machine or architecture dependent but, since we are trying to solve the same (abstraction penalty) problem for model transformation languages which by nature promote platform independence, we are not interested in machine/platform dependent optimizations.

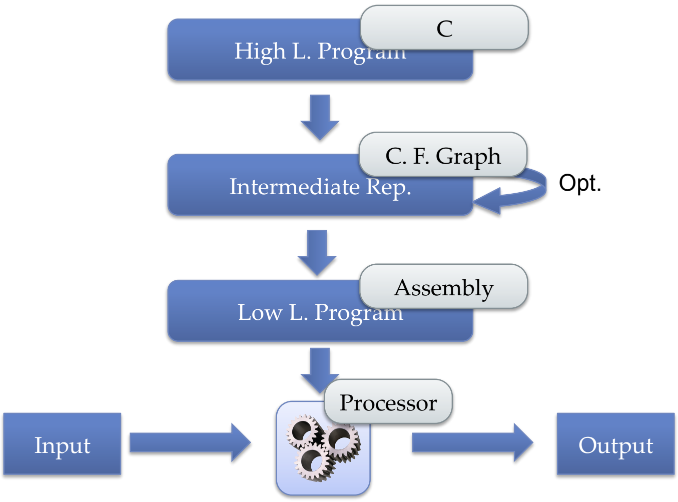


Figure 5.2: C Compilation Overview.

If we bring that architecture for the model transformation languages domain, we have something similar to what is illustrated in Figure 5.3: an High-Level language, a process that translates the high-level representation of transformations to an intermediate one and then another that generates the final representation of the transformation. The high-level transformation language has to promote productivity and maintenance while the lower level language has to be fast. The intermediate representation has to promote analysis and optimization.

DSLtrans [**Barroca:2011wr**] which we presented in section 3.2, page 14, is a good example of high level transformation language and, since it was developed internally, we have a much better understanding of its semantics than we have of other high-level transformation languages.

Java, which we have shown to be really fast in the chart of Figure 3.5 (section 3.5), serves well enough to be our low-level transformation language. Other GPLs such as C or C++ have better performances but Java has a complete model manipulation library support from Eclipse Modelling Framework (EMF).

What is missing in Figure 5.3 is an appropriate intermediate representation.

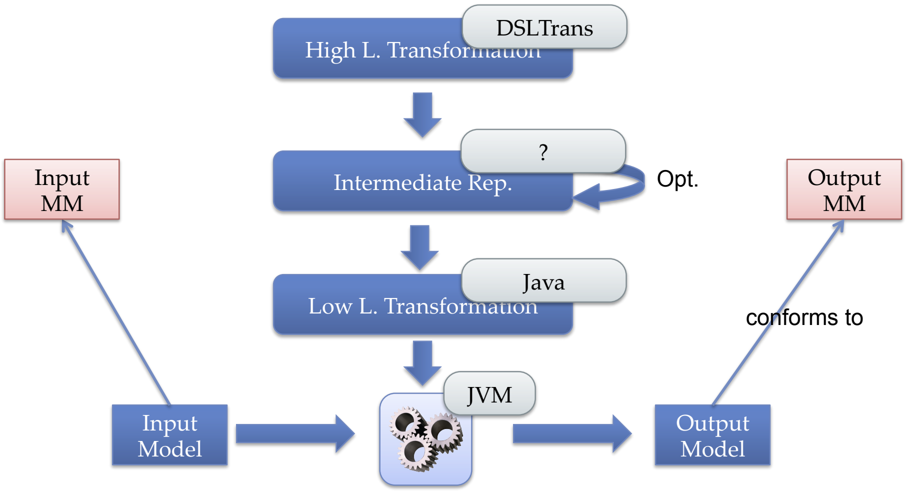


Figure 5.3: High level model transformation language compilation overview.

5.3 General Requirements

As we have seen in Chapter 4, the most performance-critical operation is the pattern matching so it is only natural that this intermediate representation has to support natively the representation of this operation. It also has to allow for a fine-grained control over the how the operation will be executed. Off course, simplicity is always desirable when we want to achieve analyzability.

Since we are in the context of model transformations and model driven development, all the transformations in figure 5.3 should be model transformations and the intermediate representation should be a full fledged language: with syntax and semantics.

With these requirements in mind we set out to design a proper intermediate representation together with its supporting analyses and optimizations. The next chapters tell that story.

6

Design

We need to find a language that links an high-level and productive transformation language to a low-level, fast one. In this thesis we will be using DSLTrans as the high-level language and Java as the low level language but we believe that the principles studied here are applicable to other high-level transformation languages and are independent of Java.

The main desirable traits of our language are: the ability to model the pattern matching operation and both syntactic and the semantic simplicity.

Figure 6.1 shows the general architecture of the solution to the abstraction penalty. The intermediate language is called TrNet (Transformation + Network) because it resembles a network of channels where input model elements and patterns flow and are transformed until they reach the output models. Analysis and optimizations operate as model transformations, processing the network, rearranging channels, defining parameters, etc…

In the following section we will the languages involved in this architecture, namely, the intermediate languages (TrNet), the high-level language (DSLTrans) and the technologies used to design such languages.

6.1 TrNet

A transformation expressed in TrNet resembles a network of channels with a series of filters and intersections. The input model is broken down into several pieces. These pieces are distributed across the channels' entrances, where they are propagated deeper and deeper into the network. In the intersections, elements are mixed together, forming other elements, which continue to be propagated. Some elements are filtered. All the

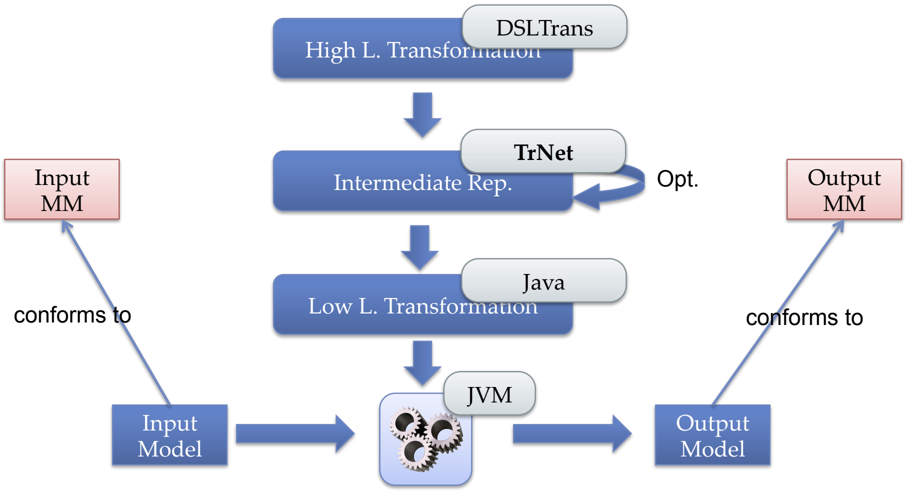


Figure 6.1: Proposed approach to solve the abstraction penalty.

pieces are propagated until they reach the other end of the network, were they combined to form the output model.

What does this analogy have in common with model transformations? Well, if we hard-code the behaviour of these channels, and if we input them the correct input model's elements, then we can get the expected output models' elements without having to repeatedly search the input model for certain patterns.

Traditionally, we look at model transformations with a computational mindset in which there is an engine that actively searches the input model for specific patterns in order to run some rules. Here, we try to invert that search: the input elements go meet the rules that will handle them and the rules just passively wait for them to come. So there is no explicit notion of indexing, search plan optimization, of overlapped pattern matching. But these notions will emerge (some sooner than we thought) as we develop the language and they will be intrinsic to the language and not creations of an optimizing engine.

6.1.1 Transformations

Figure 6.2 shows a simplified version of TrNet's metamodel main concepts and Figure 6.3 shows an example of concrete syntax with a few extra labels and callouts to allow for easier reference. The metamodel is expressed in the Ecore[1] language and the concrete syntax editor was built using Eugenia[2].

A *TrNetModel* represents the transformation itself and contains *Patterns*, *Operators*, *Operands* and *Results*. A *Pattern*, represented as the white outer rounded rectangles in Figure 6.3, is an intermediate storage for pattern elements. A *Combinator*, represented as a grey rounded rectangle in the concrete syntax, is an operation that manipulates the

[1] http://www.eclipse.org/modeling/emft/?project=ecoretools

[2] http://www.eclipse.org/epsilon/doc/eugenia

pattern elements the exist inside *Patterns*. An *External* operator represents a source ou a sink of pattern elements. In the concrete syntax, *External* operators are represented as empty, white rounded rectangles. *AnyOperands* and *AnyResults* are represented as arrows connecting patterns to operators and vice versa, respectively. The concrete syntax example shows one *AnyOperand* and two *AnyResults*.

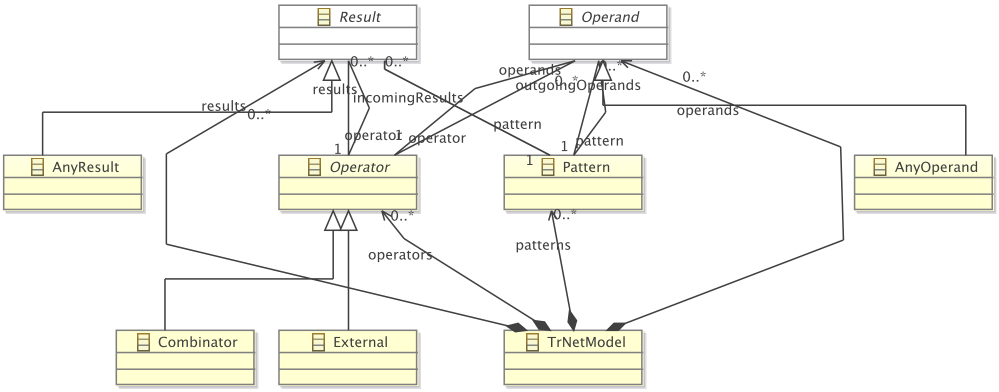


Figure 6.2: Excerpt of TrNet metamodel - top level elements (simplified).

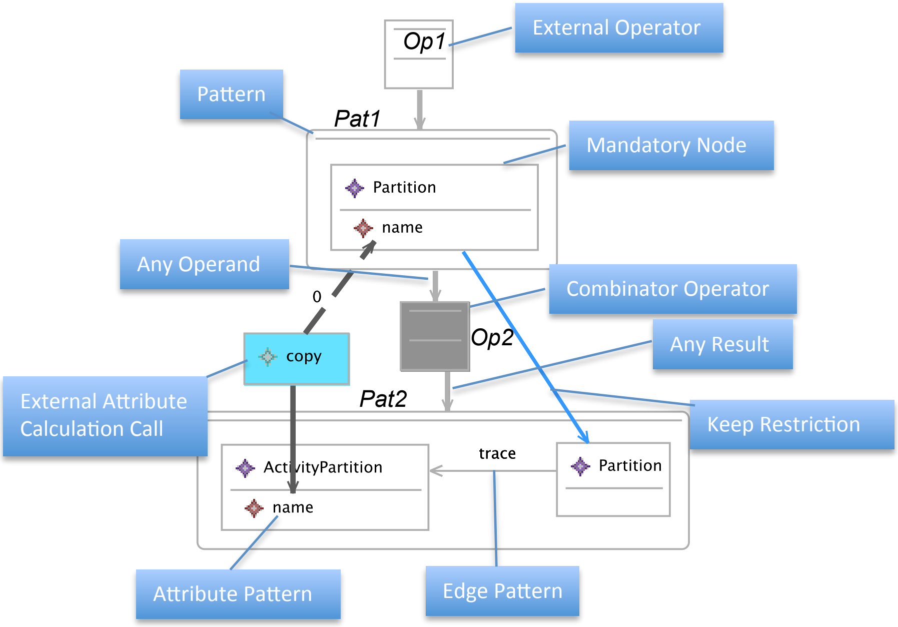


Figure 6.3: TrNet sample transformation labelled.

As figure 6.4 shows, *Patterns* contain *MandatoryNode* elements and *EdgePatterns* that connect those nodes. In Figure 6.3 there is one *MandatoryNode* named "Partition" in the topmost *Pattern* and, in the bottommost *Pattern*, there are two *MandatoryNodes* and an *EdgePattern* named "trace" connecting them. The graph formed by the *MandatoryNodes* and *EdgePatterns* represents the "type" of the pattern. It means that, during run-time, the *Pattern* will store patterns that are instances of the graph represented inside that *Pattern*. This will become more clear as we explore the language. *NodePatterns* also contain *AttributePatterns*. These attributes serve as declarations that can be used to perform *Calculations* or the evaluation of *Conditions* when executing *Operators*.

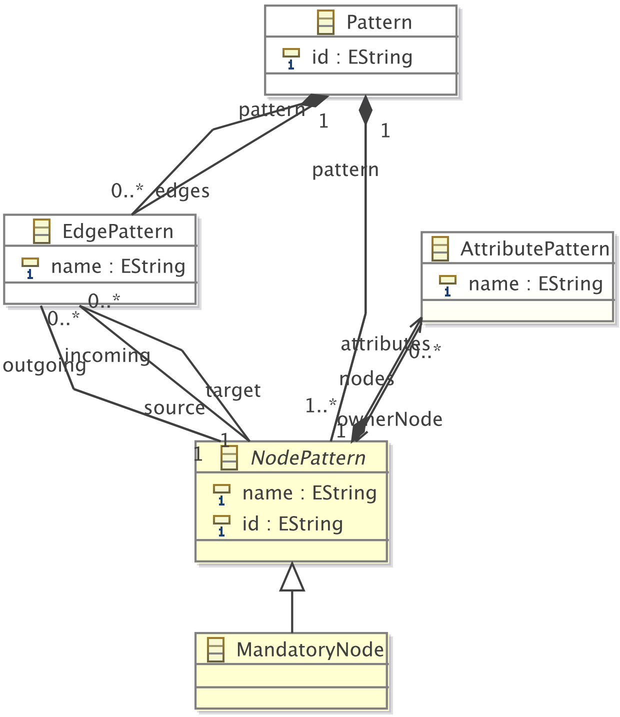


Figure 6.4: Excerpt of TrNet metamodel - nodes and edges (simplified).

The blue rectangle in Figure 6.3 represents an *ExternalAttributeCalculationCall*. In Figure 6.3, it means that the *AttributePattern* "name" of "Partition" will be used to calculate the value of the "name" *AttributePattern* inside the "ActivityPartition" *MandatoryNode*. Figure 6.5 represents the excerpt of the TrNet metamodel that is related to *ExternalAttributeCalculationCalls*. An *ExternalAttributeCalculationCall* is an *AttributeCalculation* characterized by an id attribute and a qualified name. The qualified name represents the operation, pertaining an external library, that will be responsible for performing the calculation. An *ExternalAttributeCalculationCall* can reference zero or more *Parameters* that are given to the external operation when the calculation is performed. Both *NodePatterns* and *AttributePatterns* can be used as *Parameters*. In the concrete syntax, the dashed arrow connecting the blue rectangle to the "name" *AttributePattern* represents a *ParameterRef* element and the full arrow pointing to the "name" *AttributePattern* of the "ActivityPartition" *NodePattern* is were the result of the calculation is stored.

The last uncovered syntactic element of Figure 6.3 is a blue arrow that connects the

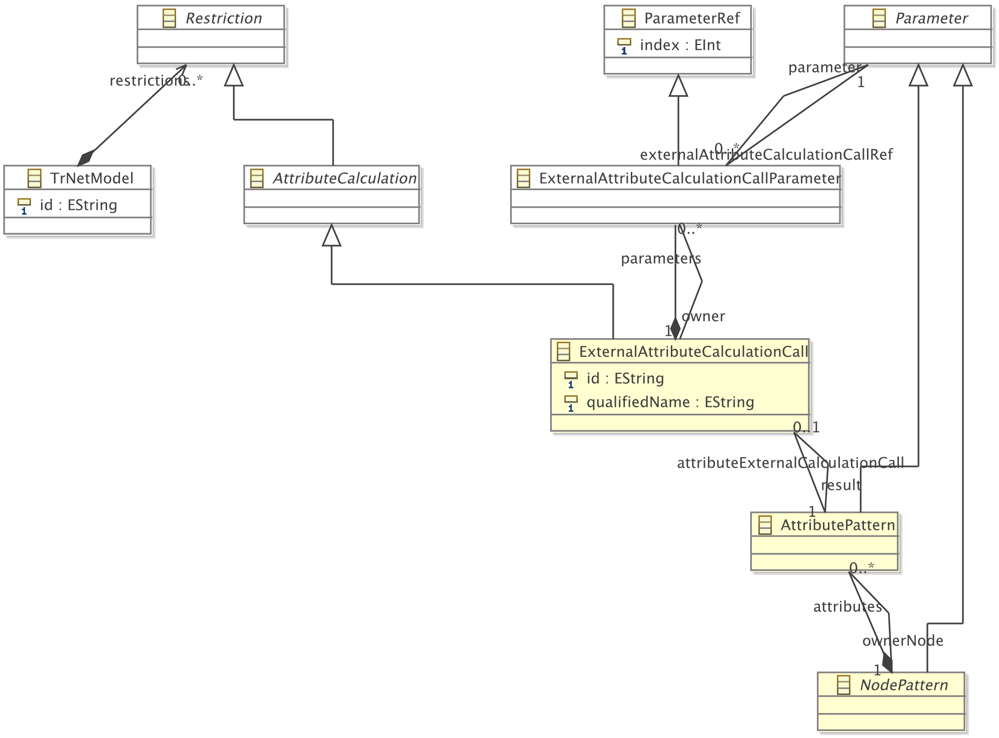


Figure 6.5: Excerpt of TrNet metamodel - Attribute Calculations (simplified).

MandatoryNode "Partition" in the topmost *Pattern* to the *MandatoryNode* in the bottommost *Pattern*. It means that, when executing, the *Combinator* will not create new "Partition" elements but instead, will place the source "Partition" element inside the bottommost pattern. The blue arrow is called a *Keep* restriction. Figure 6.6 shows the part of the metamodel where it appears.

Now that all syntactic elements of Figure 6.3 are introduced, we can see what the transformation presented there means. For the sake of simplicity, we assume that the input model exists, is valid, and is accessible to the transformation. In Chapter 7 we will show how these mechanisms are implemented.

When the transformation is executed, each *Operator* is executed in turn, according to a specific predefined order. For this example, assume that the order is: *External* ($Op1$) and then *Combinator* ($Op2$). In Chapter 9 we explain how to come up with the best possible execution order. When an *Operator* gets executed, it reads elements from its inputs, combines them and writes elements for its outputs. The inputs/outputs of an *Operator* are *Patterns*. As said previously, *Patterns* are a storage mechanism. For now, assume that each *Pattern* in Figure 6.3 is a set. So the topmost *Pattern* is the set $Pat1$ and the bottommost is $Pat2$.

In Figure 6.3, the *External* ($Op1$) operator represents a source of elements that come from the input model. This *External* operator will filter all the input model elements and store only those that are "Partition" instances in the topmost *Pattern* ($Pat1$). For instance,

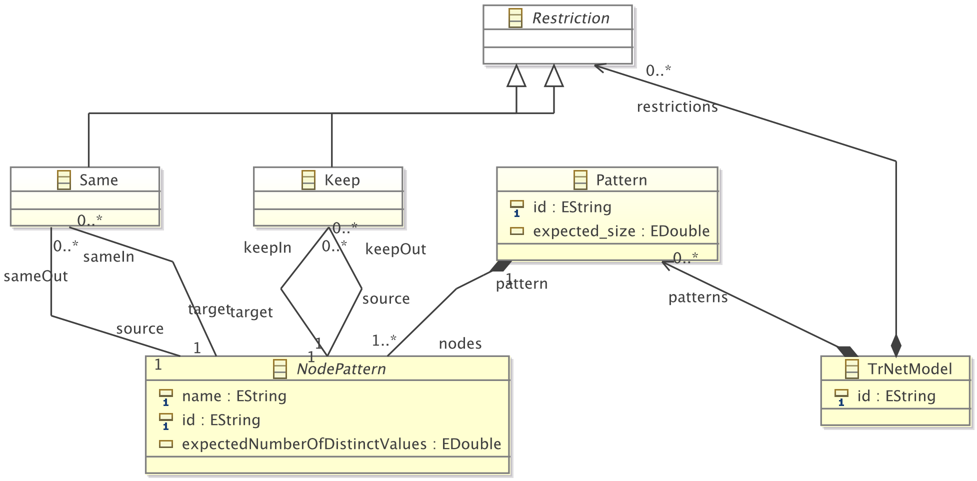


Figure 6.6: Excerpt of TrNet metamodel - Restrictions (simplified).

consider the sample model shown in figure 6.7 which has, among other elements, two "Partition" instances: $p1$ and $p2$. After executing, the *External* ($Op1$) operator will select $p1$ and $p2$ and place them inside the set $Pat1$, as shown in Figure 6.8.

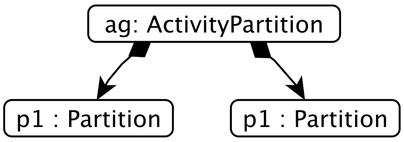


Figure 6.7: Sample model with two Partition instances.

The next operator to be executed is the *Combinator* ($Op2$). In simple terms, it will read each element existing in the set $Pat1$, create a new compound element and add it to the set $Pat2$. Suppose $Op2$ reads $p1$. It then creates a new instance of "ActivityPartition" $ap1$ and sets its "name" attribute to the result of applying the "Copy" function to the attribute $p1.name$. Then, the operator $Op2$ places the pair $(p1, ap1)$ inside the set $Pat2$. Algorithm 8 summarizes $Op2$'s behaviour, where MAKENEWNODE is a generic function that creates model elements of a given type and COPY is the function referred to by the External Attribute Calculation Call.

Notice that the "trace" edge is not represented explicitly in Algorithm 8. That is because, since *Patterns* are strongly typed, we always know that each pair $(pi, apj) \in Pat2$ has a "Partition" instance pi, an "ActivityPartition" instance apj, and a "trace" connection between pi and apj. There are many advantages in this approach:

- There is no need to represent and process associations throughout the transformation, which makes execution time independent of the number of associations managed inside the transformation.

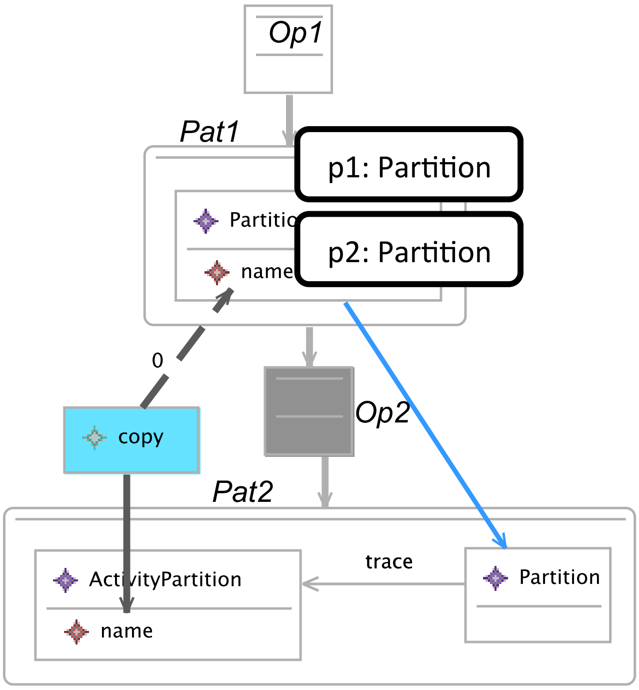


Figure 6.8: Transformation configuration after the completion of Op1.

Algorithm 8 Algorithm showing the behaviour of Op2.

```
function EXECUTEOP2
    for $p \in Pat1$ do
        $ap \leftarrow$ MAKENEWNODE("ActivityPartition")
        $ap.name \leftarrow$ COPY($p.name$)
        $Pat2 \leftarrow Pat2 \cup \{(p, ap)\}$
    end for
end function
```

- Since each *Pattern* is a set, it is necessary to avoid duplicates. Without strongly typed *Patterns*, this operation would be a graph isomorphism but with strongly typed *Patterns* it is simply a matter of comparing tuples of the same length: $(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \Leftrightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$. The comparison of individual elements is performed by comparing a unique identifier, for instance, a memory address. More details about this in Chapter 7.

But there is also a pitfall: it is possible to give the wrong name to an edge without provoking an error in the transformation. Fortunately this can easily be solved with a validation algorithm that searches the transformation for those errors and, even without validation, if the wrong association makes the output model non conformant to the output meta-model, it will not be created in that model.

There are more syntactic elements in TrNet. Consider Figure 6.9, which is the continuation of the transformation that starts in Figure 6.3.

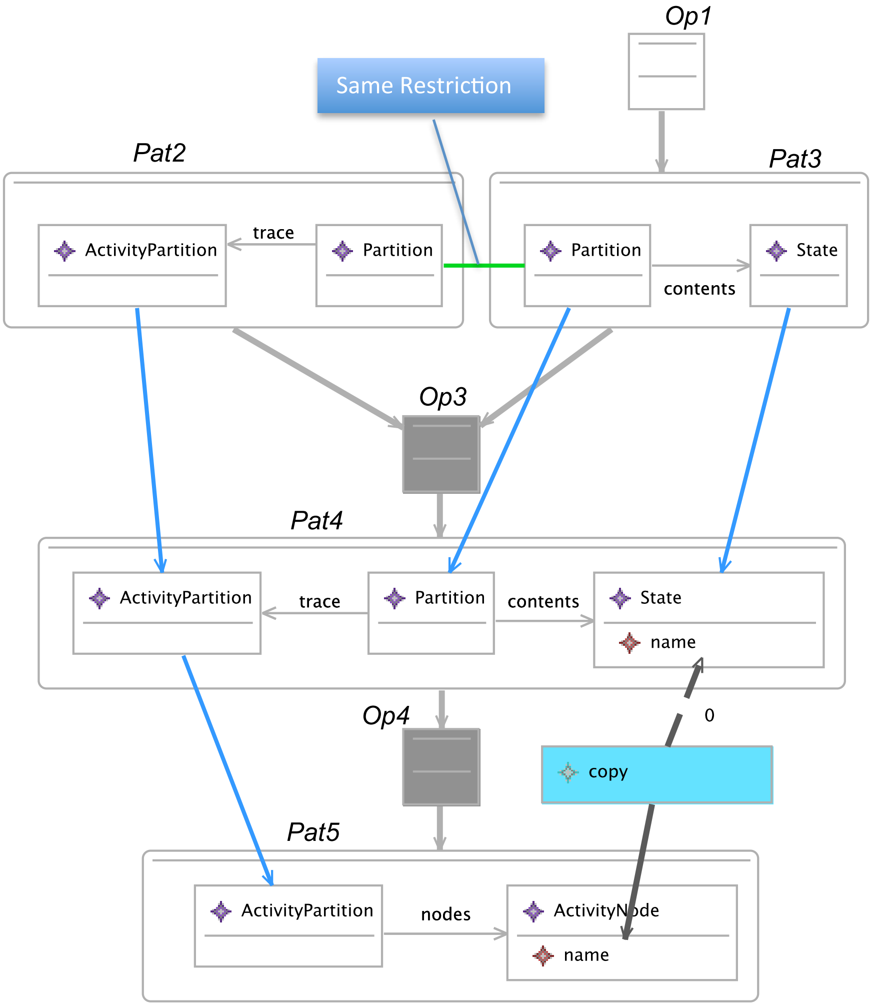


Figure 6.9: TrNet sample transformation (continuation).

The green connection between the two "Partition" *MandatoryNodes* is called a *Same* restriction. Figure 6.6 shows this construction in the metamodel. As an example, assume that $Pat2 = \{(p1, ap1), (p2, ap2)\}$ and $Pat3 = \{(p1, s1), (p1, s2)\}$. The Algorithm 9 shows the behaviour of $Op3$ when executed. You can see that the *Combinator* $Op3$ performs a Cartesian product of the two *Operands* and that, according to the *Same* restriction, guarantees the two partition instances are the same. Only if that is true, an element will be added to the output. If the *Same* restriction did not exist, a result for each element of the Cartesian product would be produced.

Notice that, because of the *Keep* restrictions, no new elements are created. When an operator does not create any new elements, we call it a matching operator since it is only producing more complex patterns from simpler ones.

Figure 6.10 shows the whole transformation and also its final configuration, i.e., the elements in each set after all operators have been executed. In case you are wondering how the elements in the patterns that are connected to the output *External* operator are

Algorithm 9 Algorithm showing the behaviour of Op3.

```
function EXECUTEOP3
    for $\{(p, ap)\} \in Pat2$ do
        for $\{(p', s)\} \in Pat3$ do
            if $p = p'$ then
                $Pat4 \leftarrow Pat4 \cup \{(ap, p, s)\}$
            end if
        end for
    end for
end function
```

going to form the output model, that is explained in section 6.1.3.

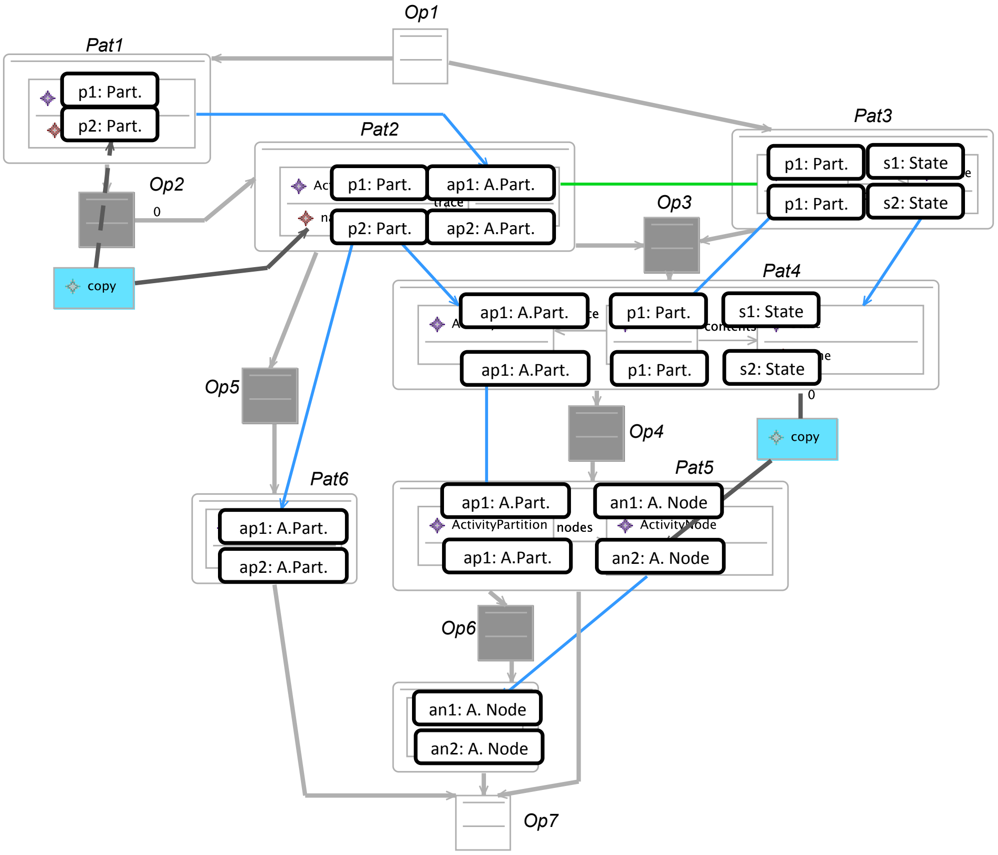


Figure 6.10: TrNet sample transformation (final configuration).

There are a few more syntactic constructions which were not used in the previous examples: *ExternalConditionCalls*, *ExternalActionCalls* and *ExternalCalculationCall*. They are shown in the metamodel excerpt of Figure 6.11, 6.12 and 6.13 and there is an example with *ExternalConditionCalls* and *ExternalActionCalls* in Figure 6.14. The main role of *ExternalCalculations* is to allow for composition of expressions inside TrNet as they are *Parameters*.

In the concrete syntax, *ExternalConditionCalls* are represented as a pink rectangle that

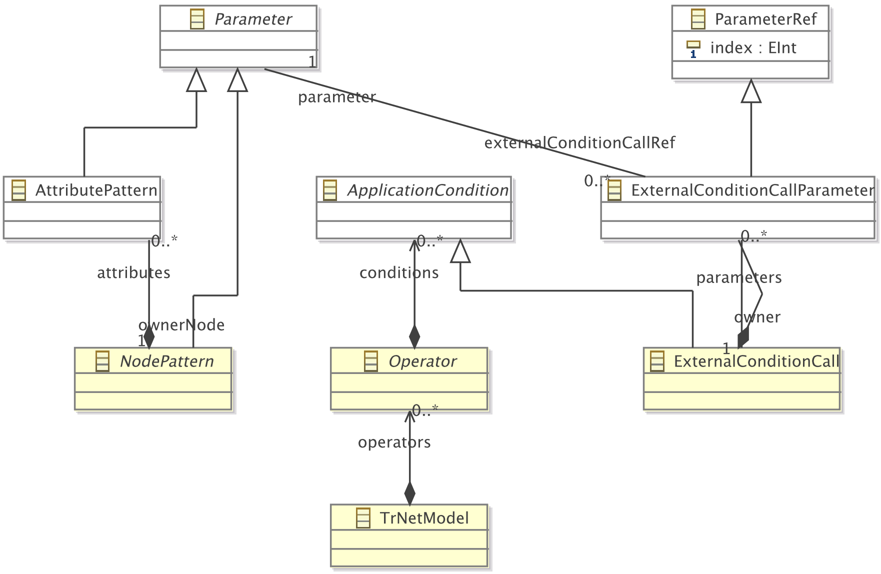


Figure 6.11: Excerpt of TrNet metamodel - Conditions (simplified).

stays inside a *Combinator*. When executing, the *Combinator* will evaluate all its inner conditions and if one of them yields false, then that combination of inputs is not passed to the outputs.

ExternalActionCalls are represented as a green rectangle inside a *Combinator* and they are evaluated for each output produced that *Combinator*.

In Figure 6.14 the two *Combinators* are using actions to count the number of "PseudoState" elements that have the attribute "kind" equal to "join" and to "fork". Algorithm 10 shows the behaviour of the left *Combinator*.

Algorithm 10 Algorithm showing the behaviour of the left Combinator in Figure 6.14.

```
function EXECUTEOP1
    for $p \in Pat1$ do
        if ISJOIN($p.kind$) then
            INCJOIN()
            ...
        end if
    end for
end function
```

There can be simple cycles in a transformation. Figure 6.15 shows a cycle that is used to compute the transitive closure of the "extends" relation between "CClass" elements.

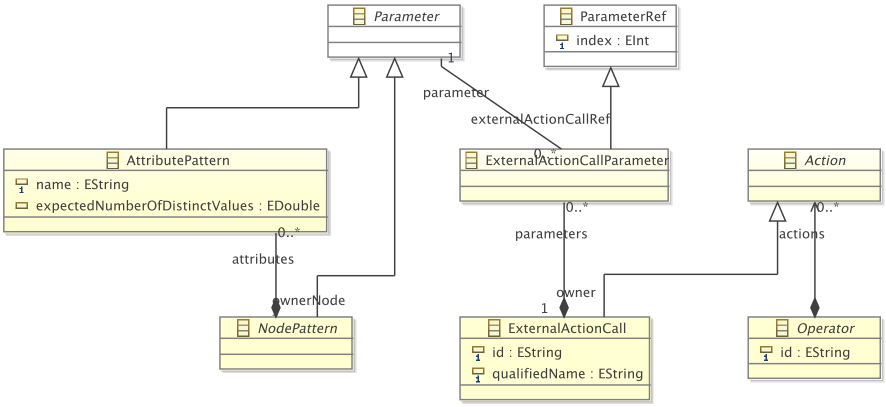


Figure 6.12: Excerpt of TrNet metamodel - Actions (simplified).

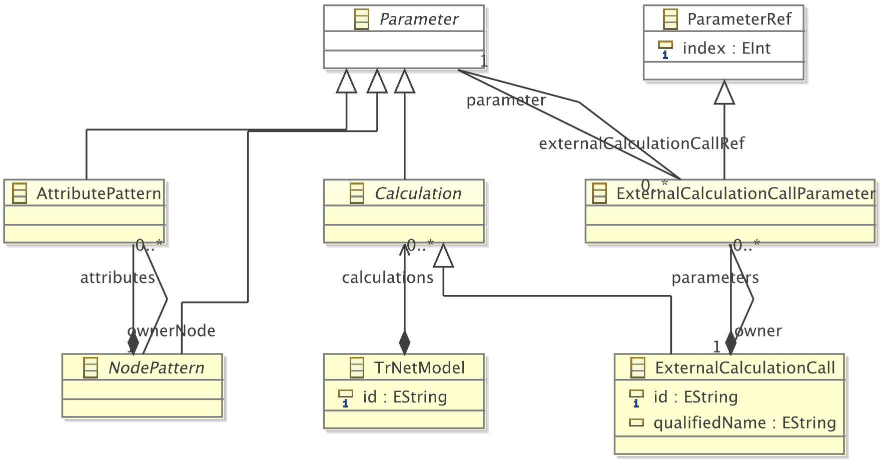


Figure 6.13: Excerpt of TrNet metamodel - Calculations (simplified).

To see how the transformation is executed, let us assume that the transformation of Figure 6.15 has the following initial configuration:

$$C_0 : \begin{array}{rcl} Pat1 & = & \{(c1, c2), (c2, c3)\} \\ Pat2 & = & \{\} \\ Pat3 & = & \{\} \\ Pat4 & = & \{\} \end{array}$$

When executed, the *Combinator* $Op1$ reads every element in $Pat1$ and adds it to $Pat2$, so, after its execution we have the following configuration:

$$C_1 : \begin{array}{rcl} Pat1 & = & \{(c1, c2), (c2, c3)\} \\ Pat2 & = & \{(c1, c2), (c2, c3)\} \\ Pat3 & = & \{\} \\ Pat4 & = & \{\} \end{array}$$

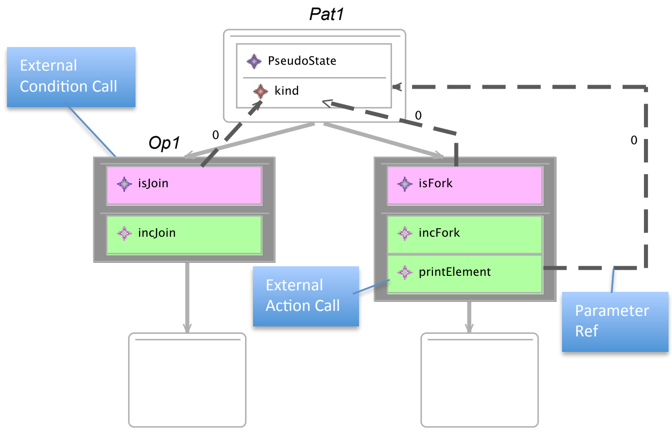


Figure 6.14: TrNet sample transformation with external action and condition calls.

Now the *Combinator* $Op2$ combines every element from $Pat1$ and $Pat2$ and produces only those that have a common "CClass"element into $Pat3$. In this case $((c1, c2), (c2, c3))$ is a valid combination while $((c1, c2), (c1, c2))$ is not. So we get to the following configuration:

$$C_2 : \begin{array}{rcl} Pat1 & = & \{(c1, c2), (c2, c3)\} \\ Pat2 & = & \{(c1, c2), (c2, c3)\} \\ Pat3 & = & \{(c1, c3)\} \\ Pat4 & = & \{\} \end{array}$$

$Op3$ just copies every element in $Pat3$ to $Pat1$, thus, after its execution, we have the configuration:

$$C_3 : \begin{array}{rcl} Pat1 & = & \{(c1, c2), (c2, c3), (c1, c3)\} \\ Pat2 & = & \{(c1, c2), (c2, c3)\} \\ Pat3 & = & \{(c1, c3)\} \\ Pat4 & = & \{\} \end{array}$$

Now the first iteration of the cycle is completed. $Op1$ executes again to produce the configuration:

$$C_4 : \begin{array}{rcl} Pat1 & = & \{(c1, c2), (c2, c3), (c1, c3)\} \\ Pat2 & = & \{(c1, c2), (c2, c3), (c1, c3)\} \\ Pat3 & = & \{(c1, c3)\} \\ Pat4 & = & \{\} \end{array}$$

Now when $Op2$ executes, it won't produce any new result to $Pat3$, so, having detected that $Op2$ is at the beginning of a cycle, there will be no more changes in every pattern that comprises the cycle. So the right thing to do is to select $Op4$ as the next operator to be executed.

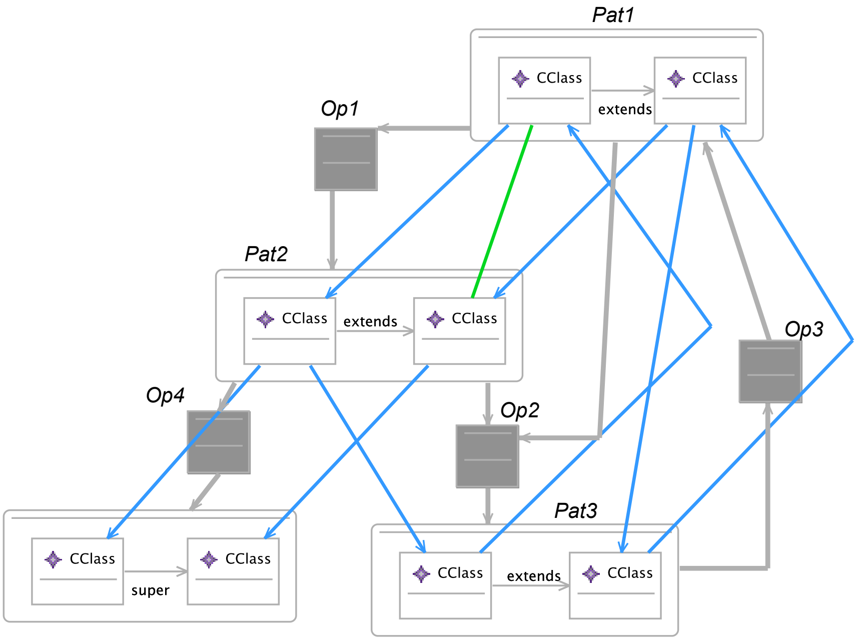


Figure 6.15: TrNet sample transformation with a cycle.

After $Op4$ execution, we have the following, final configuration:

$$C_5 : \begin{array}{lcc} Pat1 & = & \{(c1,c2),(c2,c3),(c1,c3)\} \\ Pat2 & = & \{(c1,c2),(c2,c3),(c1,c3)\} \\ Pat3 & = & \{(c1,c3)\} \\ Pat4 & = & \{(c1,c2),(c2,c3),(c1,c3)\} \end{array}$$

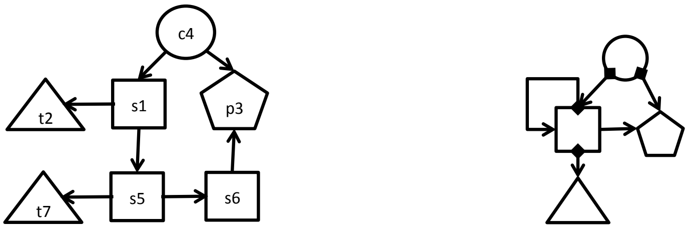
It took five steps to complete the transitive closure computation but, since at compile time we don't have access to all the input models the transformation will be applied to, we cannot estimate how many iterations are necessary for the completion of the transformations. In these cases, a proper operator execution order inference is very important. In Chapter 9 we explore and explain how to get to that execution order.

Notice that, although a Combinator can have any number of patterns, in this thesis and for performance reasons, we only use a maximum of two operands. A Combinator of any number of operands is effectively equivalent to multiple Combinators, each with two operands.

Until now we have presented how a TrNet transformation works and how model elements are processed and propagated throughout the network. But we did not explain how those model elements are read from the input model and how the processed ones are aggregated to form the output model. In the following sections we explain how this is done.

6.1.2 Model Decomposition

In order for the transformation to work, we assume that the input *External* operator selects the appropriate model elements and places them in the corresponding *Patterns*. Consider the model and metamodel shown in Figure 6.16 and the transformation excerpt shown in Figure 6.17.



(a) Sample Model.

(b) Metamodel.

Figure 6.16: Sample input model (left) and corresponding metamodel (right).

After the *External* operator is executed, we get to the following initial configuration:

$$C_0 : \begin{aligned} Pat1 &= \{c4\} \\ Pat2 &= \{(c4, s1)\} \\ Pat3 &= \{s1, s5, s6\} \\ Pat4 &= \{(s1, t2), (s5, t7)\} \\ Pat5 &= \{(s1, s5), (s5, s6)\} \\ Pat6 &= \{t2, t7\} \\ Pat7 &= \{(s6, p3)\} \\ Pat8 &= \{p3\} \\ Pat9 &= \{(c4, p3)\} \end{aligned} \tag{6.1}$$

Before the *External* operator executes there is a process responsible for breaking down the input model into the set[3]:

$$M_{input} = \{c4, (c4, s1), s1, s5, s6, (s1, t2), (s5, t7), (s1, s5), (s5, s6), t2, t7, (s6, p3), p3, (c4, p3)\}$$

The *External* operator only copies the appropriate elements to the corresponding *Patterns*.

The model loading process loads the model and then, starting at the root element[4], visits the entire model. Algorithm 11 shows the set of functions that visit each type of element found in the input model and produce the M_{input} set. Each function is responsible for decomposing the input model recursively starting with its parameter and they all are mutually recursive. The input model is decomposed by calling the VISITCIRCLE function with the root element and an empty set. Note that the process always terminates because

[3] The term "set" is used just to note the non-existent duplicate elements.

[4] We assume that input and output models have one root element. This is a fairly common assumption and it is a good practice when creating a language. It there is a model with more than one root element, it is easy to adapt the algorithms.

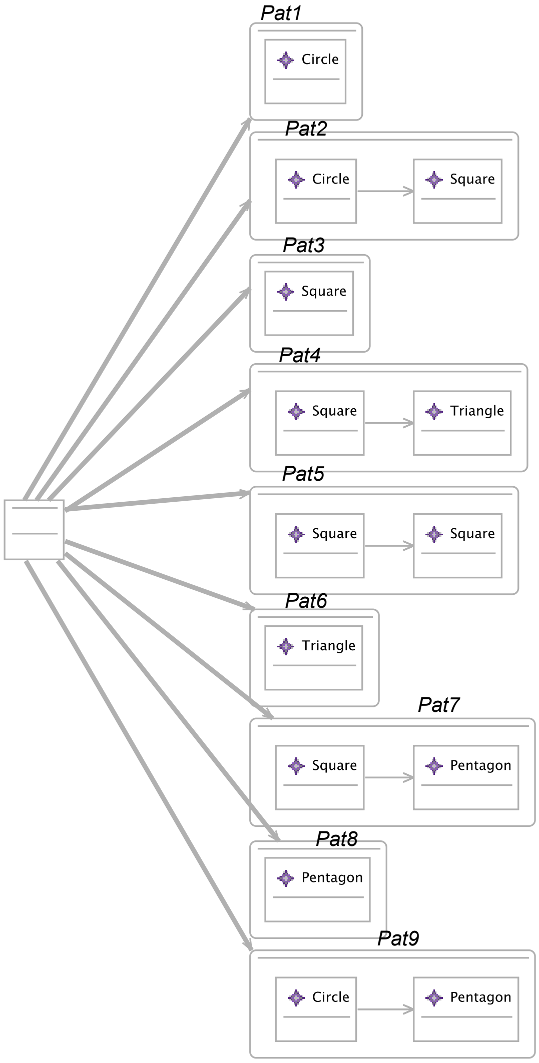


Figure 6.17: Transformation excerpt - inputs.

only the containment relations are navigated recursively. Other relations, such as the one between squares and pentagons, which can form cycles, are not visited recursively.

Notice that the model loading process only produces elements that are *atomic*, i.e., either an element (e.g., $c4$) or an association (e.g., $(c4, c1)$). This means the transformation can only begin with *Patterns* that correspond to either elements or associations. While this can be seen as a limitation, there is no interest in allowing the model loading process to build more complex patterns because that is the role of the transformation. And the transformation is more efficient at that than the model loading process.

The Algorithm 11 depends solely on what is described in the metamodel. In Section 7.1.2 we demonstrate the implementation of this process in more detail. Now how can we build a model from a set of atomic elements?

Algorithm 11 Functions that perform the decomposition process of all models conforming to the metamodel of figure 6.16b.

```
function VISITCIRCLE( $c$, $M_{input}$)
    $M_{input} \leftarrow M_{input} \cup \{c\}$
    for $s \in c.squares$ do
        $M_{input} \leftarrow M_{input} \cup \{(c, s)\}$
        VISITSQUARE($s$, $M_{input}$)
    end for
    for $p \in c.pentagons$ do
        $M_{input} \leftarrow M_{input} \cup \{(c, p)\}$
        VISITPENTAGON($p$, $M_{input}$)
    end for
end function
function VISITSQUARE($s$, $M_{input}$)
    $M_{input} \leftarrow M_{input} \cup \{s\}$
    for $p \in s.pentagons$ do
        $M_{input} \leftarrow M_{input} \cup \{(s, p)\}$
    end for
    for $s' \in c.squares$ do
        $M_{input} \leftarrow M_{input} \cup \{(s, s')\}$
        VISITSQUARE($s'$, $M_{input}$)
    end for
    for $t \in s.triangles$ do
        $M_{input} \leftarrow M_{input} \cup \{(s, t)\}$
        VISITTRIANGLE($t$, $M_{input}$)
    end for
end function
function VISITTRIANGLE($t$, $M_{input}$)
    $M_{input} \leftarrow M_{input} \cup \{t\}$
end function
function VISITPENTAGON($p$, $M_{input}$)
    $M_{input} \leftarrow M_{input} \cup \{p\}$
end function
```

6.1.3 Model Composition

Consider Figure 6.18 which shows the output of a transformation. Assuming the transformation has already completed and we have the configuration C_{final}, how can we build the output model? The metamodel of the output model is the one shown in Figure 6.16b.

$$C_{final} : \begin{array}{rcc} Pat1' & = & \{cd\} \\ Pat2' & = & \{(cd, sa), (cd, sw)\} \\ Pat3' & = & \{sa, se, sf\} \\ Pat4' & = & \{(sa, tb), (se, tg)\} \\ Pat5' & = & \{(sa, se), (se, sf)\} \\ Pat6' & = & \{tb, tg\} \\ Pat7' & = & \{(sf, pc), (sx, py)\} \\ Pat8' & = & \{pc\} \\ Pat9' & = & \{(cd, pc), (cd, pv)\} \end{array} \tag{6.2}$$

Similarly to the model loading process, the model storage process performs a visit to the output model, at each step collecting the model elements that ought to be there. This information comes exclusively from the metamodel of Figure 6.16b.

Algorithm 12 shows the behaviour of the output model storage process. The output model is built by calling the *BuildModel* with the final configuration of the transformation. In this example we show how the output model is built starting in the circle but the output model can be built from any element in the metamodel.

Apart from the *BuildModel*, each function corresponds to a relation in the metamodel of figure 6.16b starting from an element given as parameter. It is responsible for materializing that relation in the output model. The most important step in this materialization process is to find all elements that are connected to the parameter by the relation that the function corresponds to. This is done by the GET<SOURCETYPE><RELATION> functions which we do not show for brevity reasons but, as an example, consider the GETCIRCLESQUARES function shown in Algorithm 13. In simple terms, it selects all the squares that are connected to the given circle and that must exist alone in M_{output}. Notice that the this last condition guaranties that there will be no *dangling edges* in the output model. For instance, the relation element (cd, sw) does not appear in the final model because sw does not exist by itself, only a relation exists. All the other GET<SOURCETYPE><RELATION> function are very similar to GETCIRCLESQUARES.

The recursive process always terminates because only the functions that materialize containment relations have recursive calls. For more details about the implementation of the composition process, please refer to section 7.1.4.

Algorithm 12 Functions that perform the composition process of all models conforming to the metamodel of figure 6.16b.

```
function BuildModel( $M_{output}$)
    $output \leftarrow \{\}$
    for $c \in$ GetCircles($M_{output}$) do
        BuildCirclePentagons($c$, $M_{output}$)
        BuildCircleSquares($c$, $M_{output}$)
        $output \leftarrow output \cup \{c\}$
    end for
     return $output$
end function
function BuildCirclePentagons( $c$, $M_{output}$)
    for $p \in$ GetCirclePentagons($c$ , $M_{output}$) do
        $c.pentagons \leftarrow c.pentagons \cup \{p\}$
    end for
end function
function BuildCircleSquares( $c$, $M_{output}$)
    for $s \in$ GetCircleSquares($c$ , $M_{output}$) do
        BuildSquarePentagons($s$, $M_{output}$)
        BuildSquareTriangles($s$, $M_{output}$)
        BuildSquareSquares($s$, $M_{output}$)
        $c.squares \leftarrow c.squares \cup \{s\}$
    end for
end function
function BuildSquareSquares( $s$, $M_{output}$)
    for $s' \in$ GetSquarePentagons($s$ , $M_{output}$) do
        BuildSquarePentagons($s'$, $M_{output}$)
        BuildSquareTriangles($s'$, $M_{output}$)
        BuildSquareSquares($s'$, $M_{output}$)
        $s.squares \leftarrow s.squares \cup \{s'\}$
    end for
end function
function BuildSquarePentagons( $s$, $M_{output}$)
    for $p \in$ GetSquarePentagons($s$ , $M_{output}$) do
        $s.pentagons \leftarrow s.pentagons \cup \{p\}$
    end for
end function
function BuildSquareTriangles( $s$, $M_{output}$)
    for $t \in$ GetSquareTriangles($s$ , $M_{output}$) do
        $s.triangles \leftarrow s.triangles \cup \{t\}$
    end for
end function
```

Algorithm 13 GetCircleSquares function.

```
function GetCircleSquares( $c$, $M_{output}$) return $\{s_x \in M_{output} | s_x : Square \wedge (c, s_x) \in M_{output} \wedge s_x \in M_{outpu}$
end function
```

Figure 6.19 shows the final output model produced by applying the function *Build-Model* with the set

$$M_{output} = \{cd, (cd, sa), (cd, sw), sa, se, sf, (sa, tb), (se, tg), (sa, se), (se, sf)\} \cup \\ \{tb, tg, (sf, pc), (sx, py), pc, (cd, pc), (cd, pv)\}$$

, which comes from the final configuration C_{final}.

Now we have the whole picture of a TrNet transformation execution: first the model load process breaks down the input model into atomic elements; these elements are published to the input *External* operator, which in turn places the appropriate elements in the corresponding *Patterns*; the transformation executes, propagating the elements and transforming them into other elements through the network onto the final patterns (connected to the output *External* operator); the output *External* operator takes these atomic elements and delivers them to the model storage process; the model storage process aggregates the atomic elements into a full output model. In the next chapter we will explain in more detail each of these processes and how they fit together.

TrNet is considered a low level language because elements have to be combined to form more complex patterns and that combination has to be explicit in the transformation. While this hinders productivity and maintenance, analysis and optimizations can be greatly simplified. More about this in Chapter 9.

6.2 DSLTrans

While TrNet is a low level model transformation language, DSLTrans [**Barroca:2011wr**] is considered to be an high level one because transformations are comprised of a set of rules that contain patterns of any complexity.

Figure 6.20 shows a transformation expressed in DSLTrans that is equivalent to the one expressed in TrNet shown in figure 6.10. It is comprised of 2 layers and 3 simple rules. The topmost rounded rectangle is a FilePort and represents an input model. The first layer, represented by a blue rounded rectangle is executed first. Its rules state that each Partition found in the input model is translated into an ActivityPartition in the output model and that each State is translated into an ActivityNode. The "name" attribute of both elements is copied to the corresponding ones.

DSLTrans transformations are formed by a set of FilePorts (in figure 6.20 we have only one called "InputActivityDiagram") and a list of Layers. Both FilePorts and Layer require the identification of a metamodel. For FilePorts, this is the input model's metamodel and for Layer, this is the output model's metamodel. Each Layer can produce an output model, as long as its "OutputFilePathURI" attribute is properly filled.

A Layer is comprised of Rules and each Rule contains one or more MatchModels and an ApplyModel. Each MatchModel has one or more match elements and each ApplyModel contains one or more apply elements. Match elements can be AnyMatchClass, PositiveMatchAssociation, BackwardLinks, etc. . . For the sake of simplicity, we will mostly work with these elements. Apply elements can be ApplyClasses or ApplyAssociations.

In order to better understand how the DSLTrans engine processes transformations, consider the model illustrated in figure 6.21 and the transformation shown in figure 6.20.

Each layer is executed sequentially according to its dependencies (expressed by the arrows between Layers). The execution of a layer consists of executing each individual rule in a non-deterministic order.

When executing the rule P2A, the engine will search for all Partition instances in the input model and then translate each partition into an ActivityPartition. Thus, the rule's output is $R_{P2A} = \{(ap1, p1)\}$. We represent each Partition in the output of the rule because one traceability link has been created for each ActivityPartition generated that leads to the corresponding Partition. The traceability link for rule P2A is identified by P2A_Trace. Next, Rule S2A gets executed. It takes each State instance in the input model and produces one ActivityNode instance. The rule's output is $R_{S2A} = \{(s1, an1), (s2, an2)\}$. Again, in addition to the newly created ActivityNodes, we represent the States that originated them because of the traceability link being created.

When rule C2N, in layer Relations, executes, it searches for all Partitions instances connected to State elements, and also the ActivityNode and ActivityPartition elements that were used in the previous Layers' execution. So the execution of a rule with backward links not only searches for instances of the match pattern in the input model, but also combines those results with the traceability links produced in the previous layers' rules. This rule creates a new relation between the previously created ActivityPartitions and ActivityNodes. The result is $R_{C2N} = \{(ap1, an1), (ap1, an2)\}$. This rules does not create traceability links because no name is given to them (as it is done in rules P2A and S2A). The resulting output model is shown in Figure 6.22.

Transformations expressed in DSLTrans normally start by translating individual elements in the first layer, and creating the relevant traceability links between them. Then, in the following layers the rules capture more complex patterns and create more relations and elements in the output model.

During the transformation process, DSLTrans engine has to repeatedly perform pattern matching in the input model and in all the traceability links. Once for each rule.

Currently, transformations expressed are executed using a Prolog engine: the input models are translated into facts and, upon each rule's execution, those facts are queried to obtain the matches. Despite Prolog's highly optimized engine, for large transformation, this process is really slow. The output model is built throughout the transformation, after each rule is applied.

In the next chapter we will show how DSLTrans transformations can be translated into TrNet transformations.

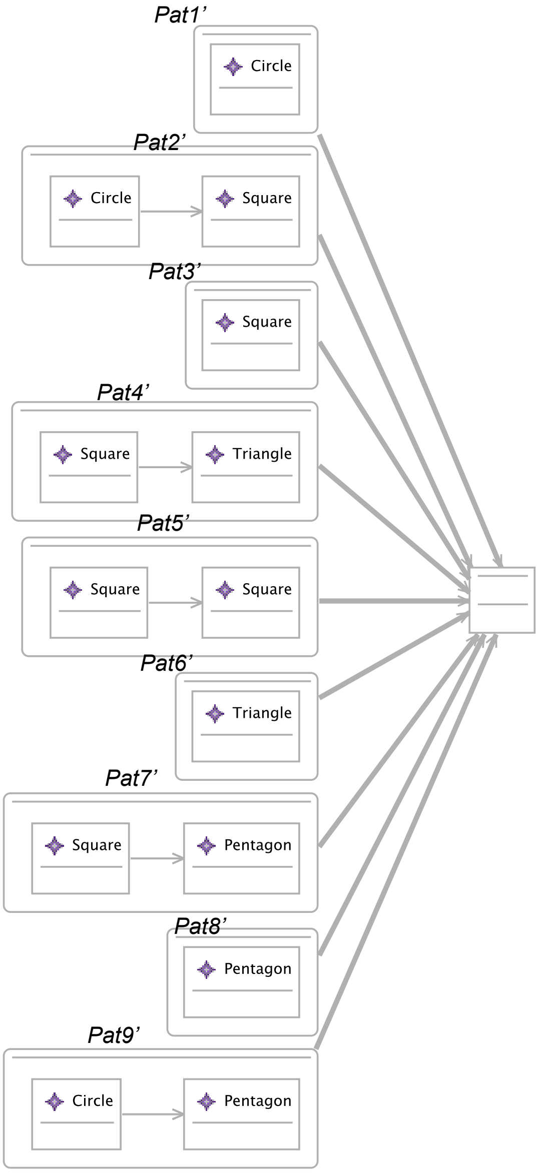


Figure 6.18: Transformation excerpt - outputs.

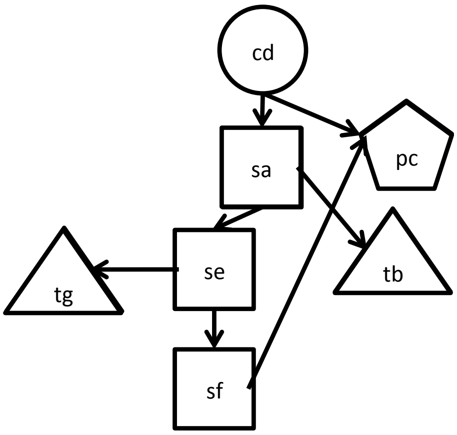


Figure 6.19: Output model build with Algorithm 12 .

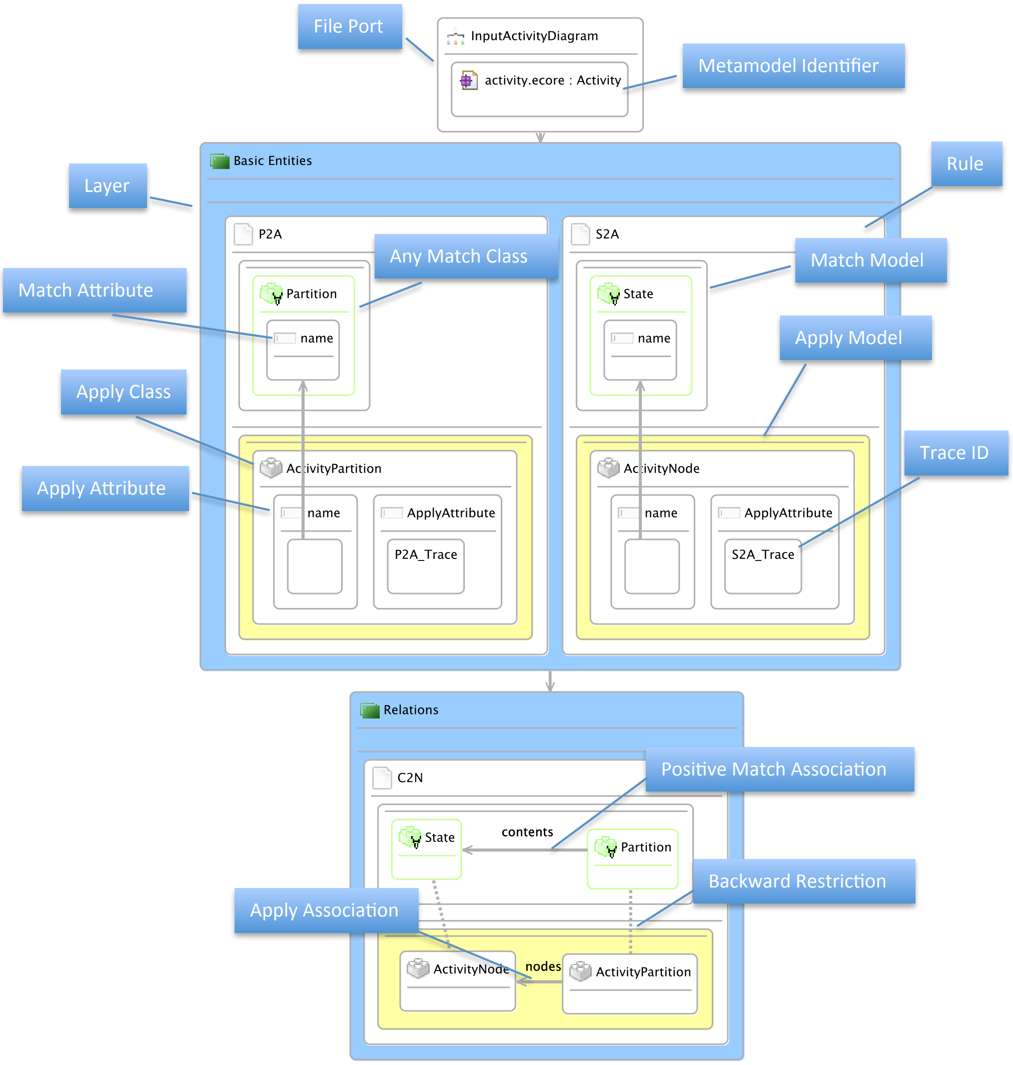


Figure 6.20: DSLTrans sample transformation equivalent to the one in figure 6.10.

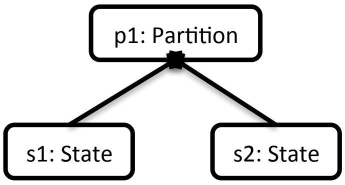


Figure 6.21: Sample activity diagram model (UML 1.4).

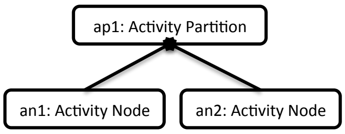


Figure 6.22: Sample activity diagram model (UML 2.1).

7

Implementation

7.1 TrNet Compilation

Transformations in TrNet are compiled to Java code and ran.

We decided to compile the transformations instead of interpret them for obvious performance reasons.

We use Java as the target language because Java is really fast (see figure 3.5 in page 19) and has a great library for model management (EMF) which helps a lot when loading and storing models.

When executed, a complete TrNet transformation has three phases: input model decomposition, transformation and output model composition. Figure 7.1 illustrates these three processes and the information used to generate the code that implements them.

Both composition and decomposition processes' code depend only on metamodels. They are completely independent of the transformation. The transformation code is generated entirely from the transformation specification, written in TrNet. To facilitate communication between the processes, there is a common representation for model elements, listeners and publishers.

7.1.1 Runtime

In order to keep the processes as loosely coupled as possible, the publisher/listener pattern is used to pass information between them and there is a common representation for model elements. Figure 7.2 shows the class diagram.

A ModelPattern is a generic representation of an arbitrary pattern composed of ModelNodes and ModelEdges connecting those nodes. ModelNodes represent model nodes

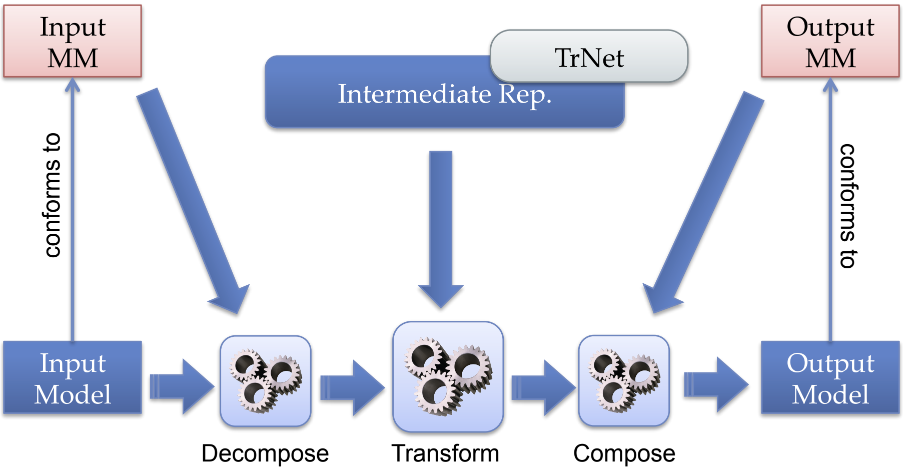


Figure 7.1: TrNet transformation execution process.

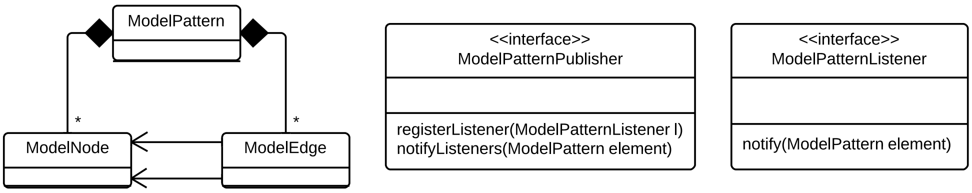

ModelPattern
*
*
ModelNode
ModelEdge
<<interface>>
ModelPatternPublisher
registerListener(ModelPatternListener l)
notifyListeners(ModelPattern element)
<<interface>>
ModelPatternListener
notify(ModelPattern element)

Figure 7.2: TrNet common runtime.

and ModelEdges represent the associations in the model. ModelNodes have a set of types which represents all types from which the actual model element inherits from, and a map of attributes. A ModelEdge has a name, which is the name of the association it represents. It is important to note that two ModelNodes are compared by comparing their memory address (e.g., using the == operator in Java). No attributes are compared.

A ModelPattern publisher is an entity that can pass ModelPatterns to a set of listeners.

7.1.2 Decomposition

The decomposition process is responsible for loading the input model, visiting each element and relation in the model, convert that element or relation to a ModelPattern, and deliver the ModelPattern to the transformation's input *External* operators.

It is implemented in Java and uses the generated classes from the metamodel to load the model to memory. EMF takes a metamodel and generates a set of classes that allow the representation and manipulation of models, conforming to that metamodel, in memory.

Most of the decomposition process is implemented in one single class: the InputVisitor class. Figure 7.3 shows the class and its relations to the common runtime classes. The class is generated from a metamodel and the name of the class is the metamodel name concatenated with InputVisitor. The InputVisitor is a ModelPatternPublisher and depends on the ModelPattern, ModelNode and ModelEdge classes to represent the input model elements.

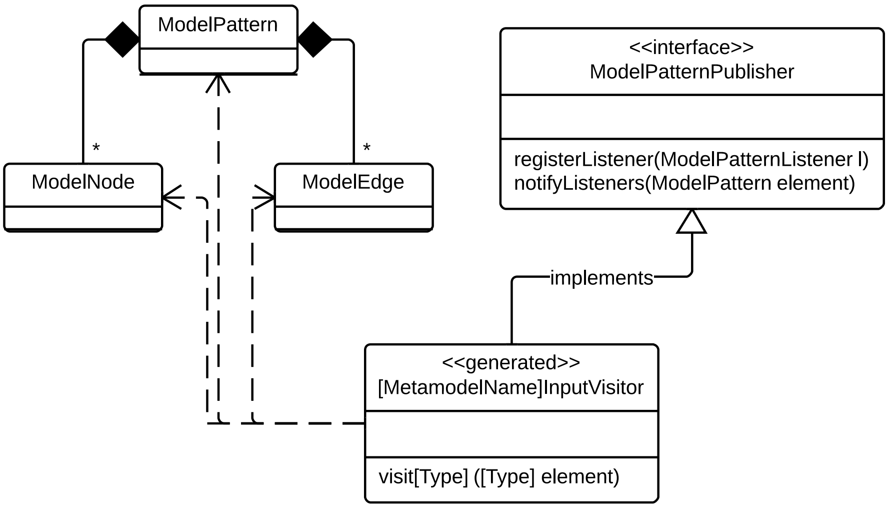

ModelPattern
*
ModelNode
*
ModelEdge
<<interface>>
ModelPatternPublisher
registerListener(ModelPatternListener l)
notifyListeners(ModelPattern element)
implements
<<generated>>
[MetamodelName]InputVisitor
visit[Type] ([Type] element)

Figure 7.3: Decomposition process classes.

If the input visitor was generated from the metamodel shown in figure 7.4 it would have the code similar to listing 7.1. In addition to the constructor and a utility method to load a model given a file path which are not shown in the Algorithm, there is a visit function for each type declared in the metamodel. Inside a visit function, the input visitor publishes a new ModelPattern with information about the current element being visited; the visitor then produces a model pattern for each relation that the current element has with other elements, according to the information given in the metamodel; and then recursively visits each element that is contained in the current one. A map is used to avoid creating two ModelNode instances for the same element in the metamodel. The memory address of each ModelNode is what identifies it uniquely.

Listing 7.1: Input visitor code generated from the metamodel shown in Figure 7.4

```
public class ShapesInputVisitor implements ModelPatternPublisher {

  LinkedList<ModelPatternListener> listeners;
  HashMap<EObject, ModelNode> nodesMap;

  public ShapesInputVisitor() {
    ...
  }
  public void registerListener(ModelPatternListener listener) {
    ...
  }
  public void notifyListeners(ModelPattern element) {
    ...
  }

```

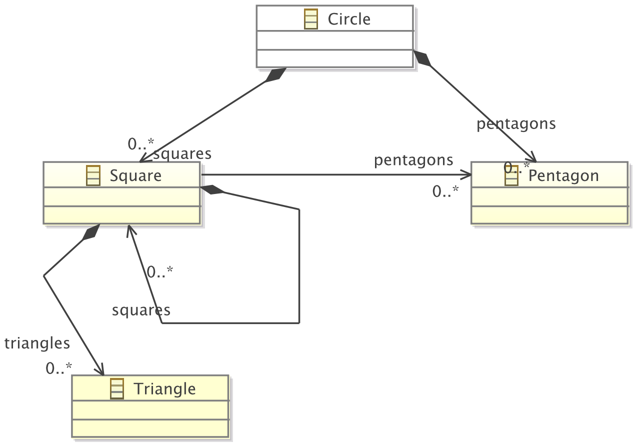


Figure 7.4: Shapes metamodel (equivalent to the one in Figure 6.16b.

```
public Circle load(String path) {
  ShapesPackage.eINSTANCE.eClass();
  Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
  Map<String, Object> m = reg.getExtensionToFactoryMap();
  m.put("xmi", new XMIResourceFactoryImpl());
  ResourceSet resSet = new ResourceSetImpl();
  Resource resource = resSet.getResource(URI.createURI(path), true);
  Circle rootEClass = (Circle) resource.getContents().get(0);
  return rootEClass;
}

public void visitSquare(Square element) {
  ModelNode node;
  if (!nodesMap.containsKey(element)) {
    node = new ModelNode();
    if (element.getClass() == SquareImpl.class) {
      node.types.add("Square");
    }
    nodesMap.put(element, node);
  } else {
    node = nodesMap.get(element);
  }
  ModelPattern pattern = new ModelPattern();
  pattern.nodes.add(node);
  notifyListeners(pattern);
  for (Triangle elementTarget : element.getTriangles()) {
    ModelEdge edge = new ModelEdge();
    edge.name = "triangles";
    ModelNode nodeTarget;
    if (!nodesMap.containsKey(elementTarget)) {
      nodeTarget = new ModelNode();
      if (elementTarget.getClass() == TriangleImpl.class) {
        nodeTarget.types.add("Triangle");
        }
        nodesMap.put(elementTarget, nodeTarget);
      } else {
        nodeTarget = nodesMap.get(elementTarget);
      }
      edge.source = node;
      edge.target = nodeTarget;
      pattern = new ModelPattern();
      pattern.nodes.add(node);
      pattern.nodes.add(nodeTarget);
      pattern.edges.add(edge);
      notifyListeners(pattern);
    }
    for (Pentagon elementTarget : element.getPentagons()) {
      ModelEdge edge = new ModelEdge();
      edge.name = "pentagons";
      ModelNode nodeTarget;
      if (!nodesMap.containsKey(elementTarget)) {
        nodeTarget = new ModelNode();
        if (elementTarget.getClass() == PentagonImpl.class) {
          nodeTarget.types.add("Pentagon");
        }
        nodesMap.put(elementTarget, nodeTarget);
      } else {
        nodeTarget = nodesMap.get(elementTarget);
      }
      edge.source = node;
      edge.target = nodeTarget;
      pattern = new ModelPattern();
      pattern.nodes.add(node);
      pattern.nodes.add(nodeTarget);
      pattern.edges.add(edge);
      notifyListeners(pattern);
    }
    for (Square elementTarget : element.getSquares()) {
      ModelEdge edge = new ModelEdge();
      edge.name = "squares";
      ModelNode nodeTarget;
      if (!nodesMap.containsKey(elementTarget)) {
        nodeTarget = new ModelNode();
        if (elementTarget.getClass() == SquareImpl.class) {
          nodeTarget.types.add("Square");
        }
        nodesMap.put(elementTarget, nodeTarget);
      } else {
        nodeTarget = nodesMap.get(elementTarget);
      }
      edge.source = node;
      edge.target = nodeTarget;
      pattern = new ModelPattern();
```

```java
      pattern.nodes.add(node);
      pattern.nodes.add(nodeTarget);
      pattern.edges.add(edge);
      notifyListeners(pattern);
    }
    for (Triangle child : element.getTriangles()) {
      if (child.getClass() == TriangleImpl.class) {
        visitTriangle((Triangle) child);
      }
    }
    for (Square child : element.getSquares()) {
      if (child.getClass() == SquareImpl.class) {
        visitSquare((Square) child);
      }
    }
  }

  public void visitCircle(Circle element) {
    ModelNode node;
    if (!nodesMap.containsKey(element)) {
      node = new ModelNode();
      if (element.getClass() == CircleImpl.class) {
        node.types.add("Circle");
      }
      nodesMap.put(element, node);
    } else {
      node = nodesMap.get(element);
    }
    ModelPattern pattern = new ModelPattern();
    pattern.nodes.add(node);
    notifyListeners(pattern);
    for (Pentagon elementTarget : element.getPentagons()) {
      ModelEdge edge = new ModelEdge();
      edge.name = "pentagons";
      ModelNode nodeTarget;
      if (!nodesMap.containsKey(elementTarget)) {
        nodeTarget = new ModelNode();
        if (elementTarget.getClass() == PentagonImpl.class) {
          nodeTarget.types.add("Pentagon");
        }
        nodesMap.put(elementTarget, nodeTarget);
      } else {
        nodeTarget = nodesMap.get(elementTarget);
      }
      edge.source = node;
      edge.target = nodeTarget;
      pattern = new ModelPattern();
      pattern.nodes.add(node);
      pattern.nodes.add(nodeTarget);
      pattern.edges.add(edge);
      notifyListeners(pattern);
    }
    for (Square elementTarget : element.getSquares()) {
      ModelEdge edge = new ModelEdge();
      edge.name = "squares";
      ModelNode nodeTarget;
      if (!nodesMap.containsKey(elementTarget)) {
        nodeTarget = new ModelNode();
        if (elementTarget.getClass() == SquareImpl.class) {
          nodeTarget.types.add("Square");
        }
        nodesMap.put(elementTarget, nodeTarget);
      } else {
        nodeTarget = nodesMap.get(elementTarget);
      }
      edge.source = node;
      edge.target = nodeTarget;
      pattern = new ModelPattern();
      pattern.nodes.add(node);
      pattern.nodes.add(nodeTarget);
      pattern.edges.add(edge);
      notifyListeners(pattern);
    }
    for (Pentagon child : element.getPentagons()) {
      if (child.getClass() == PentagonImpl.class) {
        visitPentagon((Pentagon) child);
      }
    }
    for (Square child : element.getSquares()) {
      if (child.getClass() == SquareImpl.class) {
        visitSquare((Square) child);
      }
    }
  }

  public void visitPentagon(Pentagon element) {
    ModelNode node;
    if (!nodesMap.containsKey(element)) {
      node = new ModelNode();
      if (element.getClass() == PentagonImpl.class) {
        node.types.add("Pentagon");
      }
      nodesMap.put(element, node);
    } else {
      node = nodesMap.get(element);
    }
    ModelPattern pattern = new ModelPattern();
    pattern.nodes.add(node);
    notifyListeners(pattern);
  }

  public void visitTriangle(Triangle element) {
    ModelNode node;
    if (!nodesMap.containsKey(element)) {
      node = new ModelNode();
      if (element.getClass() == TriangleImpl.class) {
        node.types.add("Triangle");
      }
      nodesMap.put(element, node);
    } else {
      node = nodesMap.get(element);
    }
    ModelPattern pattern = new ModelPattern();
    pattern.nodes.add(node);
    notifyListeners(pattern);
  }
}
```

7.1.3 Transformation

Before describing the generated code from a TrNet transformation, note that there are two important regions in a transformation: the input frontier and the output frontier.

The input frontier is the set of patterns that are directly connected to an input *External* operator. For instance, in figure 7.5, patterns $P1$ and $P3$ belong to the input frontier. Similarly, the output frontier is the set of patterns that are directly connected to an output *External* operator. A pattern can belong to the two sets without any unintended consequences. In the example, patterns $P5$, $P6$ and $P7$ belong to the output frontier.

7.1.3.1 Pattern Instances

As described in section 6.1.1, TrNet patterns are strongly typed in order to avoid comparisons between generic ModelPatterns as it would be too costly. This is implemented by generating a class for each pattern in a TrNet transformation. The class diagram is shown in figure 7.6. The Instance class, whose name depends on the id of the pattern that it represents, has one field for each node the pattern has. Each field is typed by ModelNode. For instance, for pattern $P4$ in the Figure 7.5 we would have a class similar to listing 7.2. Notice how the equality of two patterns of the same type is greatly simplified and runs linear to the number of nodes in that pattern.

Listing 7.2: Class generated from the pattern $Pat4$ in Figure 7.5

```
public class P4Instance{
  public ModelNode state;
  public ModelNode partition;
  public ModelNode activityPartition;

  @Override
```

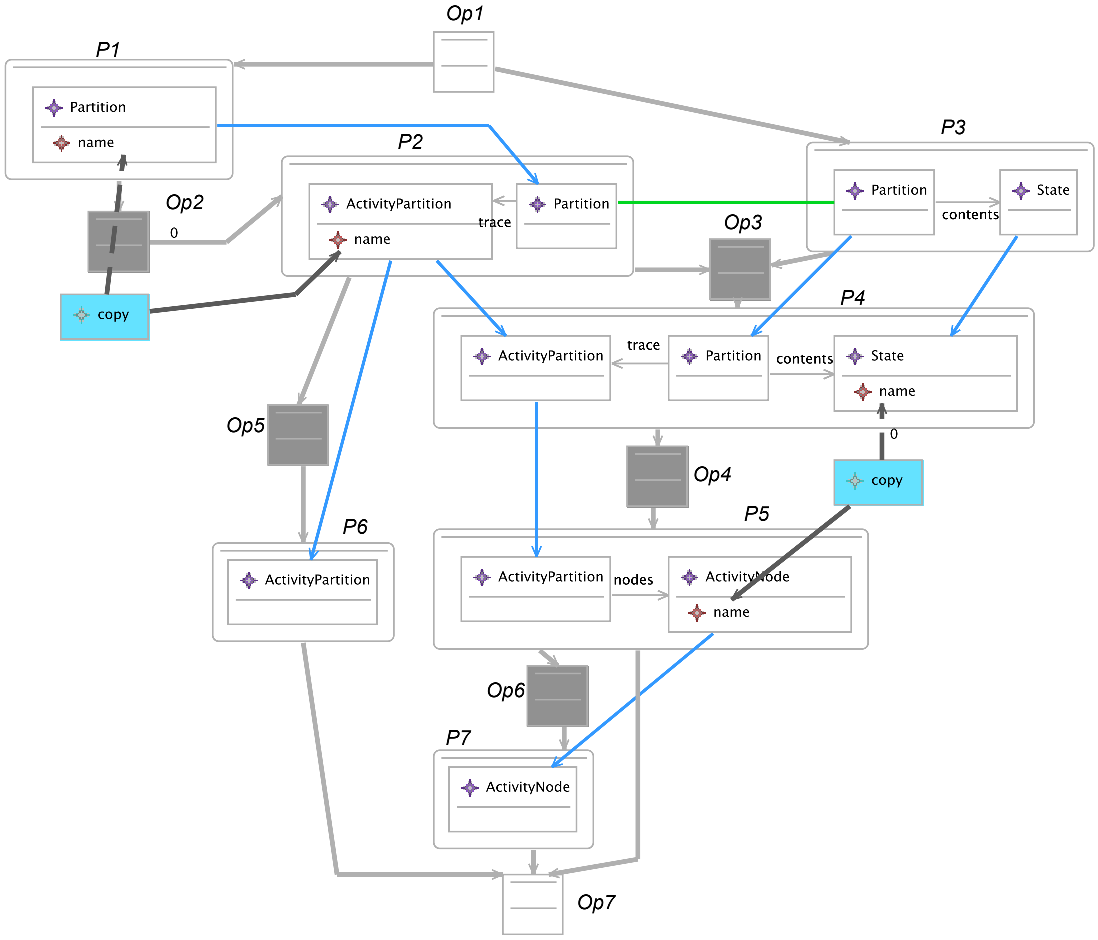


Figure 7.5: Transformation with input and output frontiers.

```
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + ((state==null) ? 0 : state.hashCode());
  result = prime * result + ((partition==null) ? 0 : partition.hashCode());
  result = prime * result + ((activityPartition==null) ? 0 :
                      activityPartition.hashCode());
  return result;
}

@Override
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;
  P4Instance other = (P4Instance) obj;
```

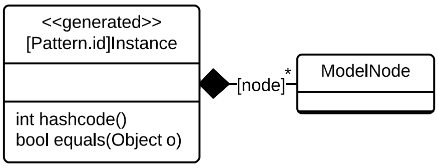


Figure 7.6: Class diagram for each class generated from a TrNet pattern.

```
    if (state==null) {
      if (other.state != null) {
        return false;
      }
    } else if (! state.equals(other.state)) {
      return false;
    }
    if (partition==null) {
      if (other.partition != null) {
        return false;
      }
    } else if (! partition.equals(other.partition)) {
      return false;
    }
    if (activityPartition==null) {
      if (other.activityPartition != null) {
        return false;
      }
    } else if (! activityPartition.equals(other.activityPartition)) {
      return false;
    }
    return true;
  }
}
```

Because we know exactly which pattern generated each class, we can always know the edges and, since they are not manipulated in the transformation as nodes are, there is no need to represent them explicitly in the generated code.

Patterns in the input or the output frontier, in addition to generating a pattern instance class like the one shown in figure 7.6, generate a publisher and listener interfaces for pattern instances as is described in the class diagram of figure 7.7. The reason for this will become clear when we explain the code for external operators.

7.1.3.2 External Operators

Each input external operator generates an ExternalInput class whose name depends on the id of the operator. In addition to being a ModelPattern Listener (remember that the

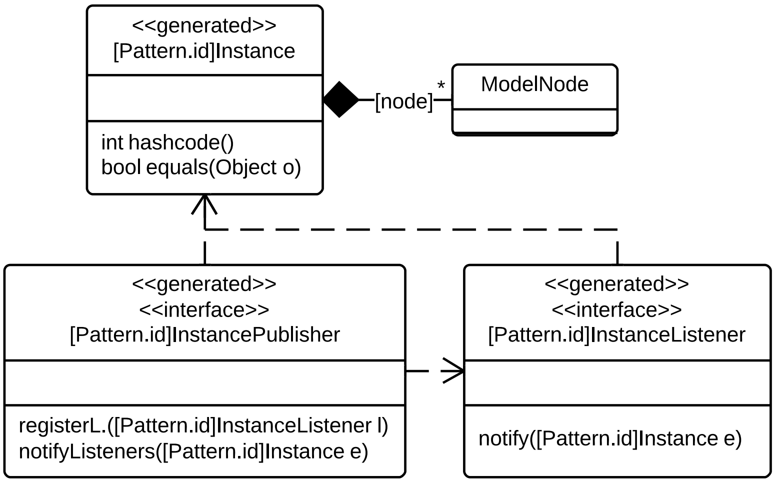

<<generated>>
[Pattern.id]Instance
int hashcode()
bool equals(Object o)
[node]
*
ModelNode
<<generated>>
<<interface>>
[Pattern.id]InstancePublisher
registerL.([Pattern.id]InstanceListener l)
notifyListeners([Pattern.id]Instance e)
<<generated>>
<<interface>>
[Pattern.id]InstanceListener
notify([Pattern.id]Instance e)

Figure 7.7: Class diagram for each class generated from a TrNet pattern that is in the input or output frontiers.

InputVisitor was a ModelPatternPublisher), this class implements Publisher for each pattern that is directly connected to the external operator. The class diagram is shown in figure 7.8. The ExternalInput class, when notified of a new ModelPattern element, attempts to convert that element to a specific pattern instance and then publishes it. Listing 7.3 shows the class generated from the input external operator of the transformation shown in Figure 7.5.

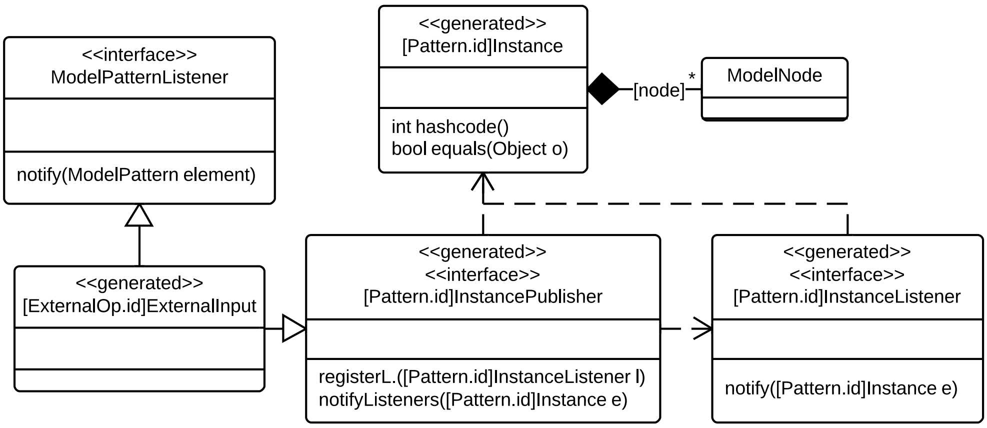

<<interface>>
ModelPatternListener
notify(ModelPattern element)
<<generated>>
[ExternalOp.id]ExternalInput
<<generated>>
[Pattern.id]Instance
int hashcode()
bool equals(Object o)
[node]
*
ModelNode
<<generated>>
<<interface>>
[Pattern.id]InstancePublisher
registerL.([Pattern.id]InstanceListener l)
notifyListeners([Pattern.id]Instance e)
<<generated>>
<<interface>>
[Pattern.id]InstanceListener
notify([Pattern.id]Instance e)

Figure 7.8: Class diagram for classes generated from the input external operators and patterns in the input frontier.

Listing 7.3: Input external operator code generated from the transformation in Figure 7.5

```
public class InputModelExternalInput implements ModelPatternListener,
    P3InstancePublisher, P1InstancePublisher {

  public void notify(ModelPattern element) {
    if (element.nodes.size() == 2 && element.edges.size() == 1) {
      ModelEdge elementEdge = element.edges.get(0);
      ModelNode elementNodeSource = elementEdge.source;
      ModelNode elementNodeTarget = elementEdge.target;
      if (elementEdge.name.equals("contents")
          && elementNodeSource.types.contains("Partition")
          && elementNodeTarget.types.contains("State")) {
        P3Instance patternInstance = new P3Instance();
        patternInstance.partition = elementNodeSource;
        patternInstance.state = elementNodeTarget;

        notifyListeners(patternInstance);
      }
    }
    if (element.nodes.size() == 1 && element.edges.size() == 0) {
      ModelNode elementNode = element.nodes.get(0);
      if (elementNode.types.contains("Partition")) {
        P1Instance patternInstance = new P1Instance();
        patternInstance.partition = elementNode;
        notifyListeners(patternInstance);
      }
    }
  }

  LinkedList<P3InstanceListener> listenersP3Instance;
  public void registerListener(P3InstanceListener listener) {
    ...
  }
  public void notifyListeners(P3Instance element) {
    ...
  }
  LinkedList<P1InstanceListener> listenersP1Instance;
  public void registerListener(P1InstanceListener listener) {
    ...
  }
  public void notifyListeners(P1Instance element) {
    ...
  }
  public InputModelExternalInput() {
    ...
  }
}
```

Each output external operator generates an ExternalOutput class, whose name depends on the id of the operator, that implements model pattern publisher (you can guess that the OutputVisitor will be a ModelPatternListener) and is a listener for each pattern directly connected to the external operator. The class diagram is shown in figure 7.9 and an example is shown in listing 7.4. When an ExternalOutput class is notified with a pattern element, it converts it to a ModelPattern instance and publishes it.

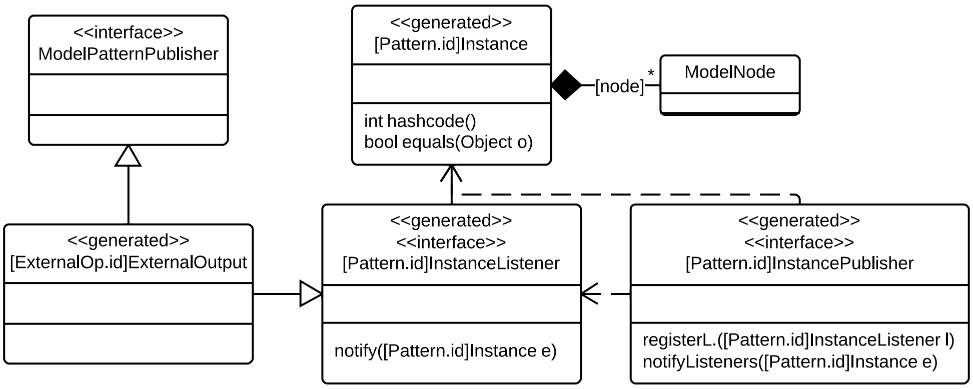

<<interface>>
ModelPatternPublisher
<<generated>>
[Pattern.id]Instance
int hashcode()
bool equals(Object o)
[node]
*
ModelNode
<<generated>>
[ExternalOp.id]ExternalOutput
<<generated>>
<<interface>>
[Pattern.id]InstanceListener
notify([Pattern.id]Instance e)
<<generated>>
<<interface>>
[Pattern.id]InstancePublisher
registerL.([Pattern.id]InstanceListener l)
notifyListeners([Pattern.id]Instance e)

Figure 7.9: Class diagram for classes generated from the output external operators and patterns in the output frontier.

Listing 7.4: Output external operator code generated from the transformation in Figure 7.5

```
public class OutputModelExternalOutput implements ModelPatternPublisher,
    P7InstanceListener, P5InstanceListener, P6InstanceListener {

  LinkedList<ModelPatternListener> listeners;

  public OutputModelExternalOutput() {
    listeners = new LinkedList<ModelPatternListener>();
  }

  public void registerListener(ModelPatternListener listener) {
    listeners.add(listener);
  }

  public void notifyListeners(ModelPattern element) {
    for (ModelPatternListener listener : listeners) {
      listener.notify(element);
    }
  }

  public void notify(P7Instance element) {
    ModelPattern genericPattern = new ModelPattern();
    genericPattern.nodes.add(element.activityNode);
    notifyListeners(genericPattern);
  }

  public void notify(P5Instance element) {
    ModelPattern genericPattern = new ModelPattern();
    genericPattern.nodes.add(element.activityNode);
    genericPattern.nodes.add(element.activityPartition);
    {
      ModelEdge edge = new ModelEdge();
      edge.name = "nodes";
      edge.source = element.activityPartition;
      edge.target = element.activityNode;
      genericPattern.edges.add(edge);
    }
    notifyListeners(genericPattern);
  }

  public void notify(P6Instance element) {
    ModelPattern genericPattern = new ModelPattern();
    genericPattern.nodes.add(element.activityPartition);
    notifyListeners(genericPattern);
  }
}
```

7.1.3.3 Transformation Class

The whole transformation behaviour is implemented in one class shown in figure 7.10. In order to implement the communication between the class and the decomposition and composition processes (implemented by the InputVisitor and OutputVisitor classes, respectively), the transformation class implements listener for each pattern in the input frontier and publisher for each pattern in the output frontier.

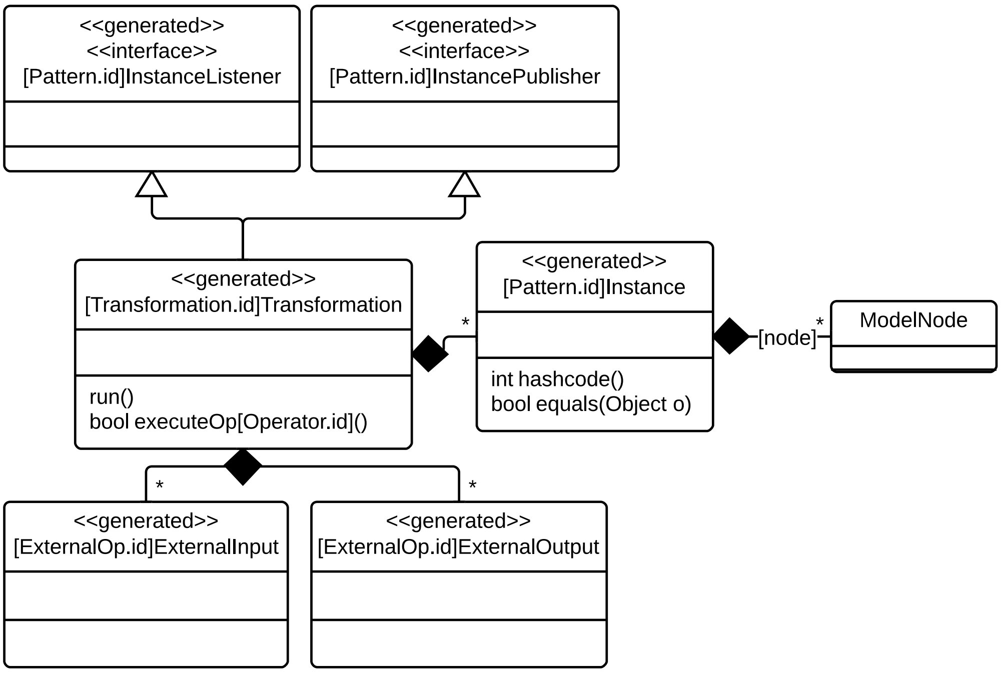


Figure 7.10: Class diagram for the class that implement the behaviour of the transformation along with the classes it relates to.

For each external operator, the class has a field to reference the class representing that operator.

For each pattern in the transformation, the class has the following fields: an ArrayList, a HashSet and a variable number of HashMaps with ModelNode as keys and the pattern class as values. These fields store the pattern instances during the transformation execution. They represent, at any given time, the transformation configuration. The ArrayList is used to iterate the instances, the HashSet is used to avoid duplicates and the variable number of HashMaps act as indexes that allow a quick retrieval of all the pattern instances with a given ModelNode. This is most useful when performing combination where there are Same restrictions.

For each operator in the transformation, a Boolean method is generated in the transformation class that implements that operator's behaviour. This method, when executed, will change the configuration of the transformation by reading from some pattern fields, create new pattern instances, process attributes and add the those instances to other pattern fields. The specific behaviour depends on the operands, results, restrictions, conditions and actions surrounding that operator in the transformation. The method returns true if it changed the configuration of the transformation and false otherwise.

As an example, consider listing 7.5 where we show an excerpt of the transformation class generated for the transformation of figure 7.5.

Listing 7.5: Transformation code generated from the transformation in Figure 7.5

```
public class SampleSimpleTransformationTransformation implements
    P1InstanceListener, P3InstanceListener, P6InstancePublisher,
    P5InstancePublisher, P7InstancePublisher {

  InputModelExternalInput inputInputModel;
  OutputModelExternalOutput outputOutputModel;
  ArrayList<P5Instance> p5Array;
  HashSet<P5Instance> p5Set;
  ArrayList<P6Instance> p6Array;
  HashSet<P6Instance> p6Set;
  ArrayList<P7Instance> p7Array;
  HashSet<P7Instance> p7Set;
  ArrayList<P4Instance> p4Array;
  HashSet<P4Instance> p4Set;
  ArrayList<P2Instance> p2Array;
  HashSet<P2Instance> p2Set;
  ArrayList<P3Instance> p3Array;
  HashSet<P3Instance> p3Set;
  HashMap<ModelNode, LinkedList<P3Instance>> partitionInP3Hash;
  ArrayList<P1Instance> p1Array;
  HashSet<P1Instance> p1Set;

  public SampleSimpleTransformationTransformation() {
    ...
    listenersP6Instance = new LinkedList<P6InstanceListener>();
    listenersP5Instance = new LinkedList<P5InstanceListener>();
    listenersP7Instance = new LinkedList<P7InstanceListener>();

    inputInputModel = new InputModelExternalInput();
    inputInputModel.registerListener((P3InstanceListener) this);
    inputInputModel.registerListener((P1InstanceListener) this);

    outputOutputModel = new OutputModelExternalOutput();
    this.registerListener((P7InstanceListener) outputOutputModel);
    this.registerListener((P5InstanceListener) outputOutputModel);
    this.registerListener((P6InstanceListener) outputOutputModel);
  }

  public void run() {
    executeOperatorInputModel();
    executeOperatorO1();
    executeOperatorO3();
    executeOperatorO2();
    executeOperatorO4();
    executeOperatorO5();
    executeOperatorOutputModel();
  }

  public void notify(P1Instance element) {
    if (p1Set.add(element)) {
      p1Array.add(element);
    }
  }

  public void notify(P3Instance element) {
    if (p3Set.add(element)) {
      p3Array.add(element);
      {
        if (!partitionInP3Hash.containsKey(element.partition)) {
          partitionInP3Hash.put(element.partition,
              new LinkedList<P3Instance>());
        }
        partitionInP3Hash.get(element.partition).add(element);
      }
    }
  }

  LinkedList<P6InstanceListener> listenersP6Instance;
  public void registerListener(P6InstanceListener listener) {
    ...
  }
  public void notifyListeners(P6Instance element) {
    ...
  }

  LinkedList<P5InstanceListener> listenersP5Instance;
  public void registerListener(P5InstanceListener listener) {
    ...
  }
  public void notifyListeners(P5Instance element) {
    ...
  }

  LinkedList<P7InstanceListener> listenersP7Instance;
  public void registerListener(P7InstanceListener listener) {
    ...
  }
  public void notifyListeners(P7Instance element) {
    ...
  }

  boolean executeOperatorO3() {
    boolean operatorHasExecuted = false;
    int sourcePatternSize = p2Array.size();
    for (int i = 0; i < sourcePatternSize; i++) {
      P2Instance sourcePatternInstance = p2Array.get(i);
      {
        P6Instance targetInstance = new P6Instance();
        {
          targetInstance.activityPartition =
              sourcePatternInstance.activityPartition;
        }
        if (p6Set.add(targetInstance)) {
          p6Array.add(targetInstance);
          operatorHasExecuted = true;
        }
      }
    }
    return operatorHasExecuted;
  }
  boolean executeOperatorOutputModel() {
    boolean operatorHasExecuted = false;
    {
      int sourcePatternSize = p7Array.size();
      for (int i = 0; i < sourcePatternSize; i++) {
        P7Instance sourcePatternInstance = p7Array.get(i);
        notifyListeners(sourcePatternInstance);
      }
    }
    {
      int sourcePatternSize = p5Array.size();
      for (int i = 0; i < sourcePatternSize; i++) {
        P5Instance sourcePatternInstance = p5Array.get(i);
        notifyListeners(sourcePatternInstance);
      }
    }
    {
      int sourcePatternSize = p6Array.size();
      for (int i = 0; i < sourcePatternSize; i++) {
        P6Instance sourcePatternInstance = p6Array.get(i);
        notifyListeners(sourcePatternInstance);
      }
    }
    return operatorHasExecuted;
  }
  boolean executeOperatorO5() {
    ...
  }
  boolean executeOperatorO4() {
    boolean operatorHasExecuted = false;
    int sourcePatternSize = p4Array.size();
    for (int i = 0; i < sourcePatternSize; i++) {
      P4Instance sourcePatternInstance = p4Array.get(i);
      {
        P5Instance targetInstance = new P5Instance();
        {
          ModelNode node = new ModelNode();
          node.attributes.put("name", generic.utils.Copy
              .copy(sourcePatternInstance.state.attributes
                  .get("name")));
          node.types.add("ActivityNode");
          targetInstance.activityNode = node;
        }
        {
          targetInstance.activityPartition =
            sourcePatternInstance.activityPartition;
        }
        if (p5Set.add(targetInstance)) {
          p5Array.add(targetInstance);
          operatorHasExecuted = true;
        }
      }
    }
    return operatorHasExecuted;
  }
  boolean executeOperatorO2() {
    ...
  }
  boolean executeOperatorO1() {
    ...
  }
  boolean executeOperatorInputModel() {
    return true;
  }
  public void registerInputModelPublisher(ModelPatternPublisher publisher) {
    ...
  }
  public void registerOutputModelListener(ModelPatternListener listener) {
    ...
  }
}
```

The transformation class' constructor is responsible for allocating the necessary memory for each pattern's fields and create and register an instance of each external operator as publishers or listeners. See Section 8.2 for more details about how to estimate the necessary memory. The transformation class registers itself as listener in each class generated from the input external operators and registers each class generated from the output external operators as listeners.

In each transformation class there is a single run() method which contains all the calls, in sequence, to the operators' methods, implementing the transformation execution. More about this method in section 9.1.

7.1.4 Composition

The composition process is responsible for performing the exact opposite of the decomposition process: from atomic model elements, i.e., ModelPattern instances, it builds and stores the output model. But it is more complex since it has to keep track of every element produced by the transformation and store it in the appropriate sets. Then is has to aggregate elements as efficiently as possible to build the output model.

Similarly to the decomposition process uses generated classes from a metamodel by EMF to create a model in memory and then store it.

Most of the composition process is implemented in one class: the OutputVisitor class as figure 7.11 shows.

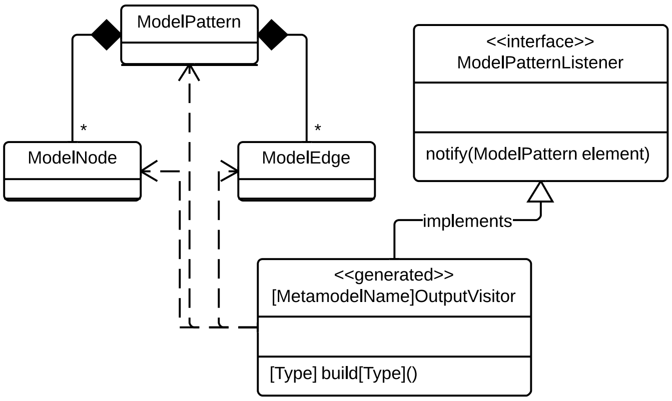


Figure 7.11: Composition process classes.

Consider listing 7.6 as an excerpt of the generated code from the metamodel show in figure 7.4.

Listing 7.6: Output visitor code generated from the metamodel shown in Figure 7.4

```java
public class ShapesOutputVisitor implements ModelPatternListener {

  HashSet<ModelNode> SquareSet;
  HashMap<ModelNode, Square> modelNode2Square;
  HashSet<ModelNode> CircleSet;
  HashMap<ModelNode, Circle> modelNode2Circle;
  HashSet<ModelNode> PentagonSet;
  HashMap<ModelNode, Pentagon> modelNode2Pentagon;
  HashSet<ModelNode> TriangleSet;
  HashMap<ModelNode, Triangle> modelNode2Triangle;
  HashMap<ModelNode, LinkedList<ModelPattern>> SquareSquaresSquareMap;
  HashMap<ModelNode, LinkedList<ModelPattern>> SquarePentagonsPentagonMap;
  HashMap<ModelNode, LinkedList<ModelPattern>> SquareTrianglesTriangleMap;
  HashMap<ModelNode, LinkedList<ModelPattern>> CircleSquaresSquareMap;
  HashMap<ModelNode, LinkedList<ModelPattern>> CirclePentagonsPentagonMap;

  public ShapesOutputVisitor() {
    ...
  }

  public void store(Circle element, String path) {
    Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
    Map<String, Object> m = reg.getExtensionToFactoryMap();
    m.put("xmi", new XMIResourceFactoryImpl());

    ResourceSet resSet = new ResourceSetImpl();
    Resource resource = resSet.createResource(URI.createURI(path));

    resource.getContents().add(element);

    try {
      resource.save(Collections.EMPTY_MAP);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void notify(ModelPattern element) {
    if (element.nodes.size() == 1 && element.edges.size() == 0) {
      {
        ModelNode node = element.nodes.get(0);
        if (node.types.contains("Square") || false) {
          node.types.add("Square");
          SquareSet.add(node);
        }
      }
      {
        ModelNode node = element.nodes.get(0);
        if (node.types.contains("Circle") || false) {
          node.types.add("Circle");
          CircleSet.add(node);
        }
      }
      {
        ModelNode node = element.nodes.get(0);
        if (node.types.contains("Pentagon") || false) {
          node.types.add("Pentagon");
          PentagonSet.add(node);
        }
      }
      {
        ModelNode node = element.nodes.get(0);
        if (node.types.contains("Triangle") || false) {
          node.types.add("Triangle");
          TriangleSet.add(node);
        }
      }
    } else if (element.nodes.size() == 2 && element.edges.size() == 1) {
      {
        ModelEdge edge = element.edges.get(0);
        if ((edge.source.types.contains("Square") || false)
            && edge.name.equals("squares")
            && (false || edge.target.types.contains("Square"))) {
          if (!SquareSquaresSquareMap.containsKey(edge.source)) {
            SquareSquaresSquareMap.put(edge.source,
                new LinkedList<ModelPattern>());
          }
          SquareSquaresSquareMap.get(edge.source).add(element);
        }
      }
      {
        ModelEdge edge = element.edges.get(0);
        if ((edge.source.types.contains("Square") || false)
            && edge.name.equals("pentagons")
            && (false || edge.target.types.contains("Pentagon"))) {
          if (!SquarePentagonsPentagonMap.containsKey(edge.source)) {
            SquarePentagonsPentagonMap.put(edge.source,
                new LinkedList<ModelPattern>());
          }
          SquarePentagonsPentagonMap.get(edge.source).add(element);
        }
      }
      {
        ModelEdge edge = element.edges.get(0);
        if ((edge.source.types.contains("Square") || false)
            && edge.name.equals("triangles")
            && (false || edge.target.types.contains("Triangle"))) {
          if (!SquareTrianglesTriangleMap.containsKey(edge.source)) {
            SquareTrianglesTriangleMap.put(edge.source,
                new LinkedList<ModelPattern>());
          }
          SquareTrianglesTriangleMap.get(edge.source).add(element);
        }
      }
      {
        ModelEdge edge = element.edges.get(0);
        if ((edge.source.types.contains("Circle") || false)
            && edge.name.equals("squares")
            && (false || edge.target.types.contains("Square"))) {
          if (!CircleSquaresSquareMap.containsKey(edge.source)) {
            CircleSquaresSquareMap.put(edge.source,
                new LinkedList<ModelPattern>());
          }
          CircleSquaresSquareMap.get(edge.source).add(element);
        }
      }
      {
        ModelEdge edge = element.edges.get(0);
        if ((edge.source.types.contains("Circle") || false)
            && edge.name.equals("pentagons")
            && (false || edge.target.types.contains("Pentagon"))) {
          if (!CirclePentagonsPentagonMap.containsKey(edge.source)) {
            CirclePentagonsPentagonMap.put(edge.source,
                new LinkedList<ModelPattern>());
          }
          CirclePentagonsPentagonMap.get(edge.source).add(element);
        }
      }
    } else {
      throw new RuntimeException("Unexpected␣pattern␣found.");
    }
  }

  public Circle buildModel() {
    return buildCircle();
  }

  Circle buildCircle() {
    for (ModelNode node : CircleSet) {
      Circle result = modelNode2Circle.get(node);

      if (result == null) {
        result = ShapesFactory.eINSTANCE.createCircle();
        modelNode2Circle.put(node, result); // coloca ja o elemento no
                          // mapa por causa das
                          // futuras referencias com
                          // target neste elemento.
        createAssocCirclePentagons(result, node);
        createAssocCircleSquares(result, node);
      }

      return result;
    }

    return null;
  }

  void createAssocSquareSquares(Square modelElement, ModelNode node) {
    LinkedList<ModelPattern> patternAssocs = SquareSquaresSquareMap
        .get(node);
    if (patternAssocs == null) {
      return;
    }
    for (ModelPattern patternAssoc : patternAssocs) {
      ModelNode targetNode = patternAssoc.edges.get(0).target;

      if (targetNode.types.contains("Square")) {
        if (SquareSet.contains(targetNode)) {

          Square targetElement = modelNode2Square.get(targetNode);

          if (targetElement == null) {
            targetElement = ShapesFactory.eINSTANCE.createSquare();
            modelNode2Square.put(targetNode, targetElement);
          }
          createNonContainmentAssocSquarePentagons(targetElement,
              targetNode);
          createAssocSquareTriangles(targetElement, targetNode);
          createAssocSquareSquares(targetElement, targetNode);

          modelElement.getSquares().add(targetElement);
        }
      }
    }
  }

  void createAssocSquareTriangles(Square modelElement, ModelNode node) {
    LinkedList<ModelPattern> patternAssocs = SquareTrianglesTriangleMap
        .get(node);
    if (patternAssocs == null) {
      return;
    }
    for (ModelPattern patternAssoc : patternAssocs) {
      ModelNode targetNode = patternAssoc.edges.get(0).target;

      if (targetNode.types.contains("Triangle")) {
        if (TriangleSet.contains(targetNode)) {

          Triangle targetElement = modelNode2Triangle.get(targetNode);

          if (targetElement == null) {
            targetElement = ShapesFactory.eINSTANCE
                .createTriangle();
            modelNode2Triangle.put(targetNode, targetElement);
          }

          modelElement.getTriangles().add(targetElement);
        }
      }
    }
  }

  void createAssocCircleSquares(Circle modelElement, ModelNode node) {
    LinkedList<ModelPattern> patternAssocs = CircleSquaresSquareMap
        .get(node);
    if (patternAssocs == null) {
      return;
    }
    for (ModelPattern patternAssoc : patternAssocs) {
      ModelNode targetNode = patternAssoc.edges.get(0).target;

      if (targetNode.types.contains("Square")) {
        if (SquareSet.contains(targetNode)) {

          Square targetElement = modelNode2Square.get(targetNode);

          if (targetElement == null) {
            targetElement = ShapesFactory.eINSTANCE.createSquare();
            modelNode2Square.put(targetNode, targetElement);
          }
          createNonContainmentAssocSquarePentagons(targetElement,
              targetNode);
          createAssocSquareTriangles(targetElement, targetNode);
          createAssocSquareSquares(targetElement, targetNode);

          modelElement.getSquares().add(targetElement);
        }
      }
    }
  }

  void createAssocCirclePentagons(Circle modelElement, ModelNode node) {
    LinkedList<ModelPattern> patternAssocs = CirclePentagonsPentagonMap
        .get(node);
    if (patternAssocs == null) {
      return;
    }
    for (ModelPattern patternAssoc : patternAssocs) {
      ModelNode targetNode = patternAssoc.edges.get(0).target;

      if (targetNode.types.contains("Pentagon")) {
        if (PentagonSet.contains(targetNode)) {

          Pentagon targetElement = modelNode2Pentagon.get(targetNode);

          if (targetElement == null) {
            targetElement = ShapesFactory.eINSTANCE
                .createPentagon();
            modelNode2Pentagon.put(targetNode, targetElement);
          }

          modelElement.getPentagons().add(targetElement);
        }
      }
    }
  }

  void createNonContainmentAssocSquarePentagons(Square modelElement,
      ModelNode node) {
    LinkedList<ModelPattern> patternAssocs = SquarePentagonsPentagonMap
        .get(node);
    if (patternAssocs == null) {
      return;
    }
    for (ModelPattern patternAssoc : patternAssocs) {
      ModelNode targetNode = patternAssoc.edges.get(0).target;

      if (targetNode.types.contains("Pentagon")) {
        if (PentagonSet.contains(targetNode)) {

          Pentagon targetElement = modelNode2Pentagon.get(targetNode);

          if (targetElement == null) {
            targetElement = ShapesFactory.eINSTANCE
                .createPentagon();
            modelNode2Pentagon.put(targetNode, targetElement);
          }

          modelElement.getPentagons().add(targetElement);
        }
      }
    }
  }
}
```

Being a ModelPatternListener, when notified of an atomic element, the output visitor class inspects it and identifies its type. Once identified, the element is stored. The storage mechanism used depends on the kind of atomic element received. If it is a individual element, it is stored in a set with all elements of the same type. Notice that the generated code stores the same ModelNode not only in the set of its type, but also in the set of each super type. The example does not show that because there is no hierarchy in the metamodel.

During the transformation execution, the output visitor will accumulate model patterns and once the transformation has finished executing, it can aggregate all those elements into a model. It does this by starting at the root element of the model, and then navigating each relation (information that hard coded from the metamodel) it collects all the remaining elements. The assumption that every model has a root element is a very common one since a model that has two root elements, when stored into xmi, does not even make a valid xml document. Be that as it may, almost all models have an aggregating class that contains all others.

In the example listing 7.6, the buildCircle method start the whole composition process. It uses EMF generated code to build a real Circle instance and then goes recursively building each sub model trying every association encoded in the metamodel. It uses maps to keep track of what model nodes gave origin to what output model elements in order to avoid creating two instances for the same ModelNode.

For a containment association, it obtains all the ModelPattern elements that correspond to that association and that start at the source ModelNode of the association. From all the associations found, it retains only those that have a well defined target, i.e., the target ModelNode of the association must also exist in its type set, or in any of its subtypes' set. And this process is repeated for the target ModelNodes of those associations.

For a Non-containment association, the algorithm is almost the same, except that it does not repeat itself for the targets of the association, since this would result in an infinite loop for some models and it would be redundant because if the target ModelNode element is contained in some parent, then eventually it will be visited and its information filled. Otherwise, if the element is not contained in any element that is already in the output model, then it will not appear in the final model, which is the expected and correct behaviour.

7.1.5 Fitting All Together

In summary, in a typical transformation we have one input visitor class, one transformation class and one output visitor class, one external input class, an external output class and several classes, each representing a specific pattern in the transformation. The input visitor class processes the input model into several ModelPattern instances that are delivered to the external input class. The external input class converts these atomic ModelPatterns into specific pattern instances which are then delivered to the transformation, by placing them in the corresponding sets. At this point the transformation runs, executing all the operators. The final operators to be executed in the transformation are the outputs which deliver the specific pattern instances to the external output class. The external output class converts those instances to generic ModelPattern instances and passes them to the output model visitor. The output model stores all the instances and, when given the instruction to build the output model, it aggregates them into the output model.

Figure 7.12 summarizes the overall process and listing 7.7 shows a sample usage in Java.

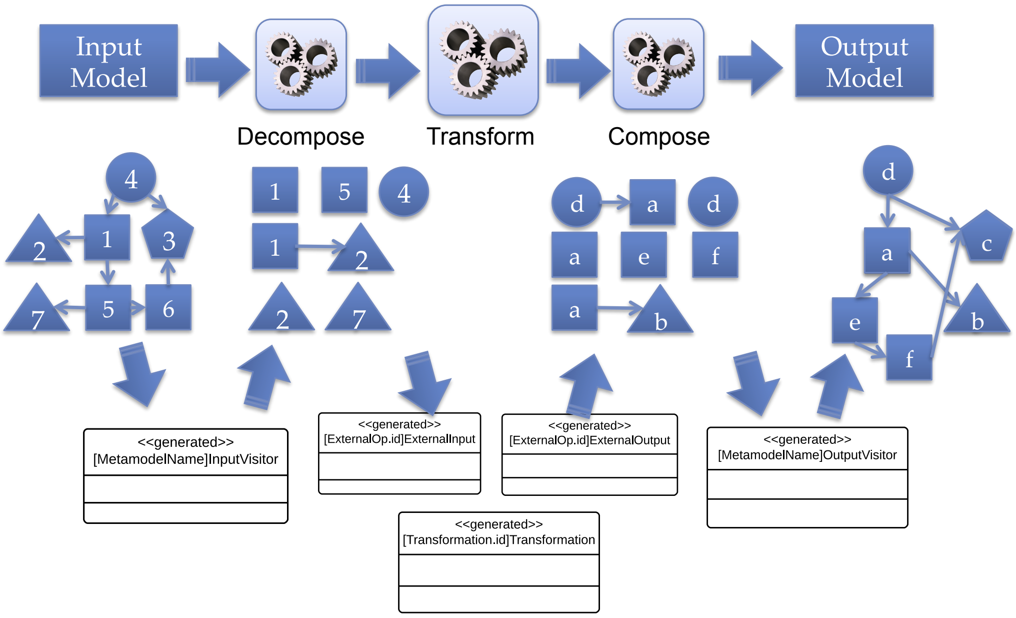

Input Model
Decompose
Transform
Compose
Output Model
4
2
1
3
7
5
6
1
5
4
1
2
2
7
d
a
d
a
e
f
a
b
d
a
c
e
b
f
<<generated>>
[MetamodelName]InputVisitor
<<generated>>
[ExternalOp.id]ExternalInput
<<generated>>
[ExternalOp.id]ExternalOutput
<<generated>>
[MetamodelName]OutputVisitor
<<generated>>
[Transformation.id]Transformation

Figure 7.12: A typical TrNet transformation process.

Listing 7.7: Typical code to run a TrNet transformation

```
void RunTransformation() {
  Uml11InputVisitor inputv = new Uml11InputVisitor();
  Uml21OutputVisitor outputv = new Uml11InputVisitor();

  ActivityMigrationTransformation transformation =
        new ActivityMigrationTransformation();
  transformation.registerOriginalInputPublisher(inputv);
  transformation.registerEvolvedOutputListener(outputv);

  ActivityGraph s = inputv.load("InputModel.xmi");
  // Decompose
  inputv.visitActivityGraph(s);
  // Transform
  transformation.run();
  // Compose
  Package d = outputv.buildModel();

  outputv.store(d, "OutputModel.xmi");
}
```

Note that the while the ModelPattern allows one to represent any arbitrary model, the instances handled by the input and output visitor are atomic. They either represent an atomic element, with only one ModelNode; or they represent an association, with two ModelNodes and a ModelEdge. This is a design decision that has the advantage of pushing the complexity onto the transformation and not to the model management processes. It is much faster to use the transformation to build complex elements from simpler ones than having the input visitor search for complex patterns in the input model. That is the performance penalty we are trying to avoid!

The main advantage of this approach is to guarantee that the transformation is independent of the model management framework. Using a common representation for patterns and the publisher listener pattern allows for a good degree of decoupling between the three main processes. If, for instance, our models were stored in a database, all we have to do is change the input and output visitors. The transformation remains the same.

The fact that the input and output visitors do not depend on the transformation contents allows one to generate the visitor for purposes other than feeding the transformation.

The code generation was performed using Epsilon Template Language (ETL) **[Rose:2010:MME:1875847.187**

7.2 DSLTrans Compilation

The DSLTrans Compilation is performed rule by rule, layer by layer.

As an example consider the DSLTrans transformation and the corresponding TrNet transformation shown side by side in Figure 7.13.

Each FilePort is translated into an input external operator.

Each Layer is translated into an output external operator (since in DSLTrans, each Layer outputs a model). In addition, the layer's rules are compiled.

The compilation of the rule is divided into three steps: compile all the rule's match models, compile the apply model and connect all together.

Each match model is translated into a Pattern containing NodePatterns and EdgePatterns. NodePatterns are created for each MatchClass in the match model and EdgePatterns are created for each MatchAssociation. MatchAttributes are compiled to AttributePattern inside the corresponding NodePatterns.

Similarly to the match model, each apply model is translated into a Pattern with its contents being the result of compiling the apply elements of the apply model. An ApplyClass is translated into a NodePattern and an ApplyAssociation is translated into an EdgePattern. In addition, depending on the existence of the trace attribute in the ApplyClass, a new NodePattern and a EdgePattern can be connected to the ApplyClass' NodePattern to represent the trace being created.

To connect the patterns created for each match model and the apply model of a rule it is necessary to: create the necessary Combinators that have as operands the pattern that were created from each match model; set as a result the pattern that was created from the apply model; add more elements to the match models' patterns to match any

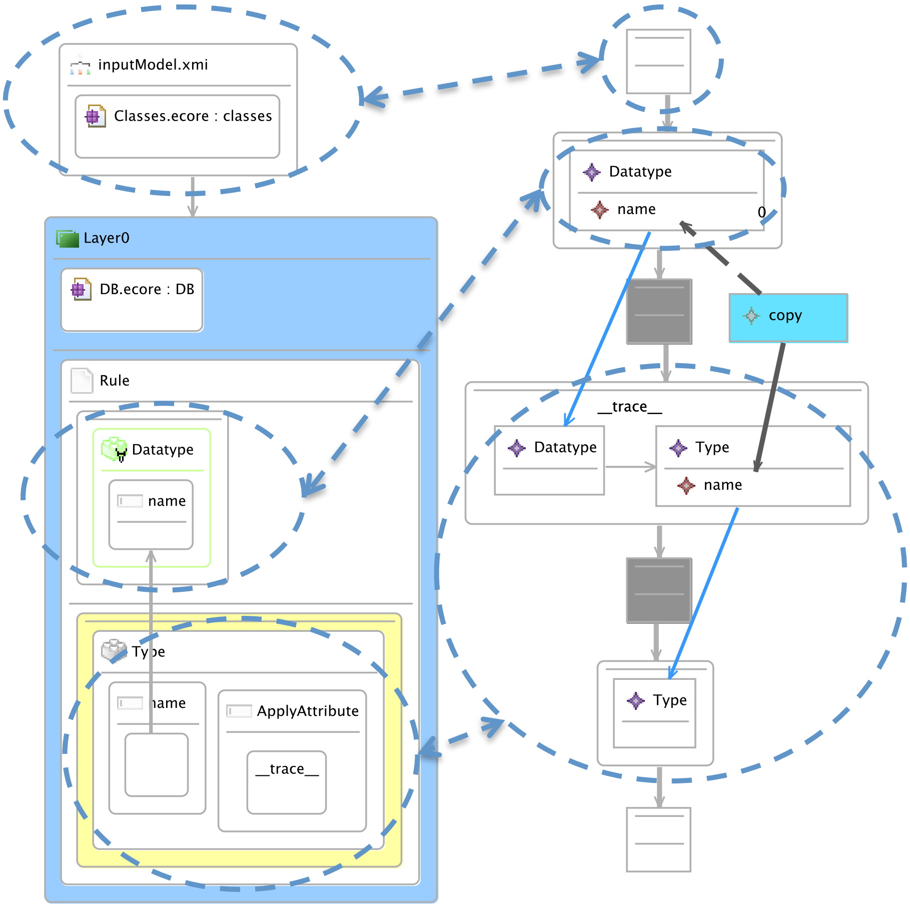


Figure 7.13: A DSLTrans transformation and its corresponding TrNet transformation.

existing backward restrictions (matching traceability links); add the necessary application conditions to the combinators to complete the possibly existing attribute matching in the match models; link the patterns created from each match model to the rest of the TrNet transformation and do the same for the pattern created form the apply model.

For the sake of brevity, assume that there is only one match model in the rule and one apply model, which is by far the most common case. The pattern created from the match model is connected to a new combinator as an operand and the pattern created from the apply model is connected as a result of that operator. The execution of this new combinator performs the application of the rule.

In that same combinator, application conditions are added for each existing match attribute condition inside any match class of the match model. This step guarantees that the combinator will only be applied to a candidate element if all attribute conditions are true, which is the expected behaviour os a DSLTrans rule.

Next, any existing backward conditions between a match class and an apply class are compiled by adding a NodePattern to the match model pattern and a NodePattern to the apply model pattern. These two NodePatterns correspond to the match class involved in the backward restriction and they are connected by a trace relation to the apply class node pattern in both patterns and with a keep element between them. This step guarantees that the traceability link represented by the backward link has to be matched before the rule can be applied.

Once all the previous tasks are performed, we have a pattern that corresponds to the match model, a pattern that corresponds to the apply model and a combinator that performs the application of the rule by connecting the match model pattern to the apply model pattern. It is still necessary to integrate the match model pattern and the apply model pattern in the rest of the transformation.

To connect the match model pattern to the rest of the transformation it is necessary to generate its construction through several combinations of atomic pattern elements. This construction ensures that, when the transformation is executing, the match pattern is gradually built from smaller patterns. In order to achieve this, an atomic element is selected in the match model pattern and two new patterns are created: one with the atomic element and the other with the match model pattern without the atomic element. The two patterns are operands of a new combinator that outputs to the initial match model pattern. This process is now repeated recursively until all patterns are atomic. then, all the atomic patterns are either connected to the input external operator, or, in they it is a trace, to all the patterns where that trace is being created. Those patterns were the result of compiling the previous layers' rules.

The apply model pattern undergoes a similar process to generate its destruction into several atomic patterns. This process however, is much more simple: a new atomic pattern is created for each atomic pattern in the apply model pattern. The apply model pattern is then connected to each atomic pattern by means of a new combinator (created for each pattern). The new atomic patterns are either connected to the output external operator corresponding to the layer, or, in case if it is a trace atomic pattern, to other pattern, pertaining to other rule's match model.

Figure 7.14 shows an excerpt of the TrNet transformation generate for the DSLTrans rule at the left.

Observing the inherent complexity of the compilation to TrNet, we quickly grasp the amount of work that is performed entirely automatically by the DSLTrans engine compared to that same work performed by a TrNet. That difference reflects the transformation complexity: TrNet's transformations are really complex and big when compared to the same transformation expressed in DSLTrans. As an example, for the benchmark that we used (see more details in chapter 10), the DSLTrans transformation has 78 rules while the resulting TrNet transformation has 467 patterns and 750 operators. Off course we developed optimizations that can lower this value to 355 patterns and 288 operators (see section 9.6) but still it shows how much a transformation developer can win by using a high-level language like DSLTrans.

On the performance perspective, it is possible to see that, while DSLTrans, during an execution, searches several times the input model for patterns, the compiler hard codes

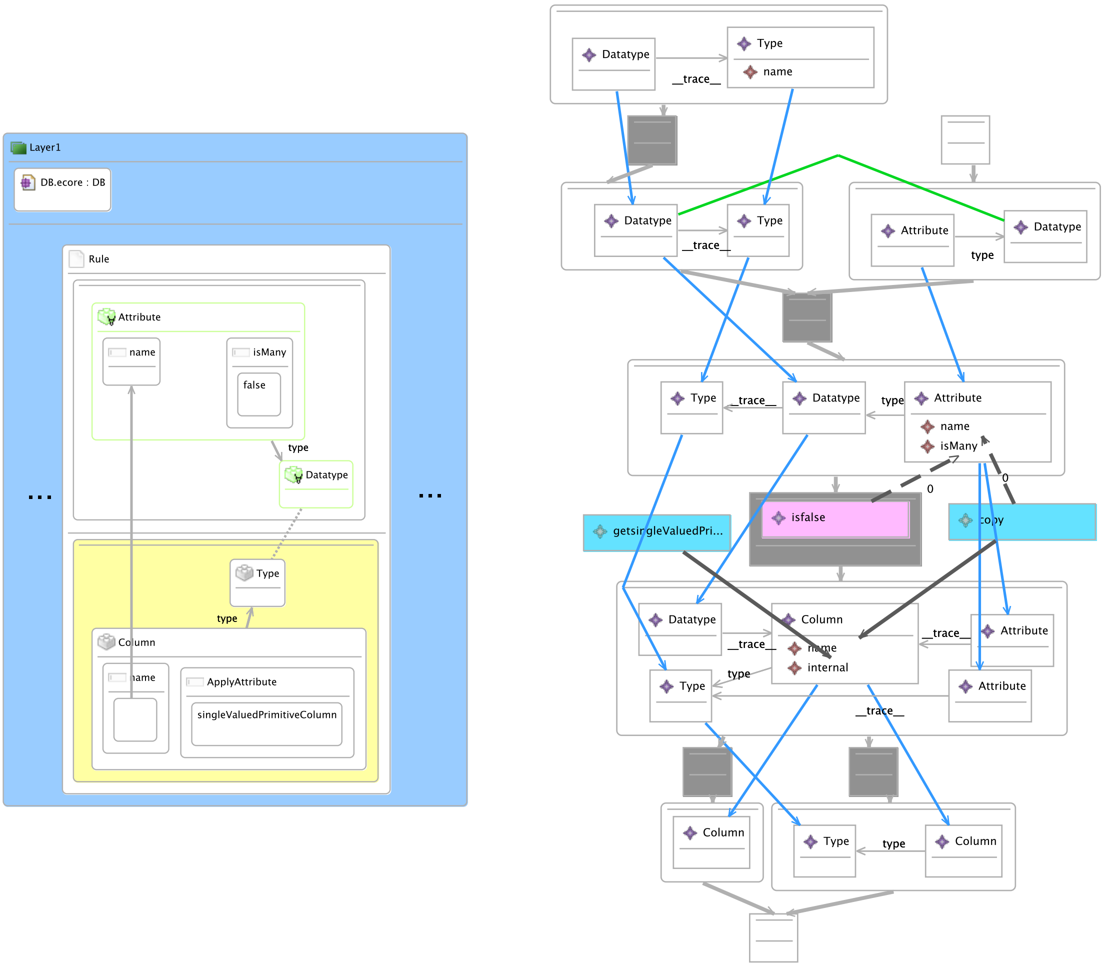


Figure 7.14: A DSLTrans transformation and its corresponding TrNet transformation (continuation).

this behaviour by creating the necessary elements to collect the proper atomic elements and combine them to achieve the pattern needed for the rule to be applied. The major advantage of this is that the input model is visited only once, at the beginning of the transformation. From that point, the operators are executed to propagate those elements throughout the entire transformation.

8

Analysis

During a compilation of a transformation written in TrNet there are a lot of decisions that affect the performance of the resulting code. As a consequence, we had to develop analysis techniques to help to generate code that is as efficient as possible. This analysis allows for both performance and memory consumption estimation. The kind of analysis performed here is very similar to what database management systems do when executing a query [**Silberschatz:2010uw**]. Off course a TrNet transformation can be several times more complex than a query and there are some peculiarities due to our approach.

As an example, consider the excerpt of a transformation shown in figure 8.1 and the code in algorithm 14 that implements the behaviour of combinator $O1$. In Algorithm 14, the instances of pattern $Pat1$ are iterated one by one. For each instance, an access to an hash map immediately returns all the interesting instances of pattern $Pat2$ (note the same restriction). Having an hash map greatly reduces the cost of the operator execution but there are, perhaps better, alternatives. For instance, we could start by iterating all the instances of pattern $Pat2$ and using an hash map to access the corresponding instances of $Pat1$ as is shown in Algorithm 15. How can we know which approach is the cheapest one?

Algorithm 14 Algorithm showing the behaviour of Combinator O1.

```
function ExecuteO1
    for (p, ap) ∈ Pat1 do
        for (p′ , s) ∈ GetPat2FromPartition(p) do
            ...
        end for
    end for
end function
```

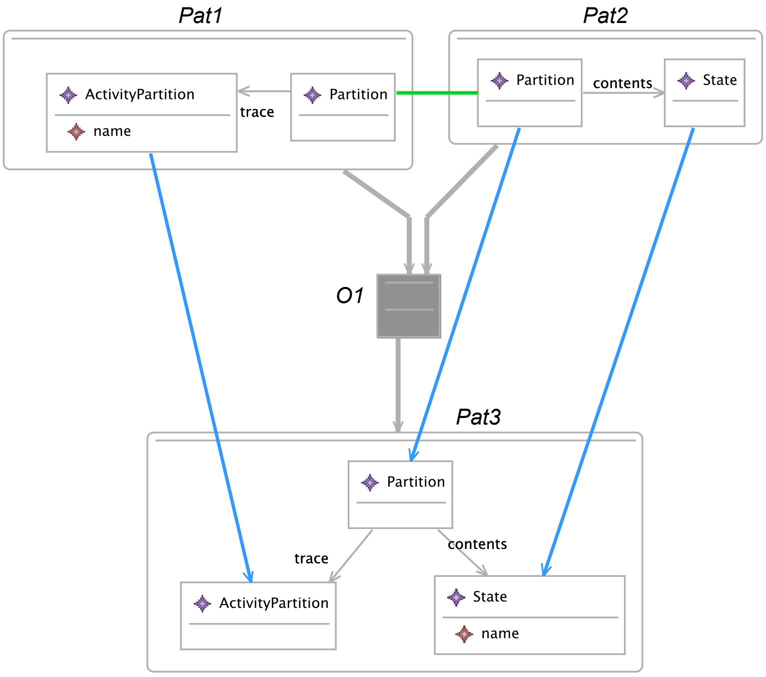


Figure 8.1: Transformation excerpt.

Algorithm 15 Alternative algorithm showing the behaviour of Combinator O1.

```
function EXECUTEO1′
    for $(p', s) \in Pat2$ do
        for $(p , ap) \in$ GETPAT1FROMPARTITION(p′) do
            . . .
        end for
    end for
end function
```

Intuitively, if we had 2 instances in $Pat1$ and 200 in $Pat2$, the second approach (Algorithm 15) would make about 200 accesses to the HashMap. If we assume that only 20 elements can be joined with the 2 instances in $Pat1$, that would give us an hash map hit rate of 10%. Which is awfull. On the other hand, the first approach (Algorithm 14) would give us a hit rate of 100% because the 2 accesses to the hash map return the 20 elements immediately.

Plus, if we decide on the approach of Algorithm 14, is it worth to keep and maintain the hash map that is never used? No. Maintaining an extra index is costly because whenever a new instance is added to that pattern, a new entry in the index has to be created as showed in section 7.1.3.3.

Since we are using hash set and hash maps for the indexes, when the transformation is initialized, all these data structures have to be initialized. That means we need to provide an estimate on the amount of memory to be allocated. Even if these structures support reallocation to increase capacity (and they do), the cost to perform this expansion in an hash set or hash map is too high for these transformations. If we can estimate the number of instances that will be in each pattern at the last configuration of the transformation, i.e., when the transformation ends, we can initialize the structures in such a way that chances of a reallocation in the middle of the transformation are greatly reduced. Given the right information, we can.

Returning to our previous example, if we knew that $Pat1$ will have 2 instances and $Pat2$ 200, we can at the very least, estimate that no more that 400 instances will be created in $Pat3$. But since there is a restriction, we know that the real number will probably be a lot less that 400. We need more information to give a better estimate. If we know could estimate with how many elements the 2 instances could be joined with we could provide a much better estimate. Using probabilities is a key technique to perform these estimations. For instance, if we know that for each instance in $Pat1$, the probability that its partition node exists in the any instance in $Pat2$ is 20% we can estimate that, executing the operator will create $2 \times 0.2 \times 400 = 160$. The probability of the operator producing an element is called its selectivity. Equation 8.1 shows the probability of operator $O1$ producing an element when executing. We have a minimum of two cases because we are considering the two possible execution order of the operator. If the two order produce different selectivities it means that the number of distinct values of the nodes in the Same restriction are different and thus, in a certain order, it is likely that there will be elements that will not be matched with the elements of the opposite pattern. Since only valid combinations are processed by the operator, the smaller of the two selectivities is likely the best guess.

Equation 8.2 shows the result of applying the operator selectivity to the maximum possible number of elements that can be created in the operation.

$$O1_{selectivity} = min(\frac{1}{Pat1.Partition.ndv}, \frac{1}{Pat2.Partition.ndv}) \tag{8.1}$$

$$Pat3.noi = min(\frac{Pat2.noi \times Pat1.noi}{Pat1.Partition.ndv}, \frac{Pat2.noi \times Pat1.noi}{Pat2.Partition.ndv}) \tag{8.2}$$

Unfortunately, with an estimate of 400 for pattern $Pat3$ we know for sure that the structures would never need reallocation but with an estimate of 160 obtained through probabilities, it is possible that the real value surpasses that estimate. But the problem with over approximating the necessary memory is that, with large transformations, more memory can be requested that what is actually necessary and in the advent that the computer does not have enough memory, pagination will spring into action, leaving the transformation hanged in the initialization phase. It is a tradeof. Our experiments (see section 8.3) show that the estimates are pretty accurate.

As we saw in the previous example, there are two main pieces of information that are necessary to perform these estimates for a given pattern $Patx$: the (expected) number of instances (NOI) of each pattern $PatY$ that is directly connected to $Patx$ and a way to estimate the probability of a successful combination.

A pattern $PatY$ is directly connected to a pattern $Patx$ when there is at least one combinator of which $PatY$ is operand and $Patx$ is a result. In the example, both $Pat1$ and $Pat2$ are directly connected to $Pat3$. That is why they are taken into consideration for the estimation of the number of instances for pattern $Pat3$.

The probability of a successful combination can be obtained by knowing the number of distinct values (NDV) of each node in the patterns. This, together with the assumption that the distinct values are distributed uniformly, allows us to come up with good enough probabilities and, thus, estimates.

In order to estimate the number of instances (NOI) and the number of distinct values (NDV) for patterns that don't have any predecessors, i.e., don't have any pattern directly connected to them, i.e., are results of some input external operator, we need real values collected from a representative sample of input models. We call this the Catalog.

The general approach to collect and propagate the statistics throughout the transformation is as follows: we start by generating the catalog model from a metamodel, this ensures that the catalog, instead of being a generic and not so useful structure, reflects the structure of the metamodel so it becomes easier to find statistical information later; next, a set of models are processed and added as samples to the catalog model; each sample holds statistics about the corresponding model; the samples are aggregated to form averaged statistics about the set of models; this information is then used to initialize the first patterns in the transformation and the rest is carried out by an algorithm that propagates the statistics across the entire transformation.

8.1 Catalog

The main purpose of a Catalog model is to store samples of models and aggregate those samples into statistics. These statistics then serve as a basis to estimate the Number of

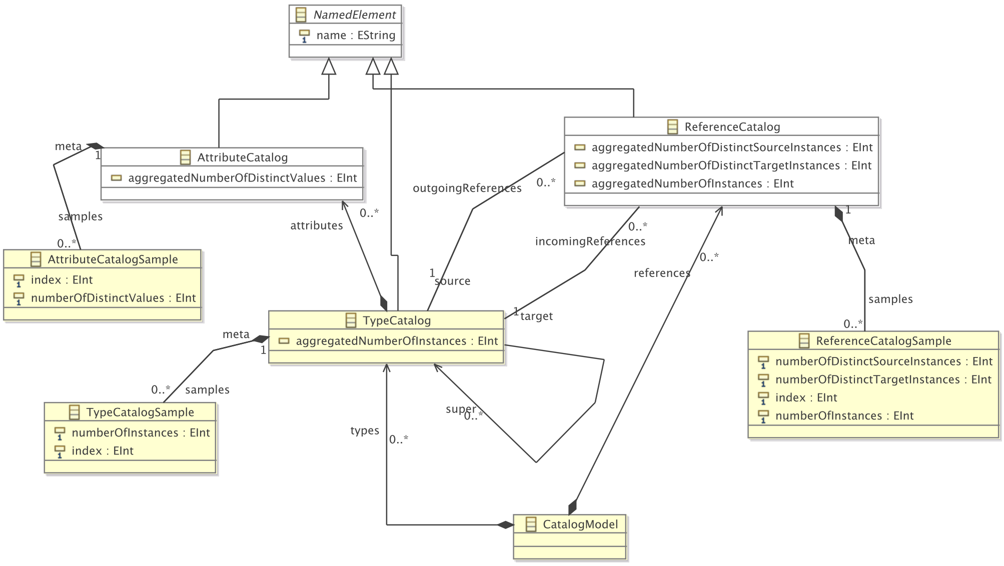


Figure 8.2: Catalog metamodel.

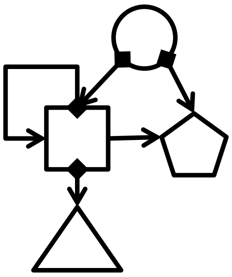


Figure 8.3: Shapes metamodel (equivalent to the one in Figure 6.16b).

Distinct Values (NDV) and Number of Instances (NOI) for all patterns and nodes in the transformation.

The metamodel of the Catalog is shown in Figure 8.2. A CatalogModel stores type catalogs and reference catalogs. A Type catalog represents statistical information about of element of a given type. For instance if the models being sampled have an element of type Square, there will be a type Catalog for square as Figure 8.5 shows. By the way, the catalog of Figure 8.5 was generated from the metamodel of Figure 8.3. A Reference catalog represents statistical information about one particular association. What completely defines an association is a the association name, the type of its source element and the type of its target element. This means a single reference in a metamodel can be translated into multiple reference catalogs if that reference's source or target elements have sub types.

Both type and reference catalog have multiple samples. Each sample is identified by an index and represent the information gathered that catalog in one particular model. In the example of Figure 8.5 we have one sample of the type catalog and one of each

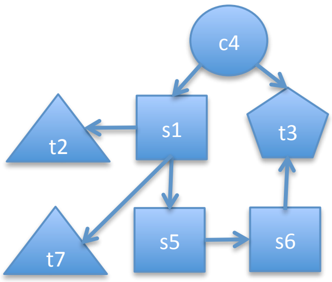


Figure 8.4: Shapes model sample.

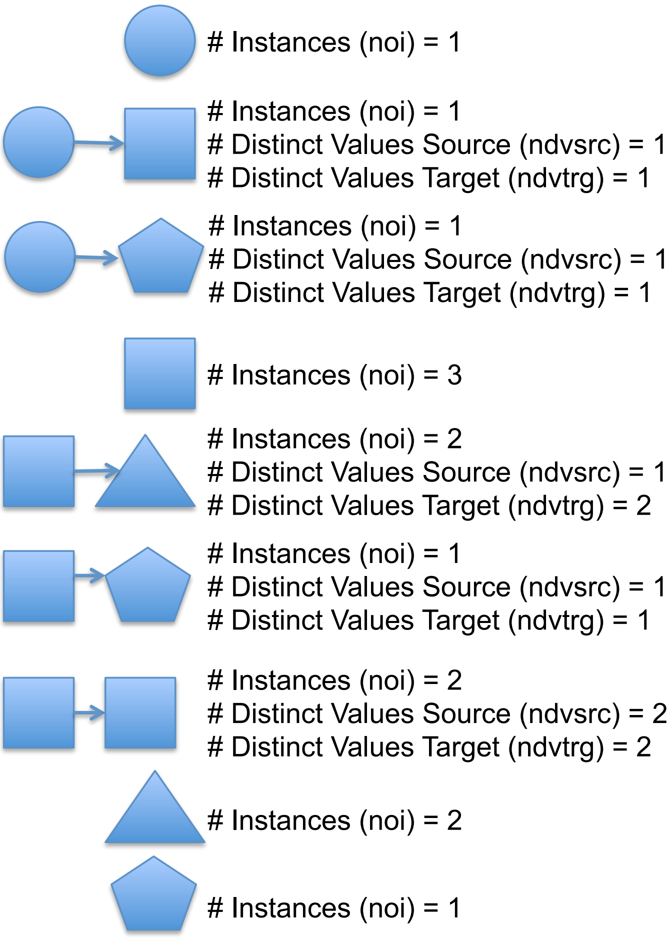

# Instances (noi) = 1
# Instances (noi) = 1
# Distinct Values Source (ndvsrc) = 1
# Distinct Values Target (ndvtrg) = 1
# Instances (noi) = 1
# Distinct Values Source (ndvsrc) = 1
# Distinct Values Target (ndvtrg) = 1
# Instances (noi) = 3
# Instances (noi) = 2
# Distinct Values Source (ndvsrc) = 1
# Distinct Values Target (ndvtrg) = 2
# Instances (noi) = 1
# Distinct Values Source (ndvsrc) = 1
# Distinct Values Target (ndvtrg) = 1
# Instances (noi) = 2
# Distinct Values Source (ndvsrc) = 2
# Distinct Values Target (ndvtrg) = 2
# Instances (noi) = 2
# Instances (noi) = 1

Figure 8.5: Catalog model sample generated from the metamodel of Figure 8.3 and the sample model of Figure 8.4.

reference catalog which indicates that the statistics where gathered by consuming one Shape model (the one on Figure 8.4).

A Type catalog sample counts the number of instances (NOI) of that type in a model wheres as reference catalog sample counts the number of instances of that reference plus the number of distinct source instances (NDSI) and the number of distinct Target Instances (NDTI). The NDSI by collecting all instances of the reference in a model and counting the number of distinct source elements. The NDTI is calculated in the same way. As an example, in the catalog of Figure 8.5 the sample of the "triangles" reference between squares and triangles has two instances but one distinct source values. This is because there is a square connected to two triangles. The same sample has three two target elements (this makes sense since triangles is a containment reference and an element cannot exist inside two parents).

There are also attribute catalogs, which store the number of distinct values of each attribute.

The aggregated attributes of a type, reference or attribute catalog is re-calculated when a new sample is added. Currently, we perform the arithmetic mean to compute all these values.

We built a simple transformation which takes an Ecore metamodel and produces a catalog model. Then we built a transformation that loads a given model and adds a new sample to all the type and reference catalogs in th catalog model and recalculates the aggregated values.

8.2 Number Of Instances Estimation

Once a properly filled catalog is available we can fill the estimates for those patterns that do not have any predecessors, i.e., those that are results of an external input operator. This is very simple since we know that all these patterns are atomic which means they are either a single element or a relation.

If the pattern is a single element, we search for the corresponding type catalog and fill the NDV with the aggregated NOI of the type catalog; and we register the attribute catalogs pertaining to that type catalog as these statistics will also be propagated throughout the transformation. The NOI of the pattern is the NOI of the type catalog. As an example, in pattern $Pat1$ of Figure 8.6, the Number of Instances is 3 and the Number of Distinct Values of the Square node is also 3.

If the pattern is a reference, we search for the corresponding reference catalog and fill the NOI of the pattern with the NOI of the reference. For the source node pattern of the reference, we fill the NDV with the NDSV of the reference catalog. Analogously, we fill the target node pattern's NDV with the NDTV of the reference catalog. Then the attribute statistics are filled for each node pattern by searching the corresponding type catalog, as is done for the case with a single element. As an example, the pattern $Pat2$ statistics were filled from the "triangles" reference catalog of Figure 8.5.

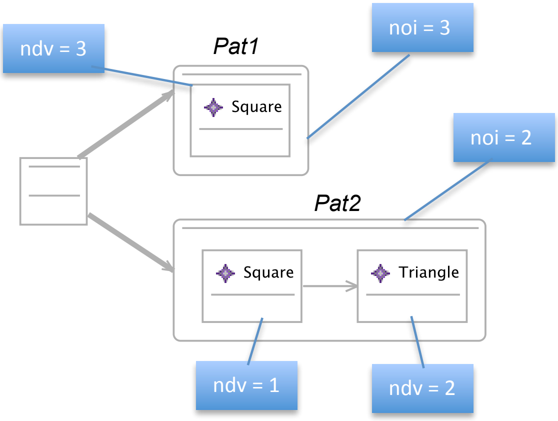


Figure 8.6: Transformation input frontier example with statistics filled from the catalog model of Figure 8.5.

Once the statistics for the patterns that don't have predecessors are filled, we can propagate these statistics through the network by using a FixPoint computation. At each iteration, one pattern is selected and statistics are attached to that pattern. For a pattern to be selected, all its predecessors have to have attached statistics. This means that, eventually, all the patterns in the network will be selected and their statistics computed [1]. The computation stops when no changes are made in the statistics of patterns.

The Algorithm 16 shows how the calculation is performed for a selected pattern. For the sake of brevity, we omit the implementation several utility functions and the calculation of the selectivity of operator conditions:

GETINCOMMINGOPERATORS(*Pat*) Returns the set of operators that, when executing, might insert elements in pattern *Pat*.

MAX(*X*, *Y*)

MIN(*X*, *Y*)

GETNODES(*Pat*) Returns all the nodes inside pattern *Pat*.

GETFIRSTOPERAND(*Op*) Returns the first (in creation order) operand of operator *Op*.

GETSECONDOPERAND(*Op*) Returns the second (in creation order) operand of operator *Op*.

HASTWOOPERANDS(*Op*) Return true if the operator has two operands. Otherwise, returns false.

COMPUTESELECTIVITYCOND(*Cond*) Computes the probability of a condition being true. This value depends on the number of distinct values of the parameters of the conditions.

[1] Remember that a network is connected, i.e., there are no separate patterns or operators.

GETSAMERESTRICTIONS(*Op*, *SrcPat1*, *SrcPat2*) Returns all Same restrictions between nodes of patterns *SrcPat1* and *SrcPat2* that affect the execution of operator *Op*.

GETNODEINRESTRICTION(*Restriction*, *SrcPat2*) Returns the node from pattern *SrcPat2* that participates in the restriction *Restriction*.

ISKEPT(*Op*, *Node*) Returns true if the Node is a target of a Keep restriction whose source belongs to some operand of operator *Op*.

GETSRCKEEPNODE(*Op*, *Node*) Assumes that ISKEPT(*Op*, *Node*) is true and returns that source node.

Also notice how Algorithm 16 selects the minimum selectivity from the two possible combination orders in function COMPUTEMINSELECTIVITY. The reason for this was already explained in the example at the beginning of this chapter.

Because Algorithm 16 is used in a fixpoint computation, when there are cycles in the transformation, only the first iteration of the each cycle is taken into account for the statistics because the statistics are based on the elements on the first pattern of the cycle. We are not able to predict of many iterations the cycle will perform with the information from the Catalog model.

We created a transformation that loads a given catalog model, initializes the estimates for the patterns that don't have any predecessors in the transformation and then propagates those estimations throughout the network with a fixpoint computation using Algorithm 16.

8.2.1 State of Art Analyses

The method we presented here to perform the estimations is very similar to the method used in database management systems [**Silberschatz:2010uw**] but adapted since in Tr-Net, pattern cannot have duplicates and there can be multiple sources of elements from one pattern.

In the state of the art tools (PROGRES, Viatra2 and GrGen.NET), cost estimation is performed each time a pattern matching operation is executed or compiled. They use the concept of a search plan to help materialize the specific of a pattern matching operation, i.e., the first element of the pattern to be searched, the order of the remaining elements will be searched, which indexes will be used, etc... A search graph is a graph from which all the possible search plans can be extracted. Each spanning tree is a search plan. For example, consider the pattern shown in Figure 8.7 and its representative search graph in Figure 8.8.

In each of these tools, the cost estimation (and subsequent optimization) of one pattern is performed by constructing a search graph, generating multiple search plans, and selecting the best one, i.e., the one with the least cost. Then they run it, or compile it to be run latter. Off course each tool has its specific way of representing the search graph and estimating the cost of each search plan.

Algorithm 16 Fixpoint algorithm used to propagate the statistics across a TrNet transformation.

```
function FILLSTATS( Pat)
    for Op ∈ GETINCOMMINGOPERATORS(Pat) do
        Selectivity ← COMPUTEMINSELECTIVITY(Op,Pat)
        MaxNOI ← COMPUTEMAXNOI(Op, Pat)
        Pat.noi ← MAX(Pat.noi , MaxNOI×Selectivity)
        for Node ∈ GETNODES(Pat) do FILLSTATSNODE(Op, Pat, Node)
        end for
    end for
end function
function COMPUTEMINSELECTIVITY(Op, Pat)
    if HASTWOOPERANDS(Op) then
        SrcPat1 ← GETFIRSTOPERAND(Op)
        SrcPat2 ← GETSECONDOPERAND(Op)
        SelectivityFirst2Second ← COMPUTESELECTIVITYSAMERESTRICTIONS(Op, SrcPat1, SrcPat2)
        SelectivitySecond2First ← COMPUTESELECTIVITYSAMERESTRICTIONS(Op, SrcPat2, SrcPat1)
        Selectivity ← MIN(SelectivityFirst2Second,SelectivitySecond2First)
        return Selectivity × $\prod_{Cond \in Op.Cond}$ COMPUTESELECTIVITYCOND(Cond)
    else
        return $\prod_{Cond \in Op.Cond}$ COMPUTESELECTIVITYCOND(Cond)
    end if
end function
function COMPUTEMAXNOI( Op, Pat)
    if HASTWOOPERANDS(Op) then
        SrcPat1 ← GETFIRSTOPERAND(Op)
        SrcPat2 ← GETSECONDOPERAND(Op)
        return SrcPat1.noi × SrcPat2.noi
    else
        SrcPat1 ← GETFIRSTOPERAND(Op)
        return SrcPat1.noi
    end if
end function
function COMPUTESELECTIVITYSAMERESTRICTIONS(Op, SrcPat1, SrcPat2)
    Selectivity ← 1
    for Restriction ∈ GETSAMERESTRICTIONS(Op, SrcPat1, SrcPat2) do
        SrcNode2 ← GETNODEINRESTRICTION(Restriction, SrcPat2)
        Selectivity ← Selectivity × $\frac{1}{SrcNode2.ndv}$
    end for
    return Selectivity
end function
function FILLSTATSNODE(Op, Pat , Node)
    if ISKEPT(Op, Node) then
        SrcNode ← GETSRCKEEPNODE(Op, Node)
        Node.ndv ← MIN(Pat.noi, MAX(Node.ndv, SrcNode.ndv))
    else
        Node.ndv ← Pat.noi
    end if
end function
```

As an example, we will demonstrate the approach of Viatra2 [**Varro2006a**]. According to Varró et al. [**Varro2008**], the cost of a search plan is given by the potential size of the search tree formed by its execution [2]. For example, the potential size of the search tree for the search plan shown in bold in Figure 8.8 is given by $\bar{z} + \bar{z} * \bar{x_z} + \bar{z} * \bar{x_z} * \bar{y_x}$, where: $\bar{z}$ denotes the expected number of model elements that can be matched by the z element; $\bar{x_z}$ denotes the average number of model elements that can be matched by x after binding z to some model element and $\bar{y_x}$ denotes the same for y after binding x to some model element. Intuitively, each term of the formula represents the number of states at a specific level of the search tree. $\bar{z}$ represents the number of states in the first level, i.e., how many candidate mappings can be found for the z element. $\bar{z}*\bar{x_z}$ represents how many candidate mappings we can find for x, provided we have already searched for z, i.e., it counts the number of states in the second level of the search tree. For any arbitrary pattern, the cost formula would be computed following this pattern.

You can see that the information we gathered in the catalog model can be used to estimate these values. $\bar{z}$ is given by the number of instances of C. $\bar{x_z}$ can be estimated using the information about the relation $C \rightarrow A$, and so on.

As a concrete example, consider the search graph shown in Figure 8.9 whose weights have been filled using statistics from some model. The $\bar{y_x}$ is represented in the edge $x \rightarrow y$ and its value is 1.333. The $\bar{y_x}$ value represents the average number of choices available after matching x to some model element. The x can be bound to any of the 3 A elements in the model of Figure 4.1a. If we consider that the 4 edges between A and B elements are uniformly distributed, we get an average of 1.33 edges going out of each A element. This means the average number of choices the algorithm has to do for each x element bound is 1.33. Following this cost model, the evaluation of the search plan of Figure 8.9 yields a potential size of 9.99 whereas a search plan that follows the order $y \rightarrow x \rightarrow z$ yields a cost of 12, thus the first search plan is better.

We can argue that TrNet is very large search plan and, having the statistics estimated for each pattern, it is easy to calculate the cost of execution of each operator. In fact, we do that in section 9.2 to decide on the order in which to combine the operands of a combinator. The cost of larger portions can then be calculated by summing the cost of each combinator pertaining to that portion. It is possible to select specific parts of the

[2] A pattern matching operation is a backtracking search.

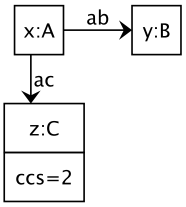


Figure 8.7: Sample pattern.

transformation, change them while keeping the intended result and evaluate them to decide if the new search plan should be kept instead of the old one. Unfortunately, due to time constrains, it has not yet been implemented.

8.3 Evaluation

In order to measure how good the estimation TrNet produces are, we took three different transformations and three input models; we created the catalog model for each of those input models; we ran run the algorithm explained previously to initialize each transformation with the statistics of the corresponding input model; we compiled the transformations and ran them. In the end of each transformation we dumped the final configuration and plotted the charts shown in figures 8.10, 8.11 and 8.12.

The vertical axis of each chart represents the number of instances and the horizontal one represents the depth of the pattern. Each point represents the number of instances (NOI) of one pattern. The more to the right a point is, the more far away from the beginning of the transformation the pattern is. We choose to sort the patterns according to their distance to the beginning (depth) because in some cases we can see the error propagation. There are two lines: the blue is the real NOI, read from the final configuration; and the red one is the estimated NOI.

Of course one can argue that the estimations are so good because we created the catalog model for the exact input model that was going to be transformed. Our purpose for this experiment was to see how the error propagates across the network using our estimation approach. As you can see in the pictures, it is fairly good.

The first chart was obtained from a transformation that translated class models into relational models. More importantly, its structure explains the chart: since the transformation has around 50 patterns, which is not much, and was crafted by hand, meaning it was not generated, it does not have a high branching factor and that is why, at the depth of 10 we start to see small errors in the estimation.

The second and third charts (figures 8.11 and 8.12) represent the same transformation: the case study for our benchmark which is explained in Chapter 10. The particularity here is that one transformation was crafted manually and has around 70 patterns while the

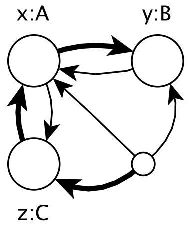


Figure 8.8: Example search graph and a possible search plan (in bold) for pattern of Figure 8.7 in page 103.

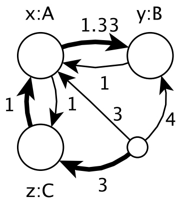


Figure 8.9: Weighted search graph a possible search plan (in bold) for pattern of Figure 8.7 in page 103.

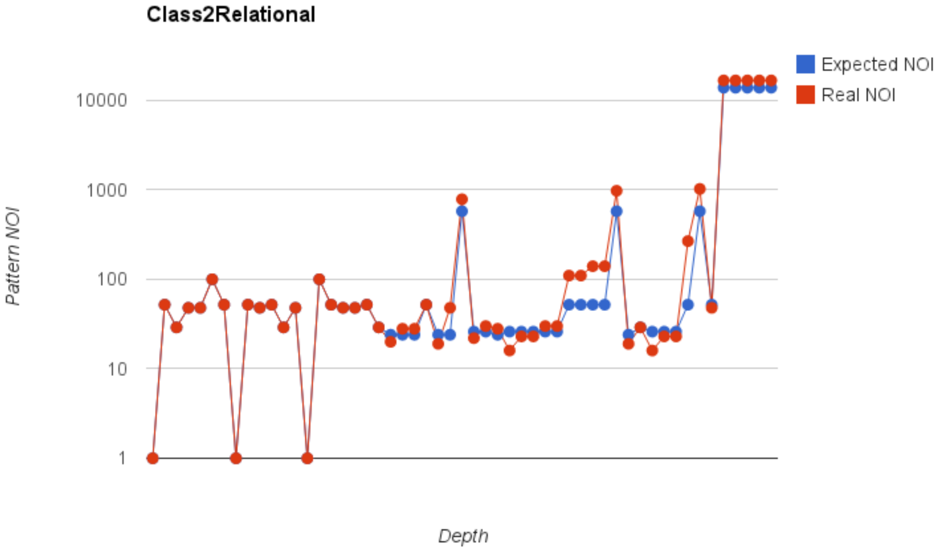


Figure 8.10: Expected and real number of instances of a transformation that translated Class models into Relational model, expressed in TrNet.

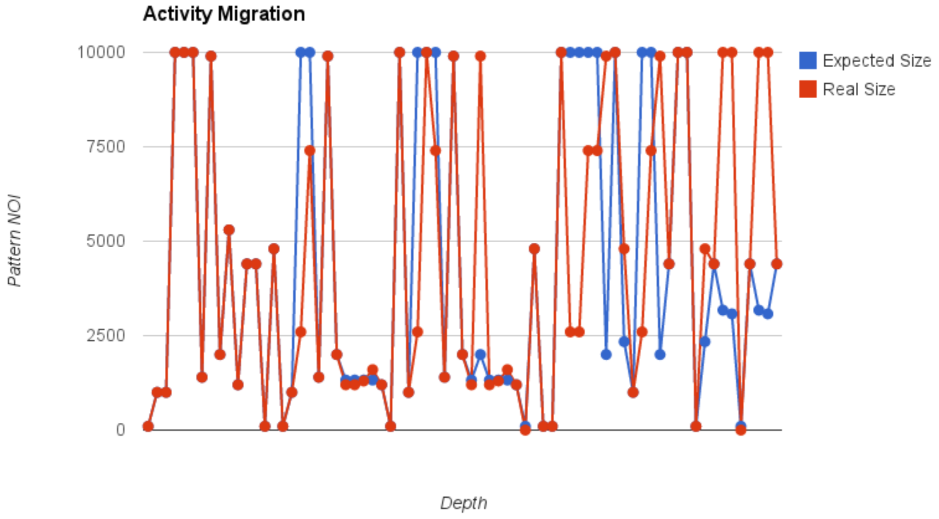


Figure 8.11: Expected and real number of instances of a transformation that migrates Activity Diagram models into UML 2.0 Activity Diagrams, expressed in TrNet.

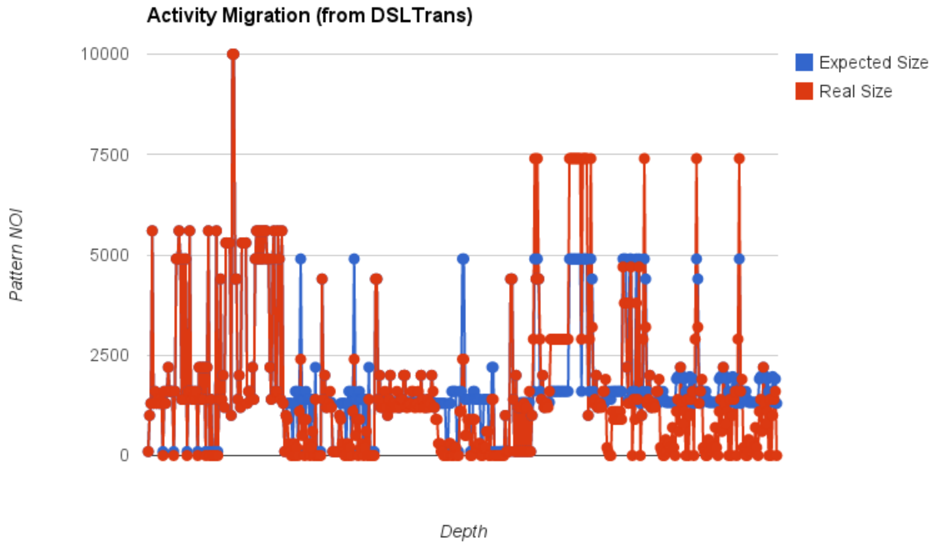


Figure 8.12: Expected and real number of instances of a transformation that migrates Activity Diagram models into UML 2.0 Activity Diagrams, expressed in DSLTrans and then compiled to TrNet.

other was generated from an equivalent DSLTrans transformation and has 470 patterns. The first one is represented in chart of Figure 8.11 and the second one is represented in figure 8.12. The curiosity is that, for the transformation crafted manually, the error per pattern is bigger than that of the transformation that was generated. This is because we took advantage of more advanced application conditions to make the transformation more compact and, since the estimation algorithm can only assume the probability of these conditions, it created slightly wrong estimations that were then propagated through the rest of the transformation. In the generated transformation this was not the case: DSLTrans does not support these advanced features so, although a lot more "verbose", the transformation generated from DSLTrans is a lot more simple causing the estimation algorithm to give better estimates.

Another peculiarity is that, although the two transformations were run with the exact same input model, with around 10000 elements of each type, we can see that most of the patterns in the transformation generated from DSLTrans do not have a NOI equal to 10000 while the other transformation has. The branching factor of the former transformation is a lot bigger than the one for the transformation crafted manually.

Overall the estimations are very accurate and extremely useful. They are used for everything since the initialization of data structures to the execution order of operators. In te next chapter we explain the various optimization techniques that we developed to enhance the speed and memory consumption of TrNet transformations.

9

Optimizations

Most of the techniques presented in this chapter depend on the analysis described in the previous chapter to evaluate the cost of certain operations and to properly manage and allocate the runtime resources.

The optimization techniques can be applied as an endogenous model transformation or during the code generation process.

The techniques that are implemented as a model transformation analyze the information added by the analyses presented in the previous chapter and then add information that later instructs the code generator to produce the most efficient code.

Those techniques applied during the code generation process take advantage of the information in the transformation to decide which structures are the most efficient.

9.1 Execution Order Inference

Execution order inference is an optimization technique that tries to define an execution order such that each operator executed only once in the whole transformation.

This optimization is so crucial that without it, executing a trnet transformation requires keeping, at runtime, a queue of operators to be executed and managing that queue throughout the transformation.

Instead, with execution order inference, the execution order is hard-coded so it runs a lot faster and the runtime environment is also more simple.

The optimization is very similar to a topological sort algorithm where the dependencies between the operations are recorded and then, eliminating one “free” operator at a time, we come up with an optimal execution order. An operator depends on other operator if the execution of the later adds elements to any of the former’s operands. Visually,

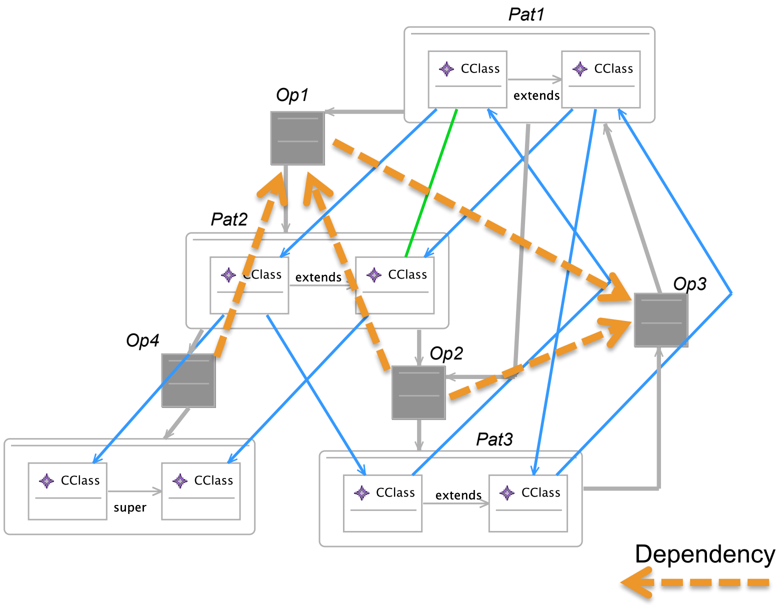


Figure 9.1: Sample cycle with dependencies between the combinators.

the later operator is connected to the former by a pattern in between. For the sake of conciseness we do not present the algorithm here. Please refer to this thesis' accompanying files.

If a TrNet transformation has no cycles, the algorithm always terminates with an execution order that covers all the operators of the network. However, the optimization supports simple cycles (like the one shown in figure 9.1). Within a cycle, all the operators have dependencies so no one can be picked (see Figure 9.1). If this happens, the algorithm has to inspect all the operators in the cycle to discover with is the one that marks the beginning of the cycle and which is the one that marks the end. With those two operators selected, it defines the execution order inside the cycle, removing the dependencies between the operators inside the cycle. The algorithm then continues for the rest of the transformation. Note that support for cycles within cycles was not implemented. Detecting complex cycles is a very complex problem as **[DBLP:journalssiamcompJohnson75]** and **[1602189]** suggest. If the transformation has complex cycles, this optimization cannot be used. However, most use cases involved simple to no cycles. They are mostly used to compute transitive closures, create auxiliary relations (such as the trace relations), and then use those relations to do something else.

The execution order inference is executed as a model transformation that adds "next" relations between operators to reflect their execution order. See figure 9.2 for an example. These relations are then used by the code generator to quickly generate the run method of the transformation class.

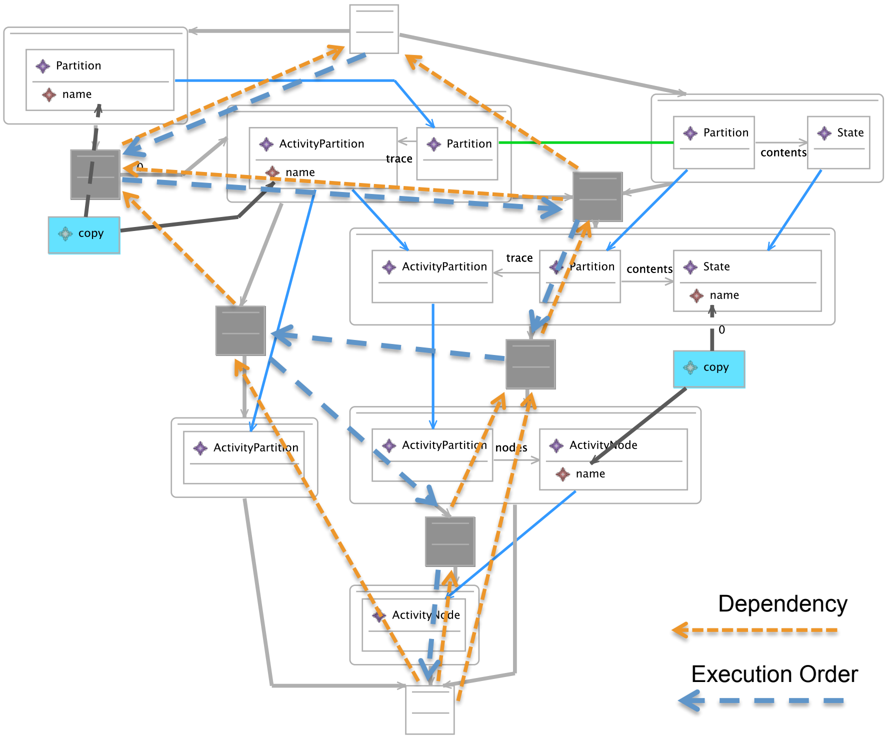


Figure 9.2: Sample transformation with dependencies between the combinators and one selected execution order.

9.2 Join Order Optimization

Join Order Optimization refers to deciding how a combinator that has two operands will execute. More specifically, since the combinator will execute two for loop, one inside the other, decider which is the outer loop and which is the inner one is an important decision regarding performance.

This decision not only affects performance but also helps to reduce memory consumption by letting the code generator component know which indexes will be used in the transformation. More on this in section 9.3.

As an example, consider the operator of figure 9.3 and the corresponding two possible algorithms to execute it, 17 and 17.

Algorithm 17 Algorithm showing the behaviour of Combinator O1.

```
function EXECUTEO1
    for $(p, ap) \in Pat1$ do
        for $(p', s) \in$ GETPAT2FROMPARTITION(p) do
            ...
        end for
    end for
end function
```

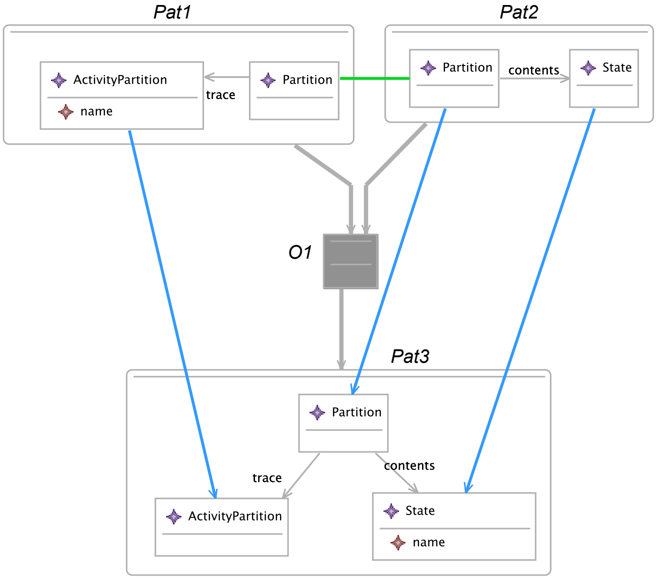


Figure 9.3: Transformation excerpt.

Algorithm 18 Alternative algorithm showing the behaviour of Combinator O1.

```
function EXECUTEO1′
    for $(p', s) \in Pat2$ do
        for $(p\ , ap) \in$ GETPAT1FROMPARTITION(p′) do
            ...
        end for
    end for
end function
```

Let us estimate the cost to execute the operator using algorithm 18: the outer loop performs $Pat2.noi$ iterations. For each iteration in the outer loop, theres is an index access, which costs 1 because it is implemented with an hash table (see Section 7.1). Following that access, it performs as many iterations as the elements that are estimated to be found in the index for each element in $Pat2$. If you remember the analysis performed in the previous chapter, the probability that a given element partition exists in $Pat1$ is $\frac{1}{P1.Partition.noi}$. Applying that probability to the number of elements in $P1$ yields an estimation of the number of elements satisfying this condition. Thus the cost formula is:

$$C_{P_2 \rightarrow P_1} = P2.noi \times (1 + \frac{P1.noi}{P1.Partition.ndv}) \tag{9.1}$$

Analogously, the cost of execution the combination using algorithm 17 is:

$$C_{P_1 \rightarrow P_2} = P1.noi \times (1 + \frac{P2.noi}{P2.Partition.ndv}) \tag{9.2}$$

Now what the optimization does is to define the order of the operands across the whole transformation by performing these calculations and choosing the order that costs less.

As for a general combinator, with conditions and other same restrictions, the term of the cost formula that varies more is the probability of processing a given element from all the possible combinations of its operands. This is the selectivity of the operator and its calculation is shown in algorithm 16 of section 8.2.

The Join Order Optimization is implemented as a model transformation that gives an index to each operand, reflecting the order of the generated loops. The code generator uses this information to generate the code.

9.3 Index Pruning

Index Pruning is a memory optimization technique that is applied during the code generation process. The code generator uses the information produced by the Join order optimization technique to determine which indexes will be used throughout the transformation and only creates those indexes. By default, without using the information left by the join order optimization, it would generate an index for each mandatory node that participates in a same restriction as is explained in section 7.1.

As an example, if the join order optimization decides that the combinator depicted in figure 9.3 is implemented using the algorithm 17, the GETPAT1FROMPARTITION index is no longer necessary and hence, will not even exist in the generated code. Off course we assume that this index is not used in the implementation of other combinators.

This optimization is applied during the code generation process.

9.4 Early Evaluation

Early evaluation is, as the name indicates, an optimization that tries to evaluate conditions as soon as possible. This is best explained with an example: Suppose the Combinator of figure 9.4 is implemented with algorithm 19. Would not it be better if the condition COND(p) was evaluated outside and before the inner for loop as is shown in Algorithm 20? This can be done since the only parameter of the condition is the first operand, so if it is false, it is false for all elements of the second operand.

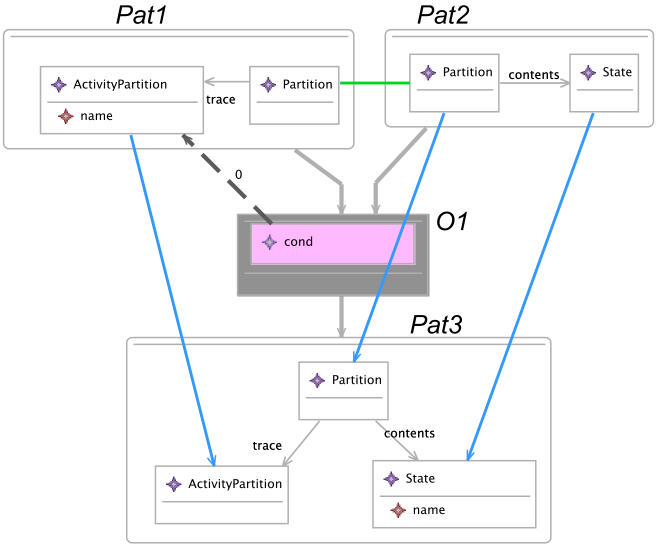


Figure 9.4: Example of a combinator where early evaluation of the condition is possible.

Algorithm 19 Algorithm showing the behaviour of Combinator O1 of the Figure 9.4.

```
function EXECUTEO1
    for $(p, ap) \in Pat1$ do
        for $(p', s) \in$ GETPAT2FROMPARTITION(p) do
            if COND($p$) then
                ...
            end if
        end for
    end for
end function
```

In general, if there are conditions that depend only on one operand, and if that operand is the first, then they are evaluated outside the inner loop. Notice that the way the join order optimization decides the order of operands favors the application of this optimization because we are taking into account the conditions when estimating the probability of a successful combination. This, by definition, selects the most restricted (both by conditions and number of instances) operand to be the first.

This optimization is applied during the code generation process.

Algorithm 20 Algorithm showing the behaviour of Combinator O1 of the Figure 9.4 with early evaluation.

```
function EXECUTEO1
    for $(p, ap) \in Pat1$ do
        if COND($p$) then
            for $(p' , s) \in$ GETPAT2FROMPARTITION(p) do
                ...
            end for
        end if
    end for
end function
```

9.5 Memory Allocation

Since the analysis presented in the previous chapter allows the code generator to know the expected number of instances (NOI) in each pattern in the final configuration of the transformation, it is possible to initialize all the structures used to keep pattern elements (see section 7.1) with the right amount of memory.

As an example, assume that, in figure 9.4, the NOI of Pat2 is 23 and the noi of Pat1 is 10. In the constructor of the transformation class, the size of the structures for Pat2 would be initialized to 23 and the same for Pat1, as is shown in listing 9.1.

Listing 9.1: Transformation class constructor.

```java
public class ActivityMigrationTransformation
  implements TrNetPat1InstanceListener, ...{

  HashSet<TrNetPat1Instance> trNetPat1Set;
  ArrayList<TrNetPat1Instance> trNetPat1Array;
  HashSet<TrNetPat2Instance> trNetPat2Set;
  ArrayList<TrNetPat2Instance> trNetPat2Array;
  ...
  public ActivityMigrationTransformation() {
    ...
    trNetPat1Set = new HashSet<TrNetPat1Instance>(10);
    trNetPat1Array = new ArrayList<TrNetPat2Instance>(10);
    trNetPat2Set = new HashSet<TrNetPat2Instance>(23);
    trNetPat2Array = new ArrayList<TrNetPat2Instance>(23);
    ...
  }
  ...
}
```

9.6 Overlapped Pattern Matching

Overlapped pattern matching is one of the most complex optimization techniques. The main purpose is to identify patterns or combinators that are redundant in the transformation and remove them. While the existence of redundant patterns and/or operator seems highly unlikely for a TrNet transformation created manually, the same is not true for an automatically generated one.

For instance, the way rules are compiled from DSLTrans to TrNet already shows us that there will be a lot of redundant patterns. It would not make sense to try to optimize the DSLTrans compilation since the purpose is that, in the future, TrNet supports the implementation of other high-level model transformation languages. Hence, the compilation process should remain as simple and naive as possible, while studying and development more sophisticated optimizations in TrNet.

For a pattern PX to be redundant there must be other pattern PY that, in the final configuration of the transformation, keeps the exact same elements that PX does. In other words, both PX and PY have the same inputs. If such PY exists, then PX can be removed from the network and all the operators that depend on PX become connected to PY. For example, consider the excerpt of a transformation depicted in figure 9.5. One of the two patterns in red is redundant because their nodes and edges are isomorphic and they both share the same inputs. Plus, we can remove one of them, provided we inherit the outputs and restriction, because they are not operands of the same Combinator.

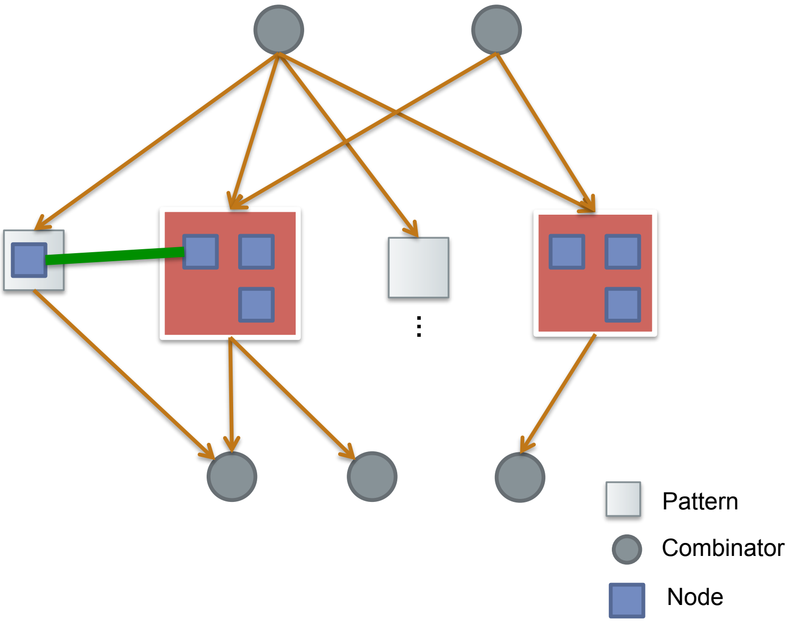


Figure 9.5: Representation of a transformation excerpt where two patterns are redundant (in red).

In general, in order to conclude that two patterns, px and py, are equal the following conditions have to be true:

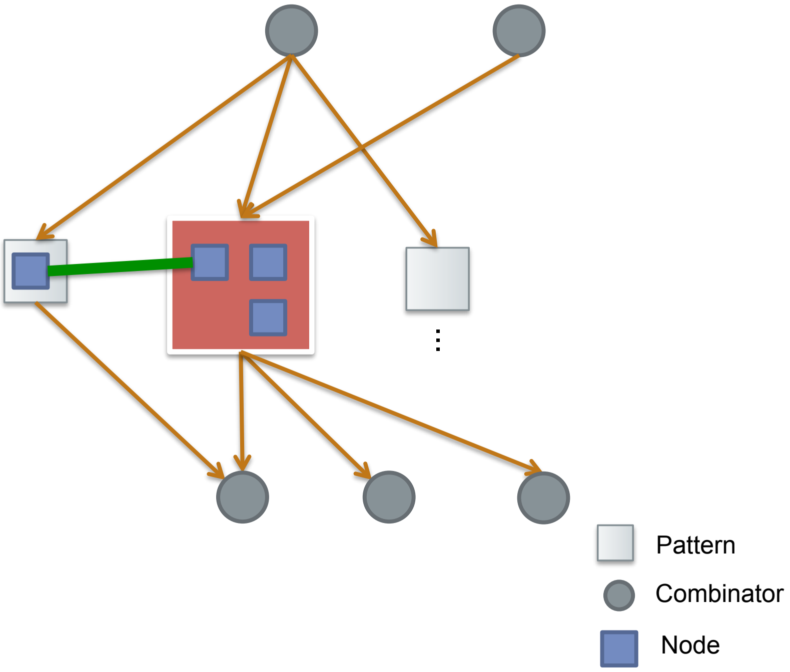


Figure 9.6: Resulting transformation after applying the overlapped pattern matching technique to the transformation depicted in Figure 9.5.

1. There must be an isomorphism between the graphs formed by the nodes and edges of px and py.
2. There must be a bijection between the operators that add elements in PX and PY.
3. PX and PY are not operands of the same combinator.

Notice that two nodes are considered equal is they have the same name and the same computed attributes. Also, a node that is the result of a keep restriction is only considered equal to other node if, and only if, the other node is also the result of a keep restriction, and both restrictions have to have the same node as source. Computed attributes, if they exist, are the attributes that are being created when elements are added to the pattern.

The third condition exists because two, otherwise equal, patterns, when operands of the same combinator, cannot be reduced to one because the combinator has to perform the combinations of the two patterns. That would affect the result of the transformation.

Once the optimizer concludes that two patterns, PX and PY are equivalent, it removes one, say PX. This operation has to ensure that all the elements that depend on the pattern PX will depend instead of the pattern PY. For example, figure 9.6 shows the result of removing the right pattern identified as redundant in the transformation of figure 9.5. Notice that the output of the pattern that was remove was inherited to the other pattern. If there were other restrictions connecting the pattern that was removed to other patterns they would be inherited too.

In the general case, removing a pattern PX that is equal to a pattern PY involves connecting PY to all the operators that are successors of PX and merging the information of the isomorphic nodes. Any same restriction that connects to a node in PX has to be altered to connect to the correspondent node in PY.

The optimization also identifies and removes redundant combinators. For a combinator CX to be redundant, other operator CY has to be found satisfying the following conditions:

1. There must be a bijection between the conditions of each combinator.
2. There must be no actions in one of the combinators.
3. There must be a bijection between the patterns that precede the combinators.

Two conditions are considered equivalent if they refer to the same function and if there is a bijection between their parameters. There can not be actions in both combinators because since we assume that actions have collateral effects, we can not remove a combinator that has actions. The third condition compares patterns with the identity equality and not with the equality described in the previous paragraphs.

As an example, the combinators depicted in figure 9.7 are considered equivalent.

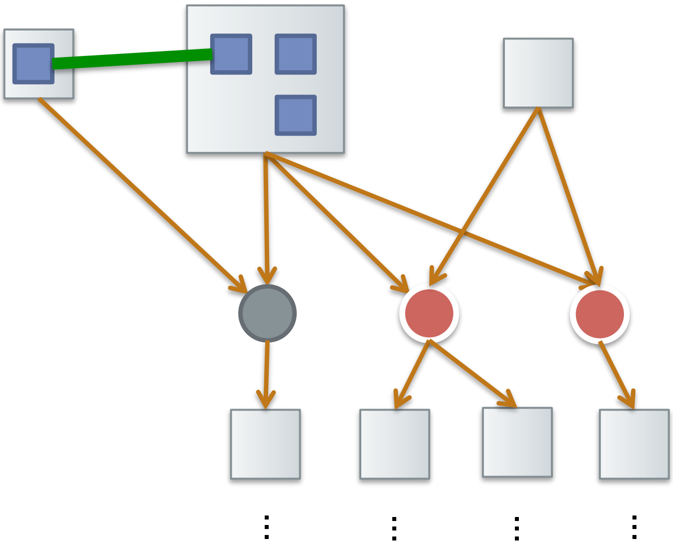


Figure 9.7: Representation of a transformation excerpt where two operators marked in red are redundant.

Once the optimizer concludes that two combinators CX and CY are equivalent, it chooses the one that has no actions, say CX to be removed. For example, figure 9.8 shows the result of removing the right combinator. Notice that the outputs of the operator that was removed were inherited by the other operator.

In general, removing a combinator CX that is equivalent to other combinator CY involves changing the patterns that succeed CX to be instead successors of CY.

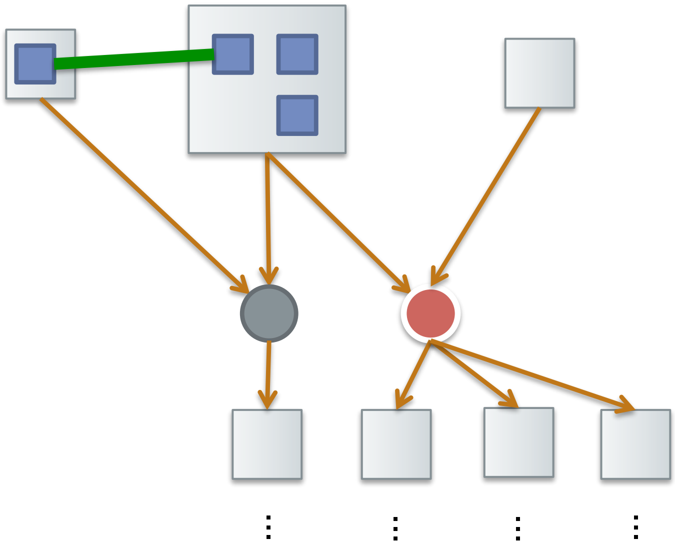


Figure 9.8: Resulting transformation after applying the overlapped pattern matching technique to the transformation depicted in Figure 9.7.

This optimization is implemented as a model transformation. Because of all the graph isomorphisms and bijections, the overlapped pattern matching optimization is a very costly operation so instead of comparing every pair of pattern or operator for equivalency, the optimizer starts in the input external operators and performs a DFS exploring every pair of patterns leaving one common operators, or every pair of operators successors of a pattern.

This optimized approach works because of one simple fact: if two patterns are equivalent, they ought to be both successors of same, common, operator. Even if there are others, at least one common operator has to exist. If there is not common operator, then it means that the optimizer as concluded that all pair of precedent combinator are not equivalent, so the current pair of patterns will not be equivalent as well. The same is true for pairs of combinators. This means that the optimizer will only explore those pairs of patterns or combinators that are interesting.

Also, in order to explore all the possible candidate pairs of patterns or operators in all the transformation, whenever a redundant operator or pattern is removed, the optimizer restarts the search at the beginning of the transformation. This ensures that, when the optimizer finishes exploring all elements of the transformation, all possible interesting pairs of patterns or combinators have been evaluated to not equivalent.

Because of the frequent restarts, we implemented caching mechanisms to avoid recomputation of interesting pairs of patterns.

Since this optimization is more complex than the others, we wanted to measure its impact, i.e., we wanted to know if it was really worth the effort to implement. The experiment consisted in instrumenting the optimizer to print the total number of patterns and operators after each restart (pattern or combinator removal). When the optimization was

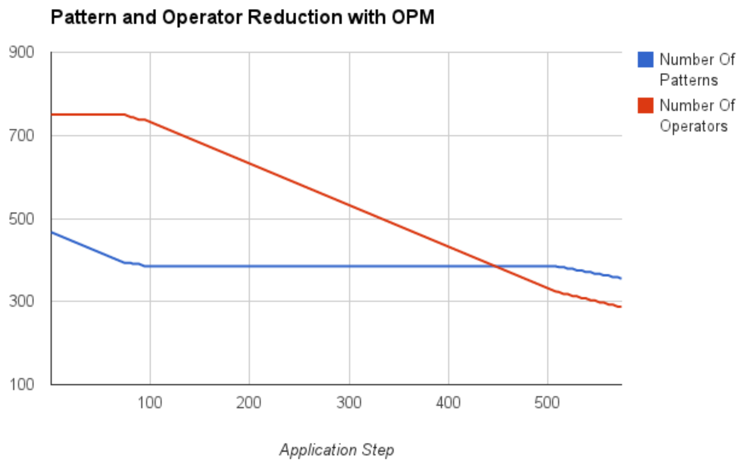


Figure 9.9: Evolution of the number of patterns and number of operators in a TrNet transformation during the application of the Overlapped Pattern Matching technique.

applied to an hand-made transformation, it had almost no impact but when it was applied to a transformation that was generated transformation from DSLTrans, the results were outstanding. Figure 9.9 shows the evolution of the number of patterns and combinators after each restart in a transformation that was generated from other DSLTrans transformation.

As the picture indicates, the total number of patterns were reduced to less than a half and the number of operators were reduced around 70%. The less patterns we have in a transformation, the less memory will be allocated and the less operators we have, the more quickly the transformation executes.

The optimization has a lot of impact in transformations that were generated from DSLTrans transformations because in DSLTrans, the match pattern of each rule has to be matched against the input model. This means that, in a corresponding TrNet transformation, most of the patterns needed to apply the rule are successors of the input external operator. For example, consider the two rules shown in figure 9.10 and note that the two rules share a common pattern. When compiled to TrNet, the two rules generated a transformation like the one shown in Figure 9.11.

After the application of the overlapped pattern matching we have a transformation like the one shown in Figure 9.12. Notice that, with the optimized version, the common pattern of the rules is matched in parallel for the two rules. The pattern matching process is overlapped for the two rules.

Considering the optimization techniques presented in Chapter 4 we conclude that TrNet:

- By design, applies structural indexing since each pattern in TrNet is an index were elements are added by the operators;

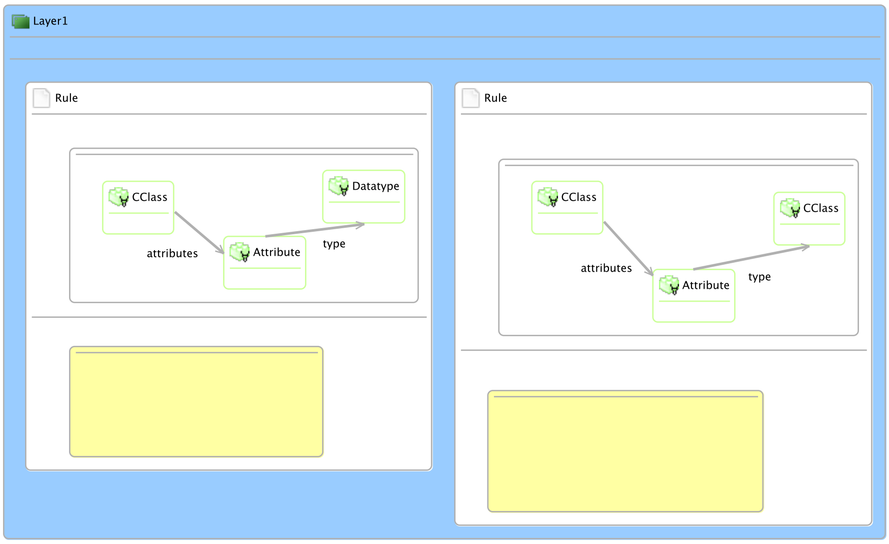


Figure 9.10: DSLTrans Transformation sample.

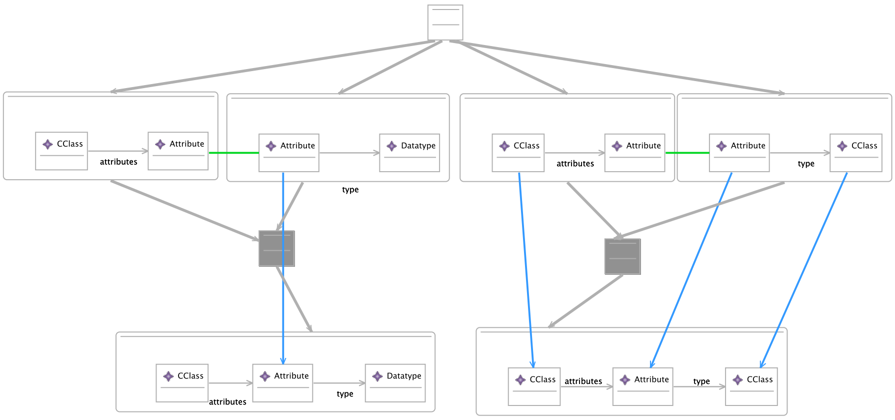


Figure 9.11: TrNet Transformation sample compiled from the one in Figure 9.10.

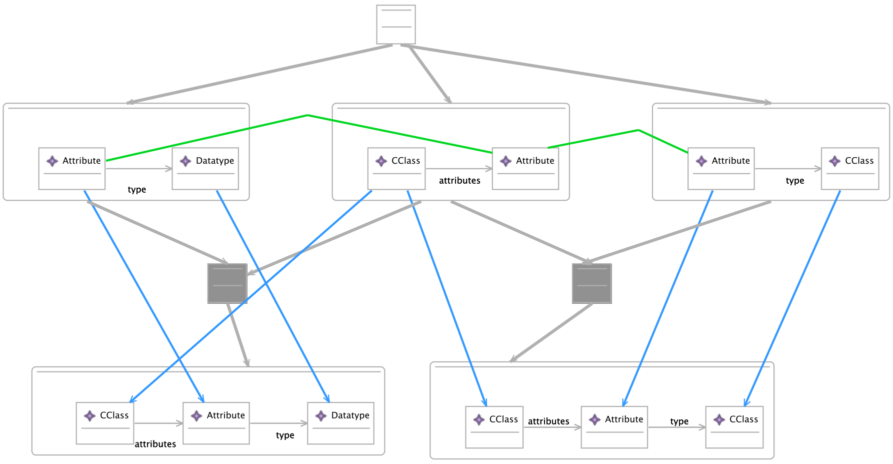


Figure 9.12: TrNet Transformation sample after the application of Overlapped Pattern Matching.

- Uses model sensitive, search plan optimization - although not to its full impact;
- Applies overlapped pattern matching without limitations.

We say that TrNet does not apply full fledged search plan optimization because the optimizer only decides the order of which the combinator will combine its operands. To apply this techniques in it maximum, certain parts of the transformation, for instance, two or three consecutive combinators and patterns in between, should be re-arranged to meet lower cost of execution. This involved detecting on which parts of the transformation there are no elements being created so that they can be arranged. Due to time restrictions, we did implement this technique, yet.

10

Benchmark

Since this thesis is about developing an approach that mitigates the abstraction penalty, we needed to found a way to measure empirically how good is our solution. The benchmark described in this chapter serves that purpose by comparing our approach with other existing languages in terms of performance.

10.1 Case Study - Activity Model Migration

The case study we used was proposed in the Transformation Tool Contest (TTC) 2010 and represents a typical model transformation language usage scenario: A normal language evolves overtime and, whenever the metamodel changes, there is a risk that all models conforming to the old metamodel become invalid in relation to the new metamodel so the model transformation language is used to create and execute a transformation that, when applied to the invalid models, makes them valid again. An high level model transformation language such as DSLTrans is preferred in this scenario because the purpose is to rapidly create a transformation that performs all the migration work.

We choose this case study, not only because it represents a typical model transformation language use case but also because of the tools that implement the case study, namely, Epsilon Flock, ATL, GrGen.NET, etc. . . These are current tools that have some success in the community, from a usage perspective. In addition, these were the only tools that can run in a MacOS X machine. Another challenge was to migrate the transformations submitted to the contest as the tools, being languages, already had changed.

A solution to the case study is a model transformation that translates activity diagram models expressed in UML 1.4 to activity diagram models expressed in UML 2.2.

Figures 10.1 and 10.2 show simplified versions of the old activity diagram metamodel and the new metamodel, respectively. The conceptual difference in the two versions of the activity diagram is that in the older version, activities are defined as a special case of state machines whereas in the later version, they are defined in a more general and intuitive way [**Rose2010_ModelMigrationCaseforTTC2010**].

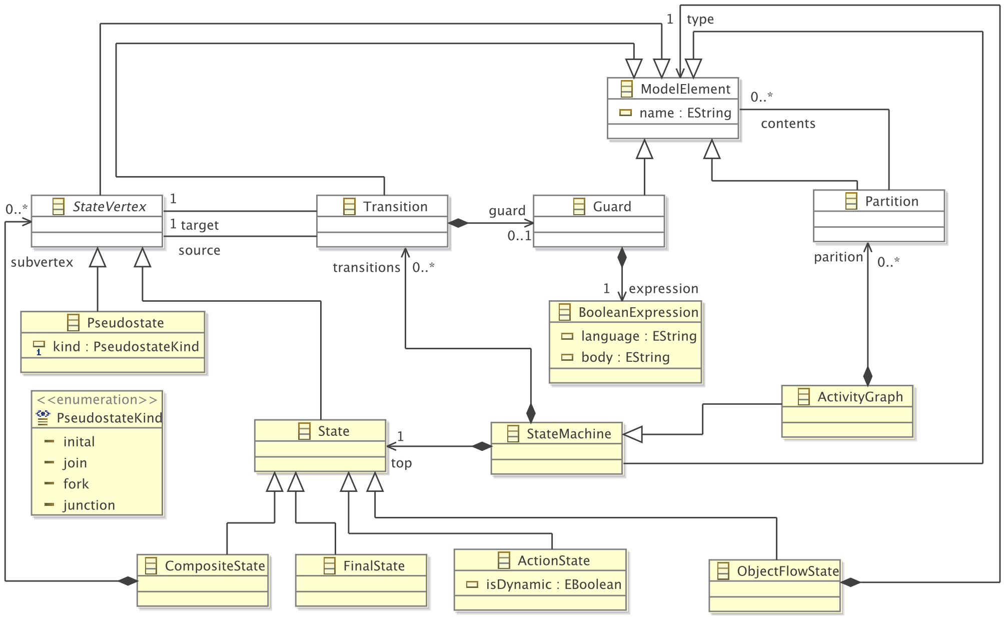


Figure 10.1: UML 1.4 Activity Graphs (based on [**Rose2010_ModelMigrationCaseforTTC2010**]).

Apart from the conceptual difference, and ignoring concrete syntax differences, activities now comprise nodes and edges, actions replace states and the subtypes of control node replace pseudo states and the kind attribute. Partitions are now ActivityPartitions; Transitions can be either ObjectFlow or ControlFlow, depending their source and target, etc. . . For full details about the differences please refer to [**Rose2010_ModelMigrationCaseforTTC2010**].

The activity migration transformation expressed in DSLTrans is quite trivial. That is the main purpose why we should use an high level language to perform the migration instead of using TrNet or other low-level language.

The DSLTrans transformation contains only two layers: the first one to perform mainly one-to-one mappings and create the elements that will comprise the migrated model; and the second one, which connects the previously created elements together to form the migrated model.

We will only show some of the rules. For the full transformation please refer to these thesis' provided projects. Figure 10.3 shows a rule that transforms a partition into an activity partition, keeping the same name. The rule of figure 10.4 shows how the migration of a guard expression is performed: an opaqueExpression is created where the body is the boolean expression body and the name is the guard name. Rule of Figure 10.5 states that for every pseudo state whose kind is "Initial", create an InitialNode in the output

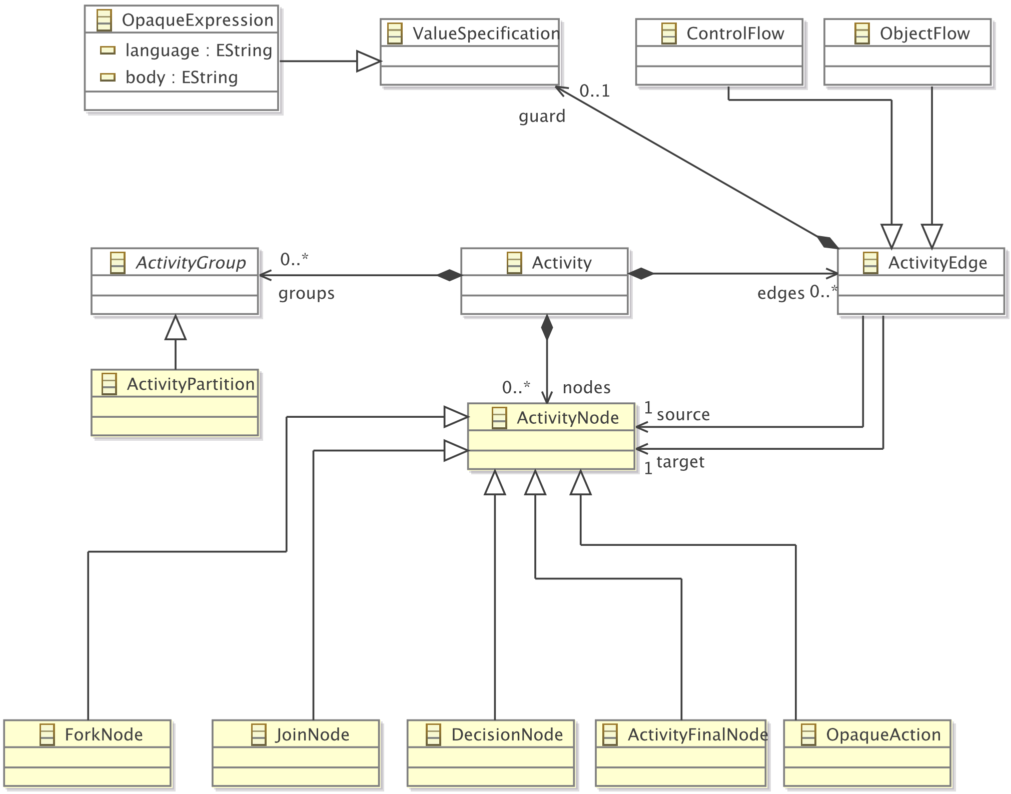


Figure 10.2: UML 2.2 Activity Diagrams (based on [**Rose2010_ModelMigrationCaseforTTC2010**]).

model. The three rules presented here pertain to the first layer.

The Rule in Figure 10.6 shows how the ActivityPartitions created in the first layer are connected to the Activities and Rule in Figure 10.7 shows how OpaqueExpressions, created from Guards in the first layer, are connected to control flows.

In addition to the DSLTrans transformation, we also build one in TrNet to compare how fast a TrNet transformation generated from DSLTrans is in comparison to one that was built "by-hand" in TrNet.

10.2 Methodology

For this experiment we used the only available machine we had:

Processor - Intel Core i5 2.4 GHz, 256 KB L2 and 3 MB L3;

Memory - 4 GB 1333 MHz DDR3

Operating System - OS X Version 10.8.4

We installed and configured Epsilon Flock, ATL and GrGen.NET.

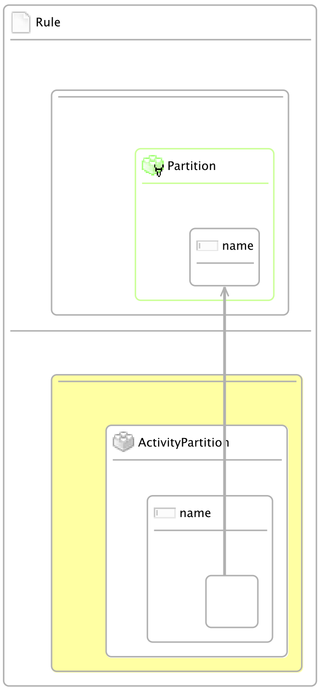


Figure 10.3: Activity migration transformation rule.

In order to reduce as much operating system interference as possible we ran the experiments with the minimal software process to ensure that the transformations could be executed and that the results could be collected.

We build a simple procedure to generate four random activity diagram models conforming to the UML 1.4 metamodel: one with a N of 10, other of 100, 1000 and 10000. The parameter N refers to a measure of complexity of the model. Note that a model with $N = 10$ will have much more than 10 elements. N is not the size. Each tool we used in the experiment will transform the same exact four input models.

For each tool, and each input model size, we ran the transformation ten times and the resulting measurement was calculated by average taking out the worst and the best times, to account for initialization delays and just-in-time compilations and other external interferences.

10.3 Results

After several measurements we plotted the chart that is shown in figure 10.8. Notice that, for clarity reasons, we use a logarithmic scale in both axes. The vertical axis denotes transformation times in milliseconds and the horizontal denotes the parameter N used to create the four random models used in this experiment. Each line is the execution time of an activity model migration transformation expressed in some language. We ran

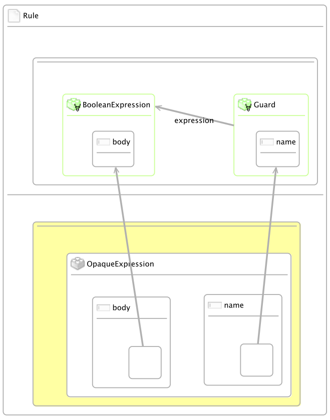


Figure 10.4: Activity migration transformation rule.

transformations in the following languages: Epsilon Flock; Atlas Transformation Language (ATL); GrGen.NET; a transformation manually crafted in TrNet and compiled to Java (without optimizations); a transformation expressed in DSLTRans, then compiled to TrNet and then compiled to Java without optimizations[1]; then the same transformation, but compiled with all optimizations turned on.

10.4 Discussion

It is important to notice the conditions on which this experiment was performed: the only available machine to us is a MAC OS X, which means that the JVM used was Apple's JVM. Apple's JVM performance is somewhat worse than Oracle's JVM so if this benchmark was run in a Windows machines, chances are that the executions times would be better. Nevertheless, all the tools that need JVM to run (ATL, Flock, TrNet and DSLTrans) share the same penalty.

After running the experiment, it is safe to say that all tools perform really well in terms of execution times but not so much in terms of memory consumption. For instance, we were unable to run the experiment with ATL and model with $N = 10000$ because there was not enough memory. Similarly, we were unable to run the experiment with a model of $N = 100000$.

The results confirm that the decision to make TrNet a compiled language was the right one. ATL, Flock and GrGen are interpreted languages and the transformations that were compiled to TrNet are faster than all of them by a large factor (note the logarithmic scale). In general, compiled languages are faster than interpreted ones, albeit for development purposes, an interpreted language is more productive.

The execution times of the DSLTrans transformation compiled to TrNet and then compiled to Java with and without optimizations show the real impact of the optimizations: the execution times were reduced to almost half.

[1] Only the execution plan inference was used.

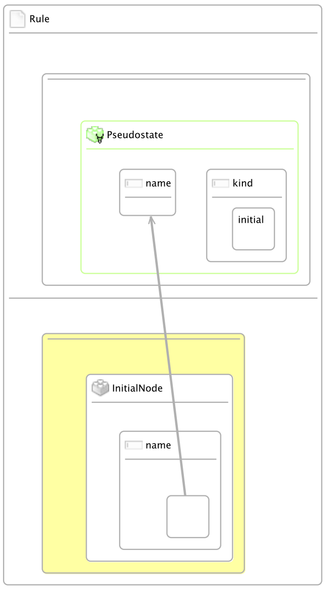


Figure 10.5: Activity migration transformation rule.

Comparing the transformation crafted manually in TrNet with the same transformation translated from DSLTrans, we confirm that transformations expressed in a low level language, were you can control most of the transformation process, are indeed faster, even with optimizations. But there is something curious: with a model with $N = 10000$ the execution time of the DSLTrans transformation is better than the execution time of the manually crafted transformation. This is explained by the fact that the transformation translated from DSLTrans being simpler, albeit more verbose and extensive, than the one we created manually. In the manually crafted transformation we used conditions and we took advantage of the hierarchy between metamodel elements to produce a more compact transformation. This means more conciseness but it makes the life of the optimizer more difficult. You may notice that we did not included the execution times of the manually TrNet transformation with optimizations turned on. That is because the impact of the optimizer was little to none. Also, the statistical analysis of the transformation generated from DSLTrans was more successful because of the simpler structure. If you observe figures 10.9 and 10.10, you will notice that, the ratio between correct estimates and total number of patterns is better in the transformation generated from DSLTrans than for the manually crafted transformation. This means that it is likely that, during transformation execution, fewer memory reallocation occurred. Also, by inspecting figures 10.9 and 10.10 you can see that no pattern need to allocate space for 10000 elements in the transformation generated from DSLTrans whereas many patterns need to allocate this chunk of memory in the manually crafted transformation. This means that the (re-)allocation operations, in addition to being more frequent in the manually crafted transformation, are also more expensive in terms of execution. All these factors combined yield a better execution time for models with $N = 10000$. Unfortunately, due to lack of memory we

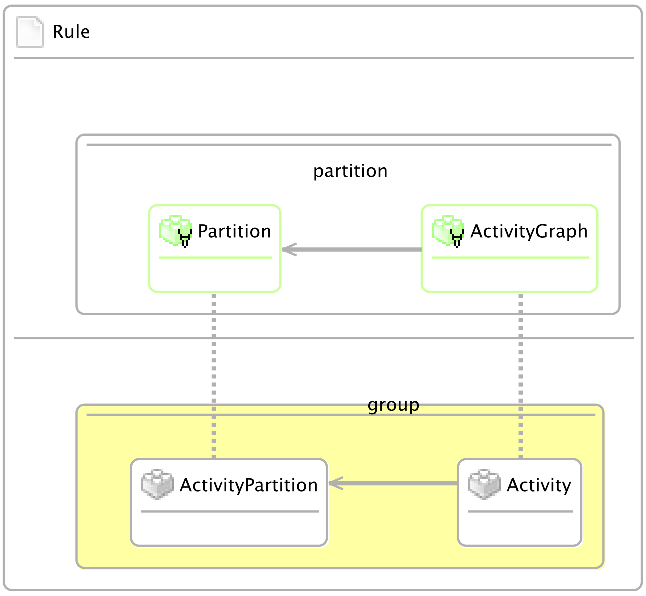


Figure 10.6: Activity migration transformation rule.

were unable to confirm this trend for models with $N = 100000$.

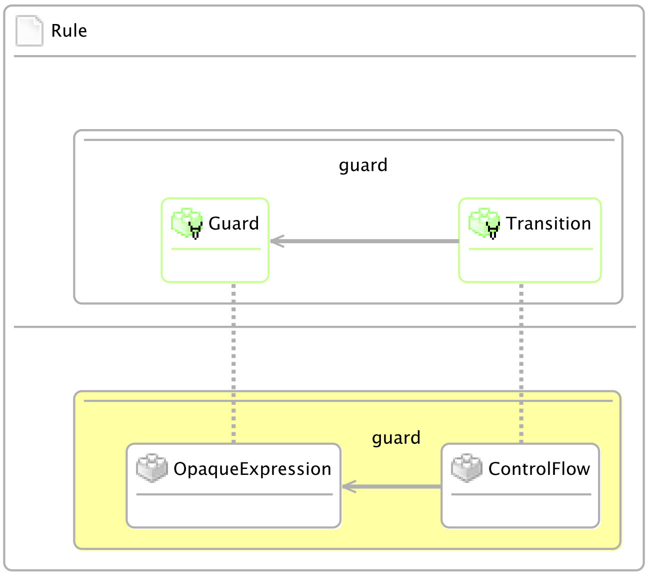


Figure 10.7: Activity migration transformation rule.

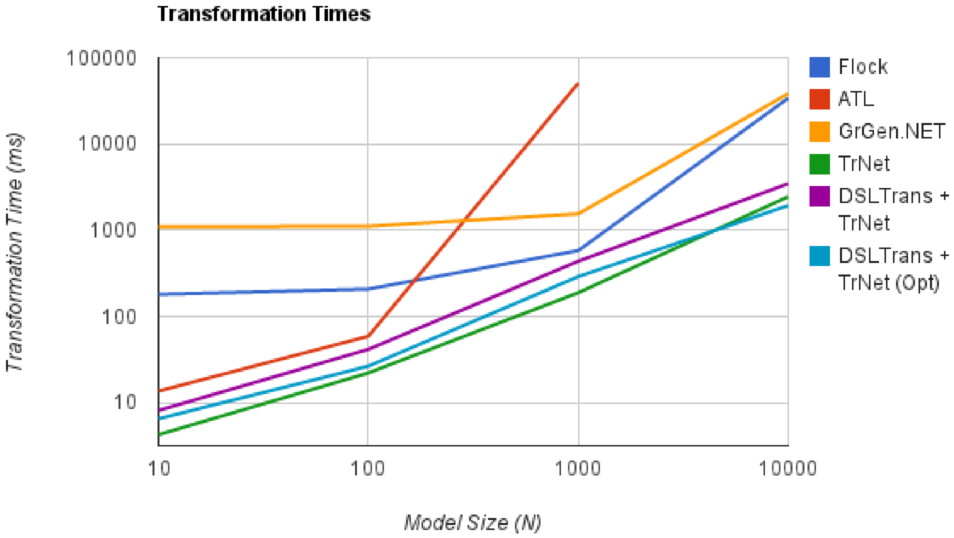


Figure 10.8: Benchmark results.

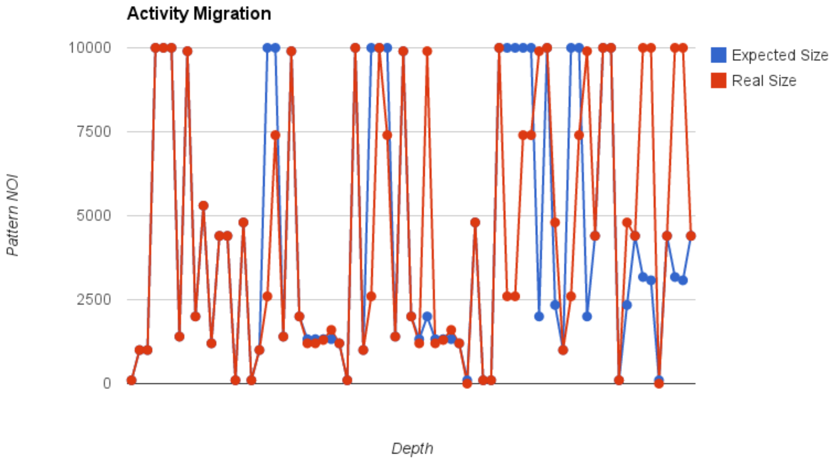


Figure 10.9: Estimated NOI for the activity migration transformation created by-hand.

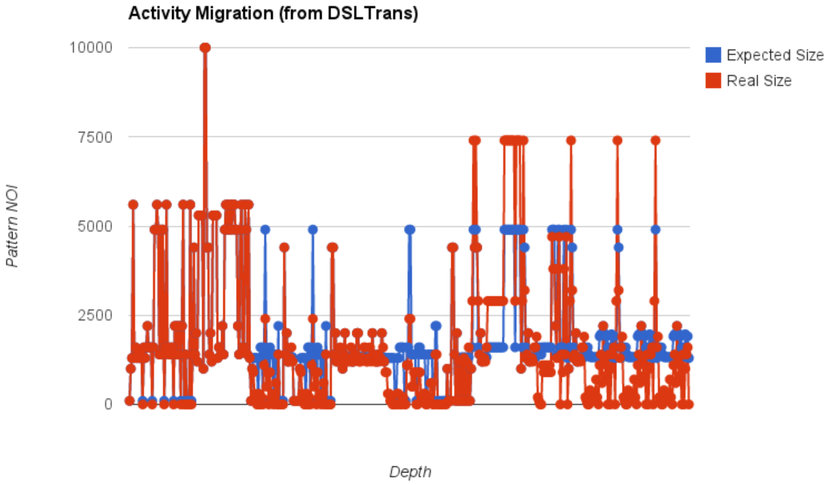


Figure 10.10: Estimated NOI for the activity migration transformation created by-hand.

11

Conclusion

This thesis addresses the abstraction penalty problem in the model transformation domain: how can a high level model transformation language be executed fast enough to be applicable in industrial scenarios?

Our approach is based on the widely used solution for the general purpose languages domain: an high level program is compiled into a lower level one. During the compilation process, intermediate representations of the same program are used to facilitate optimizations.

In our case, we studied the state of the art optimization techniques and classified them according to where they take place in the transformation process, their impact and the information required. Armed with that knowledge, we designed an intermediate representation for model transformations that facilitates the application of those techniques: TrNet. Furthermore, we developed optimization on top of that intermediate language, including one of the most advanced techniques: overlapped pattern matching.

In order to measure the impact of the optimizations in a typical TrNet transformation we ran a benchmarks. Besides comparing the unoptimized TrNet transformation with the optimized version of the same transformation, we collected the running times of other state of the art model transformation tools.

The experiment showed that not TrNet is indeed faster than the other tools, but also that the optimizations that we developed, and in particular overlapped pattern matching, have a great impact in TrNet transformations that are generated from higher level transformations such as DSLTRans'.

This already shows that our approach can help to mitigate the abstraction penalty. Since the optimizations have greater impact in generated transformations, it pays off to

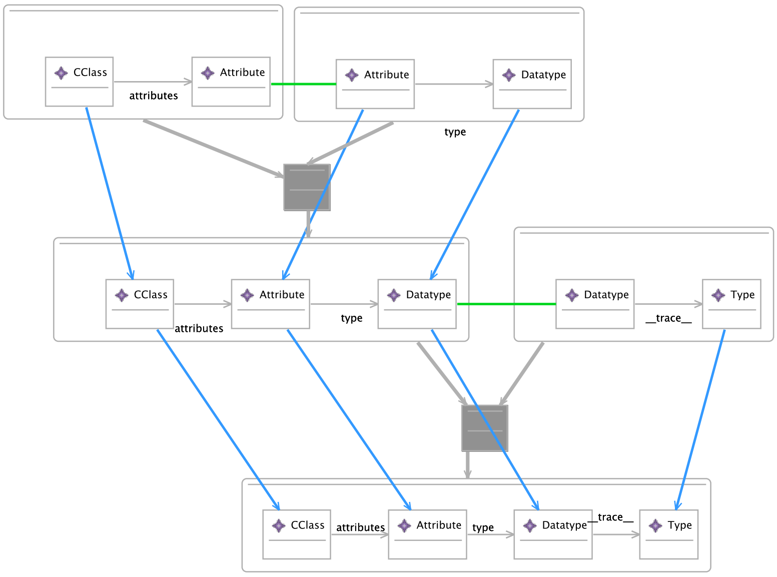


Figure 11.1: Example of a sequential join that creates a more complex pattern.

use an high level model transformation language to develop transformations in a productive fashion, then compile them to TrNet transformations where they are optimized and then compiled to java.

Due to the runtime architecture, TrNet transformations can work independently of the model management framework. In this thesis we developed the integration with the EMF but other could be used.

11.1 Future Work

In the benchmark we ran, we only collected running time statistics but we notice that most tools, including TrNet, require a lot of memory to transform large input models. Unfortunately, we did not have the time to collect memory consumption statistics for the tools but, at least for TrNet, due to its architecture, there should be more investigation in novel memory optimization techniques. A possible direction is to try to aggregate two or more sequential operators such that their execution can be in stream mode and thus, not requiring the intermediate storage of elements in data structures.

Other possible open challenge is to extend the join order optimization technique to detect sequential joins in the network, such as the one shown in figure 11.1, where no new elements are created, evaluate their total cost and try several other arrangements in order to find the cheapest one, for instance, figure 11.2.

Other high level language should be selected and a compiler to TrNet should be produced. Then, the same kind of experiments developed here should be made to compare the TrNet transformation execution with the traditional execution of that high level language.

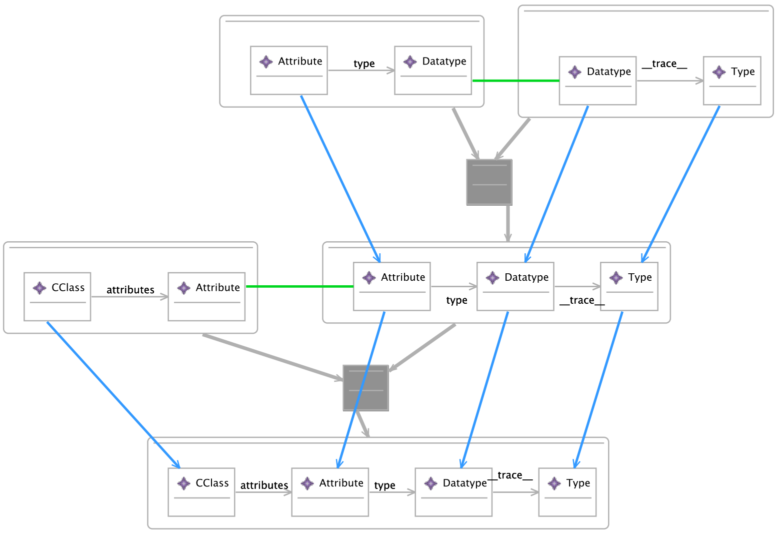


Figure 11.2: Example of a sequential join that creates a more complex pattern equivalent to the one shown in Figure 11.1.