Master's Thesis

STATIC ANALYSIS OF MODEL TRANSFORMATIONS

by
Zoltán Ujhelyi

Budapest, May 2009.

Department of Measurement and Information Systems
Budapest University of Technology and Economics

Supervisors
Ákos Horváth, PhD student
Dr. Dániel Varró, assistant professor

I would like to thank my supervisors Ákos Horváth and Dr. Dániel Varró for their continued support, friendly advice, and enthusiasm. I would also like to thank Gábor Bergmann and Zoltán Bai for reading the earlier versions of my thesis, and I very much appreciate the valuable comments of Balázs Grill and Róbert Kovács.

Abstract

Nowadays the Model Driven Software Development gains more and more weight in software development. Its most important concept is the definition of a high level Platform Independent Model that is transformated into the application using several Model Transformation (MT) steps. So it becomes crucial that development tools provide support for writing Model Transformations.

The VIATRA2 transformation framework developed at the Department of Measurement and Information Systems is such a supporting sytem, and is used in various research projects. The framework uses a high level language combining elements of model transformations and abstract state machines (ASM). For the elementary transformation steps of graph models graph transformation rules are used, while using ASM contructs these steps can be built into a complex transformation program.

This thesis introduces a static source code analyser system usable for graph transformation languages. The static source code analysers identify potential faults without running the program using only the source code structure and the syntactics and semantics of the language, thus speeding up development.

The analysis is traced back to solving Constraint Satisfaction Problems (CSP). The CSP solver systems use a declarative approach of solving combinatoric problems. They use a set of variables by narrowing their possible domain by applying constraints. If the domain of a variable becomes the empty set, the problems is not solvable. The main advantage of using CSP solvers is that they are capable of propagating the effects of the new constraints forwards and backwards so previous statements can be reevaluated.

After the general analysis the thesis describes a type safety checker for the transformation language of the VIATRA2 framework, and displays its result in Eclipse-based graphical user interface of the framework.

The thesis concludes with a case study based evaluation of the type checker, and the lists the conceptional and practical limitations.

Kivonat

Manapság a szoftverfejlesztés területén egyre többen sorakoznak fel az OMG Modellvezérelt Architektúra (MDA) kezdeményezése mellé, amelynek alapja, hogy egy magasszintű platform specifikus modellből kiindulva transzformációs lépések sorozatán keresztül ér el futatható alkalmazásig. A modellvezérelt fejlesztés sikerének egyik létfontosságú eleme a modelltranszformációk fejlesztésének (MT) hatékony támogatása.

A modelltranszformációk támogatására készült a Méréstechnika és Információs Rendszerek tanszéken fejlesztett és több kutatási projektben használt VIATRA2 keretrendszer. A keretrendszer a gráftranszformáció és az absztrakt állapotgépek magasszintű nyelveit ötvözi egy egységes formális specifikációs nyelvbe. A nyelvben a gráf alapú modellek elemi transzformációját gráftranszformációs szabályok végzik, míg az elemi lépésekből egy komplex transzformációs programot az absztrakt állapotgépek segítségével építhetünk fel.

A dolgozat egy modelltranszformációs rendszerekhez használható statikus kódellenőrző rendszer mutat be. A statikus forráskód elemzők célja, hogy a lehetséges hibákat a program futtatása nélkül, kizárólag a kód struktúrájából, illetve a nyelv szintaktikájából és szemantikájából felismerjék, gy gyorsítva a fejlesztés folyamatát.

Az ellenőrzést kényszerkielégítési probléma (Constraint Satisfaction Problem, CSP) megoldására vezeti vissza. A kényszer megoldó rendszerek deklaratív megközelítést adnak kombinatorikus problémák megoldására. Változók egy halmazán működnek, ezek lehetséges értékkészletét a kényszerek egymás után történő alkalmazásával próbálják folyamatosan szűkíteni. Ha valamelyik változónak az értékkészlete az üres halmazra csökken, akkor a problémának nincs megoldása. A CSP megoldó használatának legfontosabb előnye, hogy a kényszerek hatásait képes visszafele is terjeszteni és így korábbi megállapításokat is befolyásolni.

A dolgozat a VIATRA2 keretrendszer transzformációs nyelvéhez mutat be egy típushelyesség ellenőrzésére alkalmas eszközt, amely az előbb leírt alapokon működik, és az ellenőrzés eredményét a keretrendszer Eclipse alapú környezetébe integráltan jeleníti meg.

A dolgozat esettanulmánnyal igazolja a módszer gyakorlati alkalmazhatóságát és megállapítja a koncepcionális és gyakorlati korlátokat.

Contents

List of Figures

Chapter 1

Introduction

1.1 System Modeling Overview

Model Driven Software Development (MDSD) [33] tries to address the challenges of the ever changing environments by separating the business and application logic from the underlying platform technology. With this separation it is possible to maintain interoperability between different platforms.

One level of the MDSD models is the *Platform Independent Model* (PIM), that describes all the business functionality and behaviour, but completely lacks platform-specific details. The other level, the *Platform Specific Model* is a lower level model, it does include results of decisions the platform forced the programmer to make. And from the PSM the actual software application can be created.

The typical architecture of the MDSD based development cycle can be found in Figure 1.1. The process starts by creating PIM models - either from scratch or by reverse engineering an existing, legacy application without an existing model, and then by using model transformations and platform information we begin to produce PSM models, and finally the deployable application. To support such development cycles, a (semi-) automatic transformation methodology is used.

These transformations are basically mathematical operators, but in the special case of the MDSD process most of these models are graph-based structures so it is possible to use graph transformations instead of the more general model transformation formalisms. Informally, a *graph transformation* (GT) [40, 20] is a set of rules. These rules perform local manipulations on the graph by finding a pattern described by their *left hand side* (LHS), and altering the found part according to the *right hand side* (RHS) of the rule. In order to be able to control the transformation process more precisely, an additional *control structure* is used, which allows the description of a complex model transformation task as a series of simple graph transformation rules.

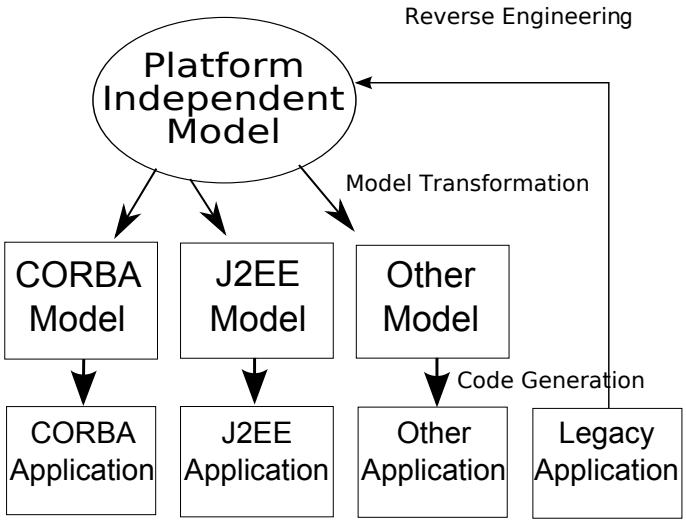


Figure 1.1: The Model Driven Architecture

1.2 Verification of Model Transformations

The definition of these complex transformation is similar to high level computer programs which means that they are also vulnerable to programming mistakes. As transformations become more complex, early fault detection is becoming a key question as faults in transformation programs can even propagate into the developed application.

On the other hand verification methods and tools researched for computer programs are also applicable for model transformation programs. These methods include different testing strategies, model checking and static analysis.

In general model transformation programs usually contain dynamic element creation and deletion that result in infinite state spaces which cannot be handled easily by model checkers [38]. As for testing transformation programs research is very lively in the area [29], but early results show that testing or comparing the outputs of transformation programs (which are usually models) is also time demanding, and hard to be used in larger scale. On the other hand the use of static analysis does not guarantee faultless programs, but in practice it highlights typical programming errors without executing the program itself.

The complexity of the static analysis depends on both the analyzed structure and the property to check. For example type safety checking is easier in Java (or in C#) as type information is available during the compilation process. But in case of dynamic languages, such as Javascript or Prolog a type checker has to infer the types of the variables and constants. Although this is less accurate than explicitly stated type information, it is still useful for error detection.

A relatively common fault (especially in dynamic languages) is the incorrect use of types that does result in a misleading output rather than a runtime exception making it hard to trace. The aim of static type analysis is to help the developer by detecting these faults.

Most static analyser tools are reading the program only once, and calculating the results on-the-fly. This allows the reduction of the memory footprint, but limits the capabilities of the analysis: if at a point we obtain some information about an already processed element, this information might not be propagated to some dependant elements.

An idea to overcome this challenge is the use of CSP (Constraint Satisfaction Problem) solvers, that are based on propagation. The conditions represented by the elements of the language can be represented as constraints in a natural way, and the solver is capable of handling these distinct constraints as a whole system.

1.3 Research Objectives

The main goal of my research is to design and implement a *static analysis* framework based on Constraint Satisfaction Programming capable of identifying different kinds of faults in transformation programs. To demonstrate the capabilities of this framework I implement a *static type checker* tool, and integrate it into the VIATRA2 model transformation framework.

Such a static analyser could help model transformation developers to enhance the quality of their transformation code by detecting faults early when its inexpensive to repair. By the integration into the model transformation IDE (Integrated Development Environment) the tool could be used to generate feedback by marking the erroneous elements directly in the editor.

The feedback should be instantly to allow the transformation developer to check any newly written code as soon as possible, so the analysis should finish in a reasonable amount of time.

My detailed objectives are:

- to describe the transformation programs in a general *Transformation Program Model*;
- to design a method of mapping the model built of *control and GT rules* into constraint satisfaction problems;
- to define an extensible list of *fault patterns*, that describe how to identify the faults;
- to propose a way of describing the *properties of the metamodel* inside a Constraint Satisfaction Problem;

- to integrate the implemented static analyser tool together with a constraint solver framework into the VIATRA2 model transformation framework. This integration should be connected to both the model space and the user interface of the framework.

1.4 Overview of the Approach

Figure 1.2 displays the position of the planned system.

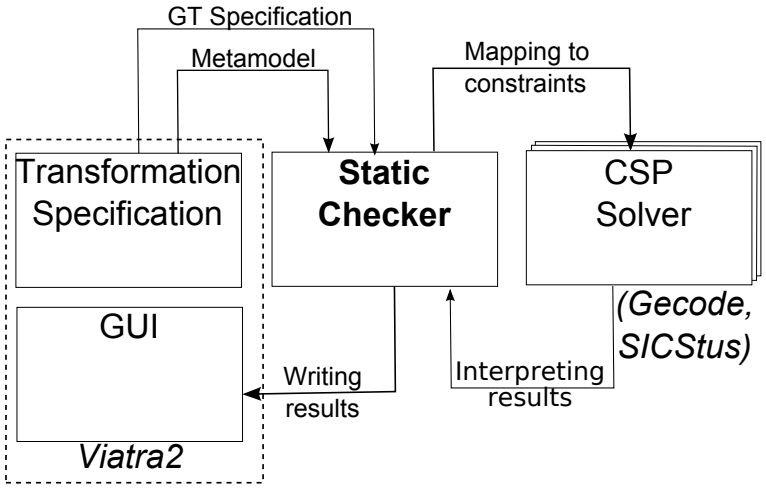


Figure 1.2: The Architecture of the Static Checker System

The proposed static checking component is positioned between a model transformation framework (in our implementation the VIATRA2 framework) and a CSP solver (after the examination of the Gecode [22] and the `clpfd` module of SICStus Prolog [13] solver a third new solver has been implemented specifically for this analyser).

The process starts with the extraction of the metamodel and the specification, that forms the basis of a Transformation Program Model. The model is traversed and gets translated into constraints for the CSP solver. The output of the solver should be displayed in the user interface of the model transformation framework.

As of most CSP solver frameworks are not capable of determining which constraints are responsible for a failure our static checker component should be able to deduce some information about the source of the error.

Two different constraint solvers were evaluated during the implementation period: the Gecode/J constraint solver and the `clpfd` module of SICStus Prolog. After hitting some performance bottlenecks we implemented a new solver tailored to the needs of the analysis.

1.5 The Structure of the Thesis

The following chapters describe this system in details, as detailed below:

- Chapter 2 gives an overview of the concepts used in the thesis. First it introduces modeling and metamodeling both in general and specific to the VIATRA2 framework, then the subjects of static analysis, type checking and constraint satisfaction problems are described.
- Chapter 3 describes the *Transformation Program Model*, a generic model representing the transformation programs, then describes how to use this model as the input of a static analysis process. The method described in this chapter is general: it can be used to check any property of the model.
- Chapter 4 defines a *type checker* tool based on the generic method from Chapter 3. To achieve this, first it describes how to represent the properties of the metamodel in a constraint satisfaction problem, then describe on a per-node basis how to generate the relevant constraints and how to represent the control flow during the analysis. Finally the Chapter describes how to interpret the results of the constraint solver to identify faults.
- Chapter 5 details how the created type checker detects faults on larger examples, and investigates its performance based on the results of some simple measurements.
- And finally Chapter 6 concludes the results of my work and presents some research directions for the future.

Chapter 2

Background technologies and concepts

This section briefly introduces the main notions used in this thesis. The first section introduces the concepts of models, metamodels, and model transformations together with the VIATRA2 transformation framework following the structure of [23], while the model transformation examples are taken from [9]. Then a brief overview is given of the static analysis methods and the constraint satisfaction problems.

2.1 Models and Transformations

A possible way to describe complex transformations is to use *graph transformation* (GT) [20] rules for local model manipulations and use *abstract state machine* (ASM) [11] rules to define control structure. A promising way to define conditions in GT Rules is the use of *graph patterns* (GP). This approach is used in VIATRA2.

VIATRA2 (VIsual and Automated TRAnsformations) [7] is a model transformation framework developed at the Department of Measurement and Information Systems. It stores models and transformations in a graph based style, but it is also capable of parsing the models and transformations from textual files.

To parse models (and metamodels) the *Visual Textual Modeling Language* is used. It contains element declarations defining the model space (or a part of the model space).

The *Viatra Textual Command Language* (VTCL) is used for defining transformations. Transformations are represented as *ASM Machines*, which consists of *Graph Patterns*, *GT Rules* and *ASM Rules*. The inner representation of transformation programs in VIATRA2 are stored as an EMF [1] model on which the interpreter works. This inner representation is also available through VIATRA2 Core Interfaces.

In this section we give a brief introduction of the VTCL language along with

the related concepts while a complete specification of both languages can be found in [6].

2.1.1 Metamodeling

Metamodeling provides a structural definition (i.e. abstract syntax) of modeling languages. Such a definition is needed to define the input and output of model transformations. Currently the most widely used metamodeling languages (e.g. ECore [1]) are based on the OMG metamodeling standard MOF (Meta Object Facility)[34].

However as stated in [43] MOF fails to support multi-level metamodeling, therefore some other approaches are also used: for instance the VPM (Visual and Precise Metamodeling) makes multi-level metamodeling available by the use of explicit and generalized *instanceOf* relation. These relations allow to store a model and it's metamodel in the same model space. The VPM concept also has a mathematically precise notation.

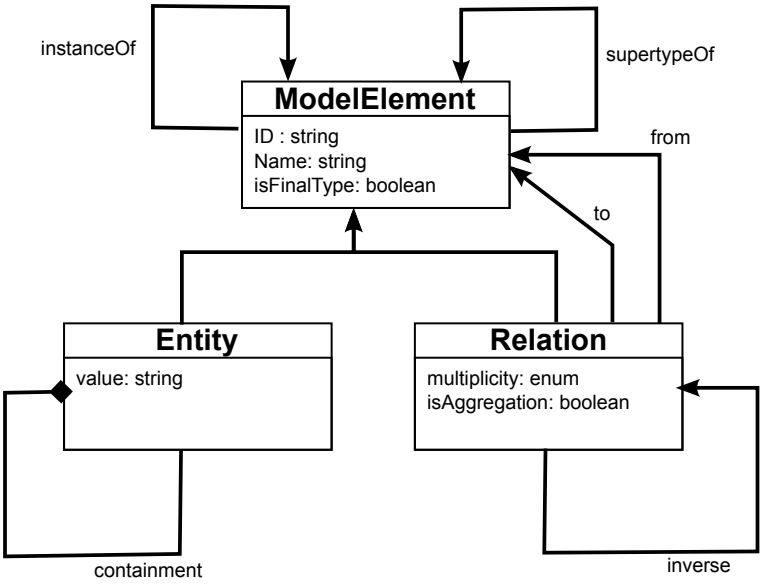


Figure 2.1: The VPM Metamodel

The VPM Metamodel which is showed in Figure 2.1 contains two different types of model elements: the *Entity* and the *Relation*.

A VPM Entity represents the basic concepts of the (modeling) domain. An entity can be considered as a common term for MOF packages, classes and objects. For each Entity it is possible to hold an application-specific associated string *value*.

A VPM Relation represents a general relationship between the Entities. They can be considered as MOF associations, basically one-to-many relations.

The built-in relations are the `instanceOf` and the `supertypeOf` (and their inverses, accordingly `typeOf` and `subtypeOf`). The `instanceOf` relation is used to explicitly describe the connection between the model and the metamodel, while the `supertypeOf` relation represents a binary superclass-subclass relation (similar to the concept of generalisation in UML). The VPM metamodel also describes the `containment` relation, which arranges the model elements into a strict containment hierarchy.

The semantics of these built-in relations include three formal transitivity rules (which are similar to the rules of UML):

$$instanceOf(a,b) \wedge subtypeOf(b,c) \Rightarrow instanceOf(a,c)$$
$$subtypeOf(a,b) \wedge subtypeOf(b,c) \Rightarrow subtypeOf(a,c)$$
$$instanceOf(a,b) \wedge instanceOf(b,c) \not\Rightarrow instanceof(a,c)$$

The concept of `instanceOf` relation is different from the concept of *instance model*. An instance model is a well-formed instance of the metamodel, while the relation describes the connection between a model element and its corresponding metamodel element.

The relations' multiplicity property imposes restrictions on the model space. These constraints are used by the pattern matcher. The allowed multiplicity values are `one-to-one`, `one-to-many`, `many-to-one` and `many-to-many`.

Example metamodel: Petri nets

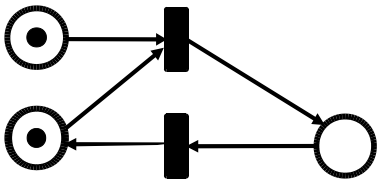


Figure 2.2: A Simple Petri net model

As an illustration of metamodeling we introduce the metamodel of Petri nets. Petri nets (a simple net can be seen in Figure 2.2) are a formal description for modeling concurrent systems. It is widely used because of the easy-to-understand visual notation and large number of available editor and analysis tools.

Throughout the paper we will use Petri nets as an example domain to illustrate the technicalities and foundations of our approach.

The Petri nets are bipartite graphs with two disjoint set of nodes: *Places* and *Transitions*. Places can contain an arbitrary number of *Tokens*, and the distribution of these Tokens represent the state of the net (marking). This state can be changed by a process called firing.

A typical graphical representation of the metamodel is depicted in Figure 2.3.

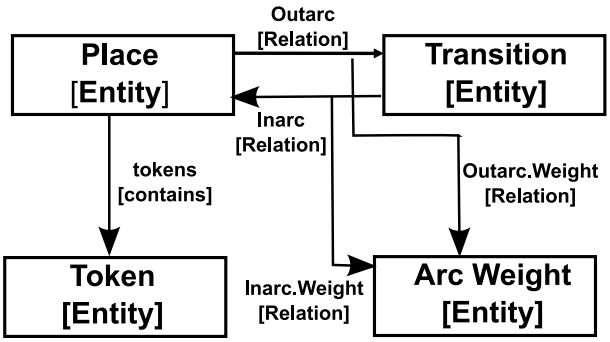


Figure 2.3: The graphical representation of the Petri net metamodel

2.1.2 Graph Patterns

Graph patterns are the atomic units of graph transformations. They represent a condition (or possibly constraints) which has to be fulfilled by a part of the model space. Graph patterns are used in transformation rules as conditions and as a description of the result pattern.

A model (a part of the model space) matches a pattern, if the pattern can be matched to a subgraph of the model using a generalised *graph pattern matching* technique. Basically this means each occurrence of the pattern is a mapping of the pattern variables to the model elements in a way to satisfy all conditions of the pattern - this is a subgraph isomorphism problem.

It is possible to write both positive and negative patterns: the positive pattern holds if the all conditions hold, while if a negative pattern condition can be satisfied, the pattern will fail. Both positive and negative patterns can be nested in an arbitrary depth [37].

Example 1 *As an example of graph patterns we describe the pattern of the fireable transitions over the metamodel defined before. A graphical representation of the pattern can be seen in Figure 2.4.*

The pattern represents, that a Transition is fireable if it is not connected *to a Place by an* Outarc*, where the Place has* no tokens*. The pattern is described in the VTCL language in Listing 2.1. The structure of the two representations are similar.*

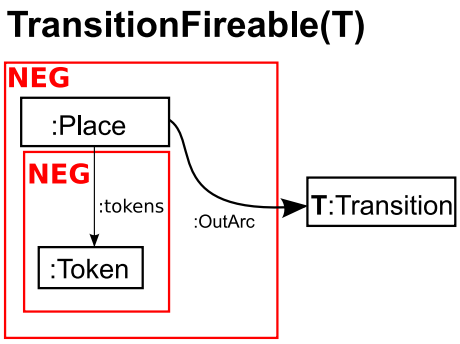


Figure 2.4: The Transition Fireable Graph Pattern

Listing 2.1 The Transition Fireable Graph Pattern in the VTCL language

```
pattern TransitionFireable(Transition) =
{
	'PetriNet'.'Transition'(Transition);
	neg pattern notFireable(Transition) =
	{
		'PetriNet'.'Place'(Place);
		'PetriNet'.'Transition'(Transition);
		'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
		neg pattern placeToken(Place) =
		{
			'PetriNet'.'Place'(Place);
			'PetriNet'.'Place'.'Token'(Token);
			'PetriNet'.'Place'.'tokens'(X, Place, Token);
		}
	}
}
```

The `pattern` keyword is used to define a pattern (or `neg pattern` in case of a negative pattern), and in parentheses are the parameters defined. To express a type constraint on a variable, the name of the metamodel type is used with the name of the variable in parentheses (e.g. line 3 of Listing 2.1).

It is possible to add further conditions to the patterns by the use of the `check` keyword, which allows the checking of a boolean formula.

To allow the description of more complex patterns, in the VTCL language it is possible to call other patterns with the `find` keyword. This also enables reusing existing patterns. The semantic of the construct is similar to the clauses in Prolog: the caller pattern is fulfilled if and only if all the called subpatterns are fulfilled.

Alternate patterns are also available in the language: a pattern can have multiple bodies by connecting them with the `or` keyword. When several alternative patterns are defined, the pattern is fulfilled if any of the bodies can be fulfilled. This semantic is also similar to the Prolog clauses.

Pattern calls and alternate patterns together can be used to define *recursive patterns*. Recursion is typically used with two pattern bodies: one which have a call to itself, while the other defines the condition to stop the recursion.

2.1.3 Graph Transformation Rules

For defining a graph transformations *Graph Transformation Rules* (GT Rule) are used. These rules rely on the Graph Patterns as defining the application criteria for the steps. A GT Rule application transforms a graph by replacing a part of it with another graph.

In order to describe GT Rules *preconditions* (also known as the Left Hand Side graph, LHS) and *postconditions* (also known as the Right Hand Side graph, RHS) are defined, where the precondition acts both as application criteria and the part of the model to change, while the postcondition describes how the match will look like after the rule application. The required changes can be computed by calculating the difference between the precondition and postcondition patterns that can be interpreted as a series of model manipulation steps.

Example 2 *Figure 2.5 shows the graphical representation of two transformation rules related to firing a transition. We describe the meaning of the* `addToken` *rule in details, the* `removeToken` *rule can be interpreted similarly.*

The LHS graph pattern of the transformation consists of a `Transition` *(called* `T`*) and a* `Place` *(called* `P`*) connected with an* `InArc` *relation while the RHS pattern adds an unnamed* `Token` *element and a* `tokens` *relation between the* `Place` *and the new* `Token`*. That means, after the execution of the rule a new* `Token` *is created an assigned to a* `Place`*.*

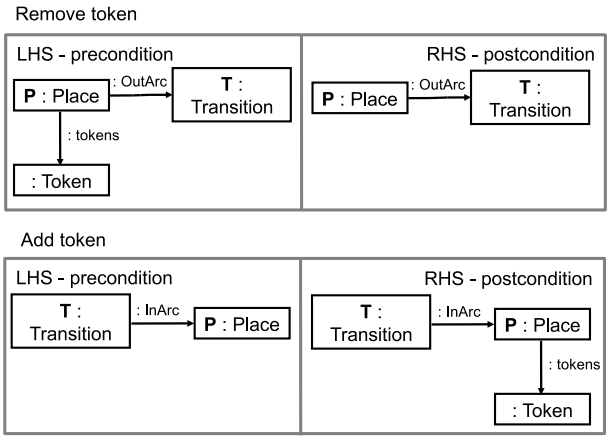


Figure 2.5: Graphical Representation of Graph Transformation Rules

In the VTCL language the GT Rules are marked with the `gtrule` keyword, and the definition can have parameters. These parameters have to be marked with the `in`, `out` or `inout` to mark whether they can be changed during the rule execution.

To attach the LHS and RHS patterns to the rule, they have to be entered with the keywords `precondition` and `postcondition`.

Example 3 *To illustrate the capabilities of the GT Rules description, in Listing 2.2 we include the code of the* ***`addToken`*** *rule in VTCL.*

Listing 2.2 The addToken GT Rule in the VTCL language

```
// Adds a token to the place 'Place'.
gtrule addToken(in Place) =
{
	precondition find place(Place)
	postcondition find placeWithToken(Place, Token)
}

pattern placeWithToken(Place, Token) =
{
	'PetriNet'.'Place'(Place);
	'PetriNet'.'Place'.'Token'(Token);
	'PetriNet'.'Place'.'tokens'(X, Place, Token);
}

pattern place(Place) =
{
	'PetriNet'.'Place'(Place);
}
```

In a common transformation rule the LHS and RHS graphs are nearly the same (typically only a part of the match changes in a rule application). In order

to avoid the need for two graphs, there is an alternate notation for describing a transformation. The FUJABA [32] notation annotates the graph with the keywords `new` and `delete`, where the keywords mean, that during the execution a new element is added, or a found element is deleted. The LHS and RHS graphs can be created from this notation: those elements (either entities or relations), which are not tagged with either of the keywords, are members of both graphs, while elements tagged with the `new` keyword, are only elements of the RHS graph, and elements tagged with the `delete` keyword, are only part of the LHS side.

A similar construct is also available in the VTCL language: after the precondition pattern instead of a postcondition pattern a sequence of ASM rules (*actions*) can be defined. For ASM rules see Section 2.1.4.

It is important to note, that actions can also entered when using a postcondition pattern: the typical usages are debugging and code generation.

In the VTCL language actions can be entered using the `action` keyword inside a GT rule.

Example 4 *The* ***`addToken`*** *rule can be written using the FUJABA notation as in Listing 2.3.*

Listing 2.3 The addToken GT Rule - FUJABA notation

```
// Adds a token to the place 'Place'.
gtrule addToken(in Place) = {
	precondition find place(Place)
	action {
		new('PetriNet'.'Place'.'Token'(NewToken) in Place);
		new('PetriNet'.'Place'.'tokens'(Temp,Place,NewToken));
	}
}

pattern place(Place) =
{
	'PetriNet'.'Place'(Place);
}
```

The interpreter of the VIATRA2 framework supports these formats simultaneously, so developers can choose between the notation that is more suitable for them.

2.1.4 ASM Rules

To allow the construction of complex model transformations, the assembly of the elementary GT rules into transformation programs is required. The VTCL language uses abstract state machine [11] rules to describe the control structure.

In order to semantically integrate the GT and ASM concepts, GT rules are treated the same as ASM rules (the `apply` construct can be used for calling both rule types) and graph patterns can be used as existentially qualified Boolean formulae in ASM conditions (by the `find` construct).

In the VTCL language an ASM Rule is described as a block marked with the `rule` keyword, with parameters. The parameters direction has to be marked similarly as of the GT Rules. An ASM Rule has to be a single ASM language construct, if multiple elements are needed, some compound rule has to be used.

The basic elements of the ASM programs are:

ASM Rules are alike methods in OO languages, they have input parameters, and represent a set of operations. In the language there are some built-in rules, and it is also possible to define new ones.

ASM Variables are similar to the variables (attributes) in OO languages, they hold values (model element references, constants, etc.).

ASM Functions are special mathematical functions, which store variables in arrays. Associative arrays in modern programming languages (e.g. in Java the `Map`) provide analogous services.

The ASM Rules are used to call GT Rules: the *apply* rule can be used with bound parameters, while the *choose* and *forall* rules with free parameters (thus allowing searching for patterns). The *choose* quantifies the unbound parameters existentially while the *forall* universally.

There are also constructs for affecting the control flow: the *iterate* rule executes a single rule repeatedly, the *conditional* rule defines a binary branch in the control flow (similar to the if-then-else constructs in OOP languages),

The *random*, the *parallel* and the *sequential* rules are used to creating compound rules. The *random* rule executes one of it's nested rules, the *parallel* executes all the nested rules simultaneously, while the *sequential* one by one.

By using a *try* rule, it is possible to detect failures and take the control. A failure can be caused by the *fail* rule or a *choose* rule which cannot find a match in the model space.

Example 5 *The transformation program in Listing 2.4 executes a number of firings defined by the input parameters.*

The program contains the `main` *and the* `fireTransition` *ASM Rules. The execution starts with the* `main` *rule, where the* `Iterations` *and the* `Net` *input parameter define the number of firings and the Petri net, respectively. First, the rule creates and initializes the variable* `Start` *to zero, then the* `iterations` *and* `firings` *values of the ASM Function* `counter` *are also initialized followed by the saving of the current system to the* `Start` *variable.*

Listing 2.4 A Simple ASM Rule Describing the Firing of Transitions

```
// fires the input transition
// @Transition: the transition to fire (input)
rule fireTransition(in Transition) = seq
{  //deletes the tokens from the input places
  forall Place with find sourcePlace(Transition, Place) do
    choose Token with find placeWithToken(Place,Token) do delete(Token);
  //adds the new tokes to the output places
  forall Place with find targetPlace(Transition, Place) do seq
  {
    new('PetriNet'.'Place'.'Token'(Token) in Place);
    new('PetriNet'.'Place'.'tokens'(X, Place, Token));
  }
}

asmfunction counter/1;

// entry point
// @Net: the container entity of the Petri net model to be simulated
// @Iterations: number of firings to be executed
rule main(in Net, in Iterations) = let Start = 0 in seq
{
  update counter("iterations") = 0;
  update counter("firings") = 0;
  update Start = systime();
  iterate seq
  {
    update counter("iterations") = counter("iterations") + 1;
    if (counter("iterations") > Iterations) fail;
    choose T with find fireable(Net,T) do call fireTransition(T);
  }

  println("Simulation ended, fired " + counter("firings") + " transitions in " +
    (counter("iterations")-1) + " iterations in " + (systime()-Start)+ " msec.");
}
```

The `iterate` *structure controls the number of firings as it is invoked as many times as the* `if` *condition becomes false, because a* `fail` *construct exits from the container* `iterate` *structure. The* `update` *rule increases the* `iterations` *value of the* `counter` *ASM function, while the* `choose` *rule matches to a fireable* `T` *transition and calls the* `fireTransition` *ASM Rule. It is also important to mention that* `choose` *rules can also fail, so the* `iterate` *cycle in the program can also be stopped by an unsuccessful match meaning that there are no fireable transitions in the net.*

As for the `fireTransition` *rule, it matches to all source places of the input* `Transition` *parameter with the first* `forall` *rule and deletes a Token from each place, while the second* `forall` *rule generates a* `Token` *for all target places with the two* `new` *rules.*

After the `iterate` *cycle terminates the* `println` *rule prints out the number of firings, iterations and execution time to the output.*

2.2 Static analysis and type inference

It is a known fact in computer programming that the sooner an error is detected, the cheaper it is to correct - if the error remains undetected during a design phase, the repair cost might increase with an order of magnitude. Basically there are three ways of ensuring the correct behaviour of a computer system: either testing the running system, or proving the correctness of the constructs used to build the system - without executing the system. This second process is called static analysis. The third way is the use of model checking: it is used to decide whether the structure is the model of a logical formula.

Exhaustive testing becomes impossible with the growth of the systems, because the number of test cases is growing exponentially. The only possible solution is to choose test inputs in a rational way that would possibly detect the most common failures [10]. Alltogether testing is very expensive. Even worse, it is easy to omit a test case by mistake, which makes the process error-prone, and testing only shows the presence of faults, but cannot prove their absence.

The model checking [16] method requires the traversal of the model space, thus it is vulnerable to state explosion problem. Another drawback of the method is that it cannot always decide the neither presence nor their absence of faults, so the output of the analyser is three kind of answers are possible: the checked property holds, it does not hold, or it can't be decided.

The main promise of *static analysis* is that it is capable of detecting a predefined set of faults without even starting the application. In practice only some fixed kind of faults can be found with static analysis, but for this limited fault model a good analyser may prove that none of these errors are present in the program.

The compilers of the statically bound languages, like Java or C# include some kind of static analysis: during compilation they determine the types of the variables, watch for uncatched exceptions, etc. These verifications are performed during the compile time thus helping the early identification of some common problems.

Something very similar is possible for other structures, e.g. Petri nets. It is possible to check the P- and T-invariants of the net [31], which might be used to detect some serious modeling errors - without the expensive calculation of the state space of the net.

A static analysis is carried out by an *abstract interpretation* [17] of the program, and the description of the computation in this abstract universe. The execution of this abstract computation might offer some information about the actual computation.

Example 6 *A typical example for abstract interpretations is the rule of signs. In this case we are denoting the (integer) numbers on the abstract universe of* $\{(+), (-), (\pm)\}$. *In this example from the calculation* $-1517 \cdot 17$ *becomes* $(-) \cdot (+) =$

$(-)$, *and the properties of transformation proves that the actual result will be a negative number.*

On the other hand it is required to understand that this abstract interpretation loses information: the calculation $-1517 + 17$ *becomes* $(-) \cdot (+) = (\pm)$, *which is very inaccurate.*

Even with this inaccuracy the static checking is useful, because the operations over this abstract universe is much cheaper to calculate, and the most common mistakes of a programmer can be detected.

A typical abstract interpretation in computer programming is the domain of the type system. In this case every language element is replaced with its type, and every operation is translated to represent the type information.

For static type analysis the abstract computation is easily derivable from the specification of the language: it contains the description for every possible value.

In most languages - including the VIATRA2 VTCL language - the type system can be extended by the user (e.g. in case of Java new classes can be added, while in VIATRA2 a new model element can be added, which is the `to` element of a `typeOf` relation). On the other hand the type analysis needs a fixed set of possible types, so before a type analysis is executed the current type hierarchy has to be identified.

In statically bound languages where the type information is present at compile time it is only needed to compare the types at every function/method call. On the other hand in dynamically bound languages this type information is only available during runtime, but some of these information could be inferred - which the static type checker should be capable of.

Most static type checker algorithms try to read the program once, and try to detect the type information on-the-fly (only using information available before the current assignment). In our project we apply a CSP solver, because it is capable of propagating the information both forwards and backwards (thus making possible to determine the type later, and using that piece of information to infer a type of a variable used before).

2.3 Constraint Satisfaction Problems

The paradigm of the Constraint Satisfaction Problem (CSP) [8] comes from the field of artificial intelligence. CSP solvers are used as a high level, declarative solution for combinatorial optimizations.

The description of a CSP consists of a *set of variables*, a *domain* for each variable, and a *set of constraints* (conditions for the variables). The set of variables and their domain is the space of the constraint problem. A solution of the CSP is a set of variable assignments that fulfills all conditions of the constraints.

The variables (and the constraints) can have different domains: in most cases this domain is a finite domain, but the methodology can be used even over different domains (e.g. in SICStus Prolog there are implementations for boolean variables and real and rational numbers). Our approach is built on a finite domain CSP solver.

The CSP can be represented as a (hyper)graph [18], where the nodes are the variables, and the arcs (and hyperarcs) are constraints for the variable nodes.

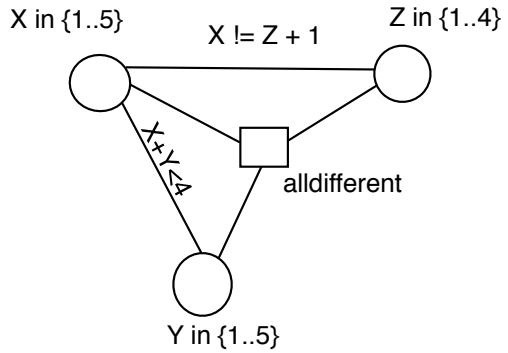


Figure 2.6: A Simple Graph Visualisation of a Constraint Satisfaction Problem

Example 7 *A hypergraph visualization of a CSP problem can be seen in Figure 2.6.*

The problem stated in the graph uses three variables, X*,* Y*,* Z*, with the domains* $[1;5]$*,* $[1;5]$ *and* $[1;4]$ *respectively, and three constraints, which tell us, that* $X \neq Z+1$*,* $X + Y \leq 4$ *and all three variables are different.*

When trying to find a solution of the CSP, there are several possibilities. Modern CSP solver use some combination of backtracking/backjumping and constraint propagation. This general algorithm of solving CSP problems is the following:

1. Define Variables
2. Set up Constraints
3. Constraint Propagation
4. Labeling
5. Repeating the last two steps until either a solution is reached or violation is found.

Constraint propagation modifies the constraint problem to get another problem, which might be easier to solve. It does some reasoning about the constraints, and in some cases it can prove the satisfiability (or violation). Usually the reduction of the domain of one or more constraint variable is determined by the posted constraints and the possible domains of the other constraint variables.

But there are some cases when constraint propagation is not enough to decide the satisfiability, in these cases *labeling* is used: the current state is stored and a possible value is assigned to a variable with more than one possible value (selected by the user of the CSP solver). After the assignments the algorithm returns to the constraint propagation step. In case these assignments cause constraint violation, backtracking (or backjumping) is used to return to a previously saved state where a new assignment is chosen.

When defining a CSP, it is not required to assert only the minimal amount of constraints. Having more constraints can improve the runtime characteristics of the solution, because they can remove symmetries, or help the solver to choose a constraint which reduces the domains of the variables more effectively.

Example 8 *When entering the constraint system defined on Figure 2.6 into the SICStus Prolog* `clpfd` *module, we get the result showed in Listing 2.5. The output domains are the result of the constraint propagation process, but caution is needed when looking at the output: this result does not show what happens with the other variables domains if one of the variables become fixed (in this case when fixing* X*, the value of* Y *also becomes fixed). When using the backtracking functionality of the constraint solver, this problem do not appear, because the backtracking algorithm does not try to find all possible solutions, only a single one.*

Listing 2.5 The SICStus Prolog clpfd Representation of the CSP Example

```
| ?- X in 1..5, Y in 1..5,
    Z in 1..4, X#\=Z+1, X+Y#<4,
    all_distinct([X,Y,Z]).
X in 1..2,
Y in 1..2,
Z in 3..4 ? ;
no
```

A logical extension of the finite domain constraint solver is the ability to check *reified constraints*. Reified constraints are used to describe more complex constraints by allowing the use of boolean functions (such as conjunction, disjunction, inversion or consequence) on constraints. The semantics of these constructs are the following: for every constraint we assign a boolean variable which represents whether the constraint holds or not; the compound constraint holds, if the result of the boolean function with the assigned boolean operands is true.

Example 9 *Using the Prolog clpfd syntax the* `X#=1 \/ X#=3` *constraint is a compound constraint, where the disjunction operator is used. That means, the constraint holds, if at least one of the operand constraints hold. In our case the the variable* `X` *has either the value of* 1 *or* 3*.*

It is important to note that the variables of a CSP are naturally single assignment variables: constraints are globally true on the related variables, while after a change it might seem natural that some (or all) previous constraints are invalidated.

There are numerous CSP solver implementations available, most of them are written in C++, Java or Prolog languages. The implementations have different capabilities, performance and licensing. Some implementations are the ILOG CP (for C++) [25], the clp modules of SICStus Prolog [13] or the Gecode library (for C++ and Java) [22], etc.

2.4 Summary

In this section we gave a broad overview of the different concepts used in this paper. We introduced the basic theory of model transformation and metamodeling, then we described how was the VIATRA2 framework utilizing these concepts.

We also presented the basics of static analysis and type checking, and we described the basic characteristics of the CSP solver frameworks.

Chapter 3

Static Analysis of Transformation Programs

In this chapter I outline a static analyser framework for model transformation programs: first I introduce a generic model for storing the transformation program on an abstract domain followed by its traversal. Then, I show how to connect a CSP solver engine as an underlying evaluator for the analysis.

3.1 Static Analysis and the Transformation Program Model

The proposed static analysis solution is based on the construction and traversal of a Transformation Program Model (TPM). The TPM is a graph model which is an abstract interpretation of the transformation program: it omits information e.g. the current values of the variables. The fact that the attached model space is not tracked allows the analyser to check every possible run path looking for some faults more efficiently.

The main reason to generate this graph model is that it makes to solution more flexible by separating the different tasks of the analysis as described in Figure 3.1. These tasks are the building of the TPM model (see Section 3.2), the traversal of the TPM model (see Section 3.3), generating and solving a CSP (see Section 3.4), and gathering the list of found problems (see Section 3.5).

The TPM can be traversed following the *visitor design pattern* [21] thus allowing the use of different traversal algorithms for the different analysis criteria.

A very important difference between the TPM and the transformation program is that the TPM uses single assignment. This means that a variable is initialized with a value and is bound to it.

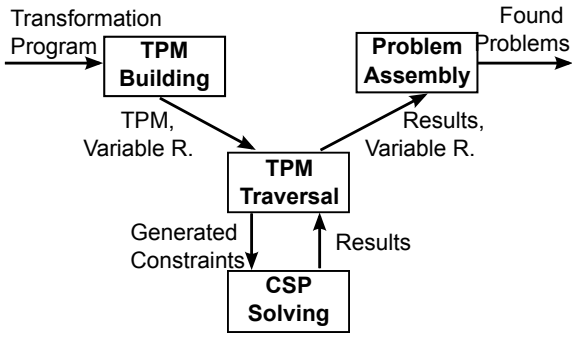


Figure 3.1: The TPM based static analysis process

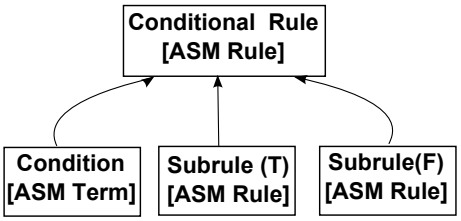


Figure 3.2: The TPM representation of the Conditional ASM Rule

By using single assignment the generation of the constraint satisfaction problems is easier as the generation does not have to handle the lifecycle of the TPM variables.

The transformation program typically has several different run paths. These paths contain different nodes or in different order. The TPM representation of them are *branches* in the graph.

To achieve full coverage in the analysis all these paths should be investigated. This could be done by creating several representation variables for the constraint solver but to avoid unnecessary memory consumption it is recommended that each branch has to be entered separately into the constraint solver.

Example 10 *The Conditional ASM Rule depicted in Figure 3.2 introduces different branches. The rule in the figure contains an ASM Term (called Condition), and two subrules (called* **T***rue or* **F***alse rules). The execution of the Conditional Rule starts with the evaluation of the condition, and then selecting one of the subrules, and only executing it.*

Together with the TPM a *Variable Repository* is also used. It's main responsibility is to store the variables referred in the TPM. The use of this repository allows the traversal to replace the calculated variables with new variables if needed when starting the analysis of a new branch by replacing the repository in order to detect faults that happens only on certain run paths.

3.2 Creating the TPM graph

The TPM is capable of representing the elements of the VIATRA2 VTCL language, and after minor adjustments it should be able representing other graph transformation languages as well.

The traversal is constructed after a traversal similar to the interpreters: it is initiated in the entry ASM Rule, and from this point it follows the control flow. The following main node types are detected:

ASM Terms are untyped expressions from ASM constants, variables and functions. The functions might have parameters of other ASM Terms.

ASM Rules are used as a control structure in the scripts that alter the control flow.

GT Rules are elementary model transformation steps. They may contain graph patterns and ASM Rule calls.

Graph Patters are conditions of the model space. A pattern may contain a pattern graph, calls to other graph patterns and ASM Terms.

In order to be able to understand what a fault described in the TPM means every node should be associated with it's source element. As most nodes semantics is exactly the same as its source element in the transformation program in this section only the differences are listed.

- For every potential failure a `Fail` node is explicitly inserted into the TPM. If the failure is conditional, it should only be inserted the corresponding branch (or branches). The TPM building process should not care about the failure handling - it is the responsibility of the traversal to find the next node in case of a failure.

- For every `Term` node a variable reference is created in the TPM, and a variable is created in the associated variable repository. This approach allows us to describe conditions on the functions without determining the type of the operands (an operand of a function can be any Term).

  These variables in the repository are not the same variables used in the transformation program: they represent the original values in an abstract domain by storing just the properties which are meaningful to the analysis. Similarly to the TPM nodes these variables also associated with their original value.

- There are three *call* constructs in the language: it is possible to invoke an *ASM Rule*, a *GT Rule* and a *Graph Pattern*, and every call can represent a recursive call hierarchy. During runtime the program state will act as a termination condition, but during static analysis this state is not available (more precisely only an abstract representation is available), so in general it is possible that a recursive call represents an infinite length of calls (even worse, the alternative bodies in the Graph Patterns may cause an infinite number of branches in the TPM model).

  Currently this problem is handled by defining a universal k *depth limit* to describe to program in a finite TPM graph. During the building of the TPM the call hierarchy is stored in a stack. When a new call is inserted, the number of its previous occurrences is checked, and in case of at least k occurrences, the called element is not extracted to the model, instead a sentinel node representing no information is inserted.

  It is important to note that this depth limiting is only applied to recursive calls, non-recursive calls are followed into an arbitrary depth (because the source program is finite, these call hierarchies are also finite). On the other hand it can detect and handle indirect (the container element is not directly called but is reached by a series of calls) and circular recursion (two elements call each other) as well.

  Although this limiting reduces the amount of available information before any analysis could happen (and thus it is possible that some errors might go unnoticed) it is conservative: by saying after a limit no information is collected no false negative fault detection can happen.

3.3 The Traversal Algorithm

The TPM is created in order to allow the analysis of the transformation program on a per-node basis: the analysis works by traversing the TPM and generates the appropriate constraints for every node. The TPM node objects support the traversal by supplying the list of nodes to visit before and after the analysis of the node. The number of branches in the current node, and allow the updating of the related variables in the Variable Repository and the constraint generation process.

According to the visitor pattern [21] the control of the traversal is handled by an external traversal control class, the *visitor*. In order to have the best fault detection capabilities the visitor should be capable of traversing every node in the TPM, identifying and handling branches, filling the constraints to the constraint solver, updating the variables in the connected Variable Repository, identifying faults and handling fail nodes.

The *traversing of all nodes* is required in order to achieve full coverage of the transformation program. If this condition is fulfilled, the type of traversal does not matter (in theory): in every node local constraints are created, and the CSP solver have to be capable of weaving these constraints. However in practice changing the traversal can help to find an unsatisfiability more efficiently.

Listing 3.1 introduces the used algorithm.

Listing 3.1 The Traversal Algorithm

```
traversal(){
  while (!allBranchTraversed){
    selectNextBranch();
    traverseNode(rootNode);
    evaluateResults();
  }
}

traverseNode(TPMNode node){
  int branchNumber = calculateActualBranch(node);
  for (TPMNode before : node.getBefore(branchNumber))
    traverseNode(before);
  node.addConstraints(branchNumber);
  node.updateVariables(branchNumber);
  for (TPMNode after : node.getAfter(branchNumber))
    traverseNode(after);
  if (FailNodeHit)
    jumpToFailHandler();
  if (CSPFailure)
    stopTraversal();//stops the traversal of the branch
  }
}
```

The the `traversal` method is used to manage the different branches, and for each branch start a traversal by calling the `traverseNode` method. After each traversal the results are evaluated and the found problems are logged.

The `traverseNode` method first calculates which branch to choose at the selected node. The path depends on the previously selected branch, and is required for both calculating the subnodes and generating the constraints.

After the branch is decided, the concrete traversal begins. The subnodes are broken into two groups: (i) nodes to visit *before* generating the constraints and (ii) nodes to visit *after* generating the constraints. The algorithm traverses first the *before* nodes recursively, then generates the constraints, updates the Variable Repository, and finally traverses the *after constraints*.

There are two cases which break that flow:

- If the constraint solver reports failure, the traversal of the current branch is stopped, and the results are evaluated.
- When a Fail node is hit, the control is given to the last fail handling node, and the other partially traversed nodes are ignored.

3.3.1 Branch Handling

A very important part of the traversal control is the *branch handling*. The traversal control is responsible to run every possible branch one by one, and start a clear CSP for every branch. The branch handling is based on a simple *backtracking algorithm*: when it reaches a branching point, it saves the current position as decision point, selects the first untested branch, and the process continues until either an end point is reached (there are no more nodes to traverse) or the CSP solver reports unsatisfiability. After evaluating the results from the CSP solver, it starts the analysis of the next branch. In the new testing it will traverse upward back until the last branching point (with an untested branch) of the last run, and changes it to the next available branch.

This algorithm is similar to depth-first search, where the branching points are represented by the nodes of the graph, and their sequence are represented by the arcs.

3.3.2 Fail Node Handling

The transformation language includes the *fail* construct for error detection. Failures are similar to the exceptions of OOP languages: they represent the fact of failure. If it happens during the execution, the interpreter jumps to the error handling routines (or if there is no handler, the execution stops).

This jump is an alternate continuation of the program, so it has to be represented by a new branch. There are two rules which can fail: (1) the `fail` rule represents an automatic failure (because the failure is automatic, only a single branch is used, which jumps to the failure handler), and (2) the `choose` rule fails if no match is found in the model space, so a `fail` node is inserted to the corresponding run path.

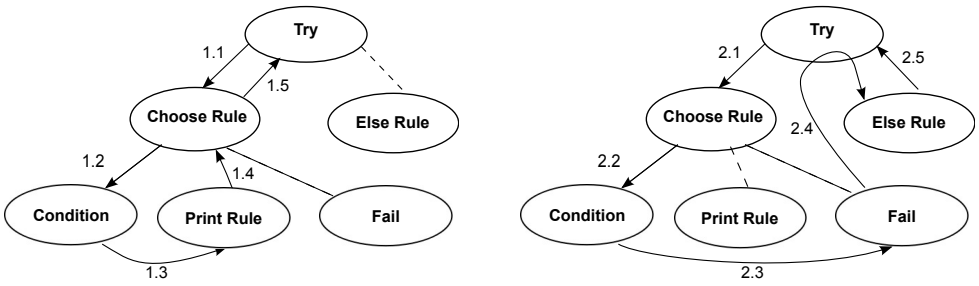
There are two rules, which allow the handling of failures: (1) the `try` rule looks for failures in it's main rule, and executes its else rule if any failure is found, while (2) the rule `iterate` finishes iteration if a failure happens.

Example 11 *Listing 3.2 describes a single* `try` *rule. Figure 3.3 displays the potential execution paths of the structure*

Listing 3.2 A VTCL rule demonstrating the failure handling

```
try
  choose Token with find placeWithToken(Place, Token) do print("token found");
else
  print("Else Rule started");
```

The main rule to test contains a single `choose` *rule, while the failure handling rule (named Else in the Figure) is not detailed. The solid arrows represent the*



(a) No Failure (b) With Failure

Figure 3.3: The Execution of the Try Rule

control flow, while the jagged arrows connects the nodes not present in the run path to their container.

The first path (a) displays the following scenario: inside the `try` *rule the* `choose` *rule is executed. This evaluates the condition, a match is found, so the corresponding* `print` *block is called. Then the calls end, and the control is returned to the caller, so the* `try` *block ends.*

On the other hand the second path (b) activates the error handler: the `choose` *rule is executed, but the condition does not hold. The rule fails (*`Fail` *node), so the control gets to the failure handling rule. If the execution of this rule finished, the control is returned to the caller, the* `try` *block ends.*

This error handling mechanism is capable of breaking the normal flow of the traversal. In case of no error handling node is found the traversal ends.

3.3.3 Updating the Variable Repository

When the traversal reaches a node the variables changed by the nodes referred program element should be updated in the Variable Repository.

As variables of the TPM model are single assignment variables the updated value must be represented by a new TPM variable, so a program variable is represented with a series of TPM variables.

The first TPM variable is created when finding the program variable first (e.g. as a symbolic parameter of a call), then a new one is created when reaching a node updating it.

The TPM variables are stored in the Variable Repository component, which allows updating an existing variable and querying the representation of the current value.

When creating a TPM variable after an update, the unchanged parameters of the previous TPM variable should be the same in both variables – this can be

reached by filling constraints into the constraint handler.

3.3.4 Enhancing the Performance of the Traversal

As described before for every branch a new Constraint Satisfaction Problem is initialized, and as described in Section 3.3.1 the TPM is traversed again to generate all the constraints. It seems that by avoiding these repeated steps the speed of the analysis could be improved.

To achieve this at every branching point the state of the analysis should be saved, and at a later point restored. This state consists of the TPM variables in the Variable Repository and the state of the constraint solver.

To support this the traversal algorithm has to be modified:

1. When a new branch is started, the traversal should only run through the nodes already used for the stored state, and the constraint solver and variable repository should be initialized from the saved state.

2. When a new branching point is detected, a new state save should be created, and attached to the branching point. This state can be discarded when all possible branches of the branching point are explored.

The effects of this performance enhancement are investigated in Chapter 5.

3.4 Constraint Generation

In order to the static checker be independent of any concrete CSP solver, an abstraction layer is implemented on the above the solver using the *Bridge* design pattern[21].

Based on the Bridge pattern we defined a general interface for handling the CSP Solvers: this interface allows filling some predefined type of constraints and checking the status of the solver (both by querying the domains of the variables and looking whether it is possible for all constraints to hold). The list of constraints needed depends on the property to check – the interface should be revisited each time a new property is added to the analysis.

The class diagram in Figure 3.4 describes the classes used for integration the CSP Solver to our architecture.

The constraints filled to the Constraint Handler are parametrized with the TPM variables - the Constraint Handler has to read the needed properties of these variables and match them to CSP variables. This matching is bidirectional thus enables the Handler to return the TPM variable as a cause of failure.

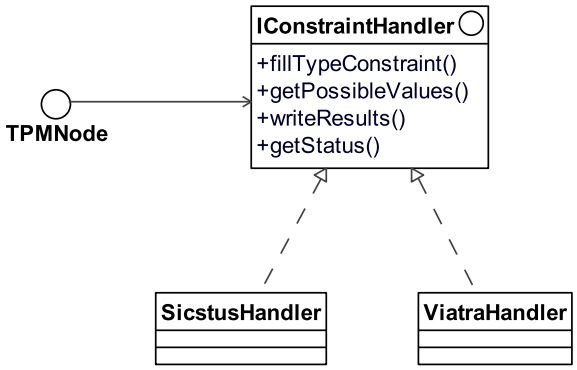


Figure 3.4: An UML class diagram describing the integration of the CSP solver

When a TPM node is reached during the traversal, the node generates its constraints using TPM variables queried from the Variable Repository. The constraints then are filled to the Constraint Handler.

This way the constraints are incrementally imported to and checked for inconsistencies in the constraint solver. If contradictory constraints are found before reaching the end of the transformation program, the solver reports the error and stops further traversal. This method gives a hint where the problem has happened - the Constraint Handler is not capable of detecting every fault where it happened.

3.5 Fault Identification

The static analyser system use three mechanisms to detect failures:

Constraint Failures It is recommended to try to identify faults during the traversal as during the traversal extra context information is available which can be used for more efficient detection of the cause of the fault. This context information is available naturally if the Constraint Handler reports failure, that happens only if the fault is related to a single CSP (and thus a TPM variable).

Inconsistencies It is possible that the fault manifests as inconsistent results on multiple variables (e.g., the TPM representation variables of a program variable do not share a common property), or different branches return different properties of the same variable, that cannot be detected easily. The current approach is to check such properties after a branch is traversed, and the results are saved for future branches.

Traversal Faults A third kind of fault to identified is directly related to the traversal: if a traversal cannot successfully finished, it also indicates an error. Such error can happen by finding a failure node without a fail handling node, which terminates the analysis.

It is also important to differentiate between faults by severity. Our model uses two severity categories: *error* and *warning*. Error means a serious fault, e.g. contradictory constraints over a property of a TPM variable. This severity is used to describe faults which cause failures during execution. On the other hand, warnings are used to indicate potential problems that may or may not cause errors on the output, e.g. the halt during an unhandled fail node may be the expected outcome, but it is not recommended to use it that way.

3.6 Related Work

There are several static analysers used for different languages with a different set of capabilities based on different approaches. We now introduce one of them that has conceptual similarities with our approach.

The FindBugs [24] is a static analysis solution for Java. It is based on the concept of *bug patterns* which are possibly incorrect usage of the language. By detecting such patterns it is possible to catch a few common errors while keeping the number of warnings relatively low[41]. To increase the error detecting capabilities of the system new patterns have to be defined. This bug pattern concept can be used for describing the various inconsistencies between the parameters of multiple TPM variables.

3.7 Summary

In this chapter we introduced a general static analysis solution for model transformation programs. The solution is based on a generic graph model of transformation programs, the TPM.

We described in general how to build such a model, and then how to use this model to analyse the transformation programs. In the next chapters we will describe how to build a static type checker based on this concept, and then evaluate the runtime characteristics of the implementation.

Chapter 4

Type Checking of the VTCL Language

In Chapter 3 a generic method for static analyis of model transformation programs was introduced. To demonstrate the capabilities of the proposed solution we describe a static type checker system for the VIATRA2 framework based on this generic method.

In this chapter we introduce first the capabilities of the implemented type checker tool, then describe its integration of the analyser into the VIATRA2 framework, then the use of the CSP Solver for type checking, and in the end of the chapter the type constraints and element-specific traversal information is described for every element of the VTCL language.

4.1 Capabilities of the Type Checker

The VIATRA2 framework uses the VTCL language for defining model transformations. Some elements of the language, such as the graph patterns contain type information implicitly, while the main, ASM-rule based control structures use untyped variables.

The goal of the type checker is twofold: first it has to determine the types of the variables regardless if it is available in the transformation program or has to be inferred, and second, it should detect the incorrect use of types, e.g. the use of a model element instead of a boolean variable.

The execution time and memory consumption of the analysis is an important concern during the implementation, because the main goal of the tool is to help the developer of the transformation program to identify the potential problems in the code as early as possible. This requires the analysis to complete within a reasonable amount of time on an average developer computer.

It is important to note that it is not the goal of the type checker to detect problems related to undefined (`undef`) values. If some variable has an undefined value, it means it is either not initialized or already deleted.

This should be mentioned here, because the undefined value is special: the `undef` value can represent any *model element types*, but nonetheless it is still a value of a variable which is irrelevant during type analysis.

To have such capabilities to the analyser, another traversal of the TPM model is suggested with the goal of finding such (and maybe other value-related) problems.

4.2 Integrating the Analyser

The VIATRA2 framework is a set of Eclipse [2] plugins. This plugin-based architecture enables extending the framework in a well defined way.

Figure 4.1 shows the main components that the static type checker is connected.

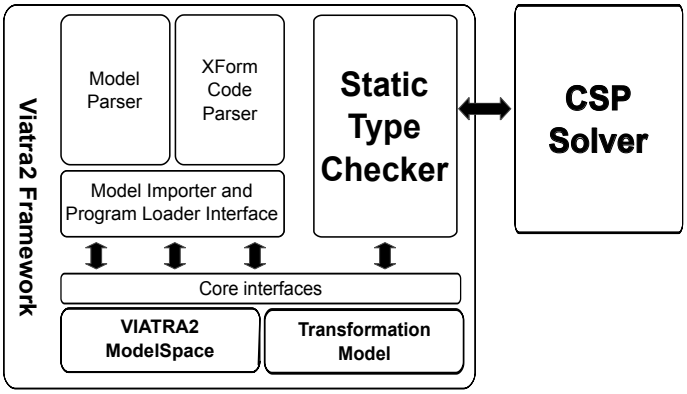


Figure 4.1: The static type checker in the VIATRA2 framework

It is important to note that the new static checker component is not parsing the various textual and graphical languages defining the model space and the VTCL language, instead it relies on the framework parsers. The analyser only communicates through the core interfaces with the model space and the program model store.

4.3 Using a CSP Solver for Type Checking

4.3.1 Representing the Metamodel as Constraints

The finite domain CSP solvers are not capable of handling neither the TPM variables nor the metamodel. In order to use a CSP solver for type checking the metamodel

has to be mapped as finite domain variables and related constraints.

In this section we are presenting a solution for two problems: first we create a representation of the hierarchy, and then we discuss how to use this representation in constraints to be evaluated by the CSP solver.

Representing Hierarchies with Integer Variables

There are some well-known ways to represent hierarchies with integer variables for hierarchies only allowing single inheritance, like the concept of nested sets [28], that is used to represent the tree hierarchy in a relational database.

The basic idea is to associate two numbers to each node in the hierarchy: an entry number, which is smaller, than all the entry numbers of its descendants, and an exit number, which is larger, than all the exit numbers of it's descendants. A node's exit number has to be larger than its entry number. It is easy to generate these numbers during a *preorder tree traversal*.

Provided that these numbers are set, deciding, whether an object is a descendant of the other only requires evaluating two simple relations: the potential descendant has (1) a larger entry number and (2) a smaller exit number than the potential ancestor. The `subtypeOf` relationship holds if and only if both relations hold.

But for multiple inheritance hierarchies this representation does not work. We have chosen another algorithm described by Yves Caseau in [14], that represents the position of a node in the hierarchy with a set of integers.

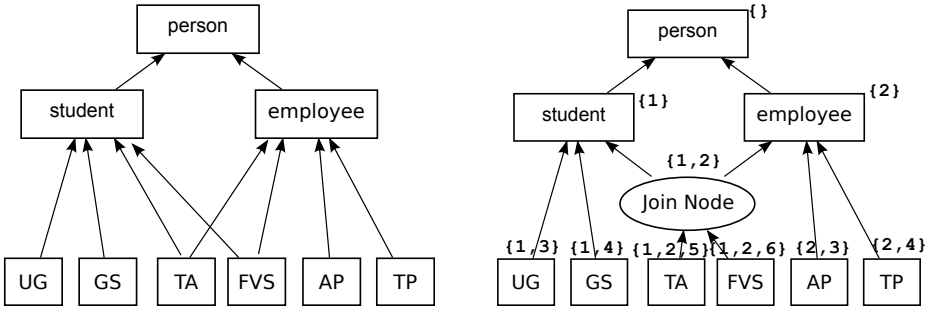
The algorithm refers to the set of numbers assigned to the nodes as *genes*, because they operate similar to the genes in biology: in the algorithm the descendant node inherits all genes of all of its ancestors. This construction guarantees that a node is descendant of another node if and only if the set of the "descendant" node is a superset of the set of the "ancestor" node.

Example 12 *As the Petri net metamodel does not contain inheritence first another gene assignment is used to describe these capabilities. In Figure 4.2 (the example is taken from the article describing the algorithm).*

The hierarchy describes members of a university. Every member is a `person`*, and they can be* `student`*s or* `employee`*s.* ***U****nder****G****raduates and* ***G****raduate* ***S****tudents are students,* ***A****ssistant* ***P****rofessors and* ***T****emporary* ***P****rofessors are employees, while* ***T****eaching* ***A****ssistants and* ***F****oreign* ***V****isitor* ***S****tudents are both employees and students.*

To illustrate the usage of the gene sets we interpret the results on some model element pairs:

- *The gene set associated to the element* `student` *is* 1*, to* `FVS` *is* $1, 2, 6$*.* $1, 2, 6$ *is a superset of* 1*, so* `FVS` *is a descendant of student.*
- *The gene set associated to the element* `GS` *is* $1, 4$*, to* `AP` *is* $2, 3$*. None of the sets are superset of the other, so the elements are not in an inheritance relation.*



(a) The Hierarchy (b) The Gene Assignment

Figure 4.2: The Gene Assignment for the University Member Hierarchy

The gene assignment algorithm consists of two phases:

1. first it transforms the original hierarchy to an algebraic lattice structure,
2. then assigns genes to the elements of this structure.

A lattice structure requires for every member a greatest lower bound and a lowest upper bound to exist - these conditions can be fulfilled by adding join nodes at well chosen places (informally nodes with multiple parents are altered: their parents are substituted with a join node as their single parent).

The gene assignment is carried out in the following way:

1. every node inherits all the genes of all their parents,
2. and every simple node (it has only a single parent) adds a new gene to the set.
3. The new gene is selected by looking the nodes with the same parent, and selecting a gene which is not used to describe those nodes.

With this simple algorithm it is possible to create conflicts in the gene assignments, because it is possible to add a join node between branches which have conflicting genes. Such a conflict should be considered harmful, because it interferes with our goal of identifying the position of the node in the hierarchy by looking at the gene set. In order to be able to use an incremental assignment algorithm, such conflicts have to be detected and resolved by changing one of the genes for another, a safe one (even for the parent nodes).

These conflicts could be avoided also by using globally unique genes, but the original algorithm did not choose this path, because one of the main goals of the algorithm was to use a minimal amount of genes. This allows to use the algorithm

even for large hierarchies. In our case it is also required to use as few genes as possible: smaller gene sets help to reduce the memory consumption (details later, in Section 4.3.1).

Example 13 *When running this algorithm on the metamodel hierarchy of the Petri nets defined in Section 2.1.1, the algorithm assigned genes between the numbers of* 1 *and* 14*. The resulting gene sets are present in Figure 4.3.*

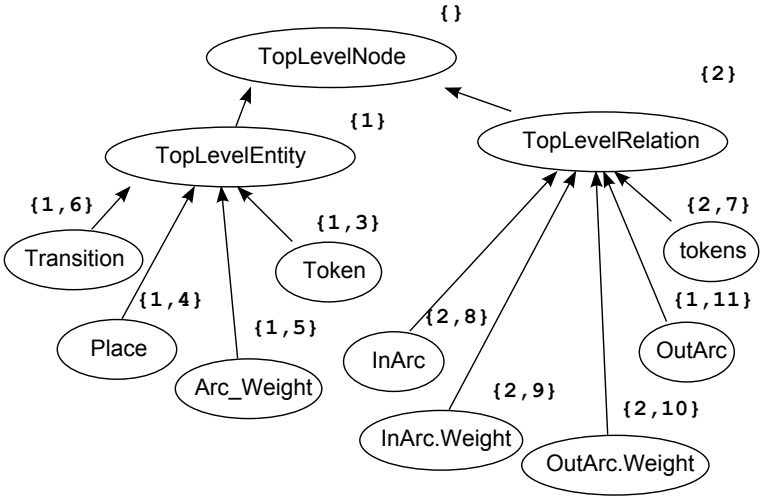


Figure 4.3: The Gene Assignment for the Petri net Metamodel

The algorithm creates three special nodes (these added nodes are omitted from Figure 4.2) that are not present in the original meta-model hierarchy: the `TopLevelNode`, the `TopLevelEntity` and the `TopLevelRelation` nodes. They specify respectively an abstract model element and relation node acting as an ancestor of all model elements and all relations. They can be used in some constraints to express the fact that the result is a model element, or more specifically a relation, but without restricting the type of the model element.

The metamodel of the Petri nets does not contain any inheritance at all, but these newly created nodes explicitly mark some inferred relationships that were only implicitly presented in the metamodel: everything is the descendant of the `TopLevelNode`, every entity is descendant of `TopLevelEntity` and every relation is descendant of `TopLevelRelation`.

It should be noted that the gene algorithm does require only a single special node for a correct gene assignment: it acts as the greatest lower bound of every type from the hierarchy in lattice representation (this is the `TopLevelNode`), but the algorithm does not require the Entity and Relation hierarchy to be explicit.

On the other hand there are some constraints (e.g. related to the ASM Term function `target`, detailed in Section 4.4) that refer (directly) to the Relation or

Entity supertype (locally we cannot decide the actual Entity/Relation type). This supertype is marked with the corresponding node.

Creating Constraints based on the Metamodel

By the description of the metamodel the main goal is to represent the *type hierarchy*, and the *to*, *from* and *inverse* parameters of the relations of the metamodel. To achieve this every TPM variable has to be mapped to CSP variable (or variables) describing it's type.

The fact that the TPM variables are single assignment variables allows each CSP variable to represent a property of a single TPM variable, thus a static mapping between CSP and TPM variables is enough.

The type of a VTCL variable (and thus a TPM variable) can be one of the built-in types (`Integer`, `Double`, `Boolean`, `String`, `Multiplicity`), or a model element (in this case the type is a metamodel element). The type information is not present compile-time, only during runtime, and the type of a variable can change runtime (dynamic binding).

To represent the type hierarchy, I defined two relations, the *type equality* and the *substitutability* relations.

The *type equality* relation is defined on two model elements (or metamodel elements). It is a symmetric relation, and it describes that the type of the two model elements is exactly the same. The typical usage of the relation is to precisely describe the type of a TPM variable - the second parameter should be a well-defined type from the metamodel. When using gene sets the type equality relation can be expressed by stating that the gene sets representing the two model elements are equal.

The *substitutability* relation is a directed relation between two model elements (or metamodel elements): it describes that the type of an element is either the same as the type an other element, or it is a descendant of the other element's type. It can be used as a representation for variable assignments: the type of the assigned value has to be the same as the variable's, or it can be a descendant of it. When using gene sets the substitutability relation can be expressed by stating that the gene set of the ancestor element is the subset of the set of the descendant element.

The substitutability relation is more general than the type equality, but for those types, which do not have any descendants, the two relations are the same. In the same way for the built-in types the two relations equal as well (because there is no descendant of e.g. an Integer number). This can be used as a kind of optimization, when choosing the constraints.

The type of a TPM variable is represented by two different CSP variables: one integer variable describing which built-in type is the type, and one integer set representing the gene set of metamodel element. This integer set is only created

when the TPM variable is a model element - this decreases the memory consumption of the type checking process.

It might be possible to store the built-in types in the same inheritance hierarchy as the metamodel: the type hierarchy might contain at the same time every possible type. It might be simpler to handle to variables this way, and there is no need to synchronize the two types of inner variables. But the main drawback of this approach would be the slightly higher memory consumption (and an increased runtime is also possible): the domain of the set variables is the powerset that is exponentially large.

The described inheritance hierarchy does not help to describe the required relation properties, because those properties are independent from the hierarchy. On the other hand these properties are easier to describe because knowing the relation determines the types of it's parameters.

To allow the constraint solver to propagate over the constraints, I propose to describe these properties by a set of conditional constraints. The constraints are all of the following scheme:

$$(Relation\ is\ \text{“}relationtype\text{”}) \Rightarrow (Parameter\ is\ \text{“}parametertype\text{”})$$

In these constriants *Relation* is the variable representing the relation, "relationtype" is a constant relation type, while *Parameter* is the variable representing the searched parameter, and "parametertype" is a constant model element type. When filling such a constraint for every "relationtype" (the "parametertype" is then fixed), it will allow the constraint engine a two-way propagation process (either determining the type of the parameter or in some cases the relation variable's type).

Example 14 *Over the domain of Petri nets (the metamodel is depicted in Figure 2.3) to express that the variable R is a relation, and the variable F is the "from" parameter of R, the following constraints are needed:*

- $(R\ is\ InArc) \Rightarrow (F\ is\ Entity)$
- $(R\ is\ InArc.Weight) \Rightarrow (F\ is\ InArc)$
- $(R\ is\ OutArc) \Rightarrow (F\ is\ Place)$
- $(R\ is\ OutArc.Weight) \Rightarrow (F\ is\ OutArc)$
- $(R\ is\ tokens) \Rightarrow (F\ is\ Place)$

4.3.2 The Constraint Handler API for the Traversal

As described in Section 3.4 it is required to define some generic constraint types, and wrote an implementor (as described in the Bridge pattern), which can handle these constraints. Our list of constraints is the following for the type checker:

Type Equals Constraint represents the type equality of two elements. This constraint represents a substitutable relation (as defined in Section 4.3.1). The inverse of the constraint states that two types are different.

Type List Constraint states that the type of an element is one of a set of types. This constraint is similar to a disjunction of several type equals constraints, but there is a huge difference: the type list defined in this constraint should consist of a set of predefined type while the type equals constraint can handle two variables as parameters. This constraint can be more efficiently implemented in the CSP solver framework as a disjunction of several type equals constraints. The inverse of the constraint states that the type of an element is not in a set of types.

Conditional Constraint is a compound constraint of two subconstraints: it represents a logical consequence relation between a condition and a consequence constraint. This constraint does not require the condition to hold.

Conjunctive Constraint is a compound constraint with an arbitrary number of subconstraints: it represents a logical conjunction between the subconstraints. Basically this means, the constraint holds only if every subconstraint hold.

Disjunctive Constraint is a compound constraint with an arbitrary number of subconstraints: it represents a logical disjunction between the subconstraints. Basically this means, the constraint holds if at least one subconstraint holds.

Inverse Constraint is a compound constraint with a single subconstraint: it represent a logical inverse of the subconstraint. Basically this means, the constraint holds if and only if the subconstraints does not.

It is deliberate to have both the *Inverse Constraint* and the possibility to define the inverse of the simple constraints (not compound, more specifically the *type equals* and the *type list* constraints). The Inverse Constraint can be implemented using constraint reification (which introduces a new boolean constraint variable) while the simple constraint may be inverted more efficiently.

These constraints are defined over the variables of the TPM, and the implementor class has to be map these into CSP variables.

4.3.3 Selecting a CSP Solver Engine

The analysis uses integer and integer set variables and some very simple constraints. That allows us to investigate several available CSP solver implementations and choose the one which suites the needs best.

During the development the Gecode/J library [22] and the clpfd module of SICStus Prolog was evaluated, but these implementations did not meet our needs exactly.

Generally supporting sets in constraint solvers needs compromises, because the domain of possible sets is the powerset of the set elements which contains exponentially many elements. The solvers that support sets use some kind of optimalisation to overcome this aspect.

The Gecode library had a set representation, but it did not work well with our specialised sets; on the other hand the SICStus module did not support neither sets nor incremental problem building.

These problems led to create our own solver implementation. This implementation uses a simple object-oriented propagation algorithm: every constraint variable notifies the related constraints of it's changes, and the constraints try to propagate that information further. This approach handles the investigated type checking problem spaces quite well, more detailed performance assessment is in Chapter 5.

4.4 Traversing ASM Term Nodes

As described in Section 3.2 for every ASM Term Node there is an assigned TPM variable, which represents the result of the the value of the current term.

To traverse an ASM Term node, all the operand nodes have to be traversed as well, if there are any (variable and constant term does not have any operands).

4.4.1 Variable and Constant Terms

At the end of a Term branch we will always find a Term without operands. These Terms represent variables and constants. These terms can be handled the same way with two differences: in case of a constant element (1) the type of the element is always available, and (2) this type cannot change.

The handling of these nodes is simple: the variable shall be put on the constraint space as described in Section 4.3.1, and the type information has to be filled (if available) using the type equality relation.

4.4.2 Arithmetic Terms

The VTCL language includes the basic arithmetic functions: *addition*, *subtraction*, *multiplication*, *division*, *remainder* and *arithmetic inverse* calculation are available. All these terms can be handled similarly, so we will cover only the addition in details (which is usually the most complex of these operations).

The addition function has two operands (similarly to the mathematical operator), both operands and the result are either `String`, `Double` or `Integer`. There are further constraints on the possible types are listed in Table 4.1.

Operand1	Operand2	Return Value
{String}	{Integer, Double, String}	{String}
{Integer, Double, String}	{String}	{String}
{Double}	{Integer, Double}	{Double}
{Integer, Double}	{Double}	{Double}
{Integer}	{Integer}	{Integer}

Table 4.1: The Type Constraints of the Arithmetic Addition Operator

To understand the used notation, we describe the first line in plain English: it means, that if the first operand is a `String`, the second is one of the types `Integer`, `Double` or `String`, than the return value is a `String`. It is important to notice, that at least one of these constraints will always hold, if all the variables are of the allowed types, and all the constraints in the list have a single type on the right side. These facts mean, that there is a deterministic connection between the types of the operands and the type of the return value, so there is no need to create branches for the different output types.

The other arithmetic operators can be treated similarly, with the following differences: (1) neither of them allow the `String` type as operand or return value, (2) the remainder operation also disallows `Double` variables, (3) and the arithmetic inverse function has only a single operand.

4.4.3 Conversion Operators

Conversion operators are used to transform it's operand to another type. There are conversion operators available for the built-in types but none for `ModelElement` types.

The conversion operators do not work on every possible operand type (e.g. an `Integer` cannot be converted from a `Modelelement`).

The detailed type constraint are listed in Table 4.2.

The conversion operations are also deterministic, they return only a single value type.

Operation	Operand	Return value
`toString`	{any possible type}	{String}
`toInteger`	{String, Integer, Boolean, Double}	{Integer}
`toDouble`	{String, Integer, Boolean, Double}	{Double}
`toBoolean`	{any possible type}	{Boolean}
`toMultiplicity`	{String}	{Multiplicity}

Table 4.2: The Type Constraints of the Conversion Operators

4.4.4 Relational and Logical Operators

The VTCL language supports the usual arithmetic comparisons: *less than*, *less than or equals*, *equals*, *more than or equals*, *more than* and *not equals*. They perform a comparison on their operands. Their type constraints are listed in Table 4.3. These operations do not need branches.

Operation	Operand	Return value
Less	{String, Integer, Double}	{Boolean}
Less or Equals	{String, Integer, Double}	{Boolean}
Equals	{any possible type}	{Boolean}
More or Equals	{String, Integer, Double}	{Boolean}
More	{String, Integer, Double}	{Boolean}
Not Equals	{any possible type}	{Boolean}

Table 4.3: The Type Constraints of the Arithmetic Comparisons

The commonly used logical operators are also supported: `not`, `or`, `and`, and `xor`. The `not` operator has a single operand which shall be of `Boolean` type, and it's result is a `Boolean`.

The other logical operators have two `Boolean` parameters, and their return values are `Boolean` values. None of the logical operators needs branching.

Model Element Query Operations

The Model Element Queries are built-in functions that let ASM Terms utilize some element properties in the VPM model space (and with the help of these terms also in ASM Rules). The names of the query functions are representing the names from the VPM metamodel.

Table 4.4 displays all functions with their type constraints. There are two types which have not been used before: `Model Element` represents any possible

model element (descendant of the root of the model element type hierarchy), while `Relation` similarly represents any possible relation (descendant of the root relation in the model element type hierarchy).

Operation	Operand	Return value
`isAggregate`	{Relation}	{Boolean}
`value`	{Model Element}	{String}
`ref`	{String}	{Model Element}
`fqn`	{Model Element}	{String}
`name`	{Model Element}	{String}
`inverse`	{Relation}	{Relation}
`multiplicity`	{Model Element}	{Multiplicity}
`source`	{Relation}	{Model Element}
`target`	{Relation}	{Model Element}

Table 4.4: The Type Constraints of the Model Element Query Operators

Relation parameter constraint sets (as described in Section 4.3.1) are used at the `inverse`, `source` and `target` queries to increase precision.

4.4.5 ASM Functions

ASM Functions are similar constructs as HashMaps in Java, or associative arrays in some dynamic languages; it is possible to put items into and retrieve items from them by assigning a Term as a key.

To handle these functions the following algorithm is used: at the initialization of the type checking process the types of the stored values are gathered (during startup the functions stores values set in the transformation program directly), and the possible outputs of an ASM Function call are these values. If there are multiple types, branching is needed.

The updating of ASM Functions has to be treated similarly to the update of variables: using the Variable Repository a copy of the function has to be created, and this copy can be modified, and later this modified copy can be constrained.

4.5 Traversing ASM Rule Nodes

Most ASM Rules does not generate constraints directly (a notable exception is the Conditional rule, detailed later), they are used to describe the possible paths in the TPM.

The ASM Rules are discussed in the following groups: Simple ASM Rules, Variable Definition Rules, Nested Rules, Conditional Rule, Model Manipulation Rules and GT Rule Invocations after the discussion of the ASM Rule Calls.

4.5.1 Calling ASM Rules

In the VTCL language the `call` rule is used for the invocation of other ASM rules. As of these call can also be recursive, the *depth limit* introduced in Section 3.2 is applied for the called rules, and in case of the depth limit is reached, an empty ASM Rule is present in the TPM.

The `call` rule's responsibility is to match the called rules symbolic parameters with the actual parameters given in the call node. This parameter matching must happen both before and after the call takes place, because only this way is it possible to handle the changes of the variables inside the called rules.

4.5.2 Simple ASM Rules

The Simple ASM rules are such rules that do not contain other ASM rules. These rules are the following:

- The `skip` rule is an empty instruction - it needs no special handling.
- The `fail` rule is used to cause failures - when it is hit, a failure handling should start.
- The `update` rule is used for modification of existing variables. As the execution changes the VTCL variable, a new TPM variable should be created during the traversal of this node in the Variable Repository, and it shall contain the new value (defined by an ASM Term parameter). From this point everybody referencing the variable shall use the new value.

4.5.3 Variable Definition Rules

The Variable Definition rule (for short the `Let` rule) defines variables. The rule consists of an arbitrary number of variable definitions and a body ASM rule. A variable definition consist of a variable and an ASM Term, while the body ASM rule defines the context in that the defined variables are available.

It is possible to extract a constraint for the variable assignments: every variable has to have the same type as the corresponding ASM Term. The execution continues with the body ASM rule.

Rule	Subnodes	Constraints	Branches
Sequential	Arbitrary number of subrules	No additional constraints	One
Parallel	Arbitrary number of subrules	No additional constraints	One
Random	Arbitrary number of subrules	No additional constraints	One for every subrule

Table 4.5: The Analysis of Nested Rules

4.5.4 Nested Rules

The Nested Rules (`sequential`, `parallel` and `random`) are used to handle an arbitrary number of subrules in a single construct.

The sequential rule runs every subrule one by one, which can be mapped to type constraints as every type constraint of every subrule must hold.

The parallel rule also runs every subrule, but at the same time. The mapping is done similar to the `sequential` rules. This approach has its limitations: it cannot find every problem. If there are conflicting rules (several rules try to change the same model element/variable), the parallel execution may introduce even race conditions: some orders may be type safe, while others not. A more refined approach would be the introduction of branchings in parallel rules: for every possible order a different branch is calculated. The number of created branches would be $n!$ (where n is the number of subrules), which means this solution cannot be used for large parallel models.

The random rule runs only a single subrule that is selected randomly. The mapping creates branches for every subrule, and checks there the constraints to hold.

Table 4.5 contains a short summary of the handling of the nested rules.

4.5.5 Conditional Rule

A Conditional rule consists of condition term and two subrules that represent the true and false cases.

The condition term of the node has to be of `Boolean` type: this constraint has to be filled directly from the rule, because on the level of the Term this information is unavailable.

Because only one of the two subrules will run, this rule is the start of two different branches: one in that the subrule representing the true case will run, and one in the other.

4.5.6 Model Manipulation Rules

The transformation control language has constructs for manipulating the model space. There are constructs for creation, change and deletion of model elements. These constructs have parameters: we know some basic things about these parameters (mostly they have to be a model element reference), this knowledge can be filled into the CSP solver.

The Create rule creates a new model element in the model space. The parameters of this construct are a variable and one or more ASM Terms. The variable will store the created value, while the Terms describe the type of the model element, and some additional parameters. The created model element (and thus the variable) will have the type given as parameter. Some further type constraints related to this node are the following: (1) the type parameter have to be a *model element type* (either entity or relation); (2) a variable is used to store the new element, it will have the same type as the type parameter; (3) if filling in a relation, the `from` and `to` parameters have to be model element references, and the relation parameter constraint sets are also filled.

The Delete rule has an ASM Term parameter: the element to delete from the model space. In this case the element reference has to be invalidated for future use - it does not refer to any elements from the model space any more. The parameter has to be a model element type.

The Copy rule is similar to the *create*: it creates a new item by creating a copy from an existing one. The constructs parameters are: the source element, the target, and a variable. The source element and the target are model elements, while the variable is a term variable (similar to the *create* rules variable parameter).

The Move and the Update rules change existing model elements. Their parameters select an existing model element, and define what to change. These information can be mapped in a similar way as the parameters of the other model manipulation constructs. There are several kinds of update rules, they are a bit different in the parameters handling, but all of them contains two ASM Term parameters. Without further explanation the possible types of the update rules are the following:

- `rename(Model Element, String)`

Rule	Subnodes	Constraints	Branches
Create (entity)	variable, type	variable is a `type` entity	One
Create (relation)	variable, type, two model elements	variable is a `type` relation, the last parameters are model elements, relation parameter constraints	One
Delete	Model element	Parameter is a model element	One
Copy	Source element, variable	The type of variable and source equals to the type of the source element, both are model elements	One
Move	Source element, target container	Both elements are model elements	One
Update	Two ASM Terms	Varies	One

Table 4.6: The Constraint Mapping of Model Manipulation Rules

- `setValue(Entity, String)`
- `setFrom(Relation, Model Element)`
- `setTo(Relation, Model Element)`
- `setMultiplicity(Relation, Multiplicity)`
- `setAggregation(Relation, Boolean)`
- `setInverse(Relation, Relation)`

In case of the `setFrom`, `setTo`, `setInverse` rules, relation parameter constraint sets are also inserted.

Table 4.6 contains a short summary of the handling of model manipulation rules.

4.5.7 Collection Iterator Rules

The `forall` and the `choose` rules execute rules for all elements (or a single element), that have (or has) a specific property. For both rules the parameters are the same:

a list of variables, the property description, which can be either an ASM Term or a GT Rule call, and an optional ASM rule to execute.

If the properties are described by a Term, than the Term has to be traversed as described in Section 4.4, while in case of GT Rules Section 4.6.2 has to be followed. If an ASM Rule is present, it has to be traversed as well.

The difference between the two rules are represented by the possible runtime paths: if the `forall` rule finds no element fulfilling the parameter property, it does not run the ASM Rule at all, but continues execution, while the `choose` rule fails when no elements are present, failure handling will follow the unsuccessful matching. A further difference is, that the ASM Rule of the `choose` rule at most once, while in case of the `forall` rule it can rule (depending on the model space) an arbitrary number of times - but it is not needed to check the run of the ASM Rule several times for type checking, because running a single ASM Rule several times does not change the types (if the rule is not deterministic than inside the rule are branches created).

Taking these properties in consideration, the `choose` rule needs two branches: one, where a match is found, and the ASM Rule is executed, and another where no match is found, and a failure handling process is initialized. In this case the ASM Rule is not executed.

Example 15 *To illustrate the handling of the* `choose` *rule, let's consider the rule presented in Listing 3.2:*

Listing 4.1 A Simple `choose` Rule

```
choose Token with find placeWithToken(Place, Token) do print("token found");
```

First of all the static checker has to evaluate the pattern call (the `placeWithToken` *call), the examination is detailed in Section 4.6.2. A successful matching (first branch) binds the Place and Token parameters to be able to use it later. In case of unsuccessful matching (second branch) a* `fail` *is emitted.*

For type checking the `forall` rule also two branches are needed: in the first one the ASM Rule is not executed, in the second one it is executed once.

It is an interesting point that after the run of a `forall` rule there should be no model element fulfilling the condition of the rule (except when the step creates such nodes, or there are conflicting applications). It needs further research whether these observations could be mapped into constraint in order to extend the number of detected faults types.

Table 4.7 contains the handling of the collection iterator nodes.

Rule	Subnodes	Constraints	Branches
Choose	Arbitrary number of variables, a condition, a rule and a fail node	The variables are model elements, the condition is boolean term	(1) condition and fail nodes are traversed; (2) condition and rule is traversed
Forall	Arbitrary number of variables, a condition and a rule node	The variables are model elements, the condition is boolean term	(1) only the condition is traversed (2) the condition and the rule is traversed.

Table 4.7: The Analysis of Iteration Rules

4.6 Traversing GT Rule and Pattern Nodes

4.6.1 Calling Graph Patterns

The VTCL language contains Graph Pattern Calls as boolean ASM Terms. This allows its use both inside Graph Patterns and in ASM Rules as conditions. The returned value of the call is true, if the pattern matching is successful.

To handle recursive calls, the *depth limit* (see Section 3.2) is applied to the bodies of the called graph patterns. This way it is possible to analyse possible sequences of the alternate bodies of graph patterns.

The `pattern call` term's responsibility is to match the called pattern's symbolic parameters with the actual parameters given in the call node. Because the pattern contain a static condition, it is not required to do this pattern matching twice (as in case of ASM 4.5.1 or GT 4.6.3 Rule calls).

4.6.2 Graph Patterns

A graph pattern is the conjunction of conditions (the negative pattern acts a logical inverse operator over this conjunction). This basically means it is enough to translate the single conditions to constraints, and the conjunction of these constraints will be the constraint of the graph pattern.

A graph pattern has parameter and local variables: the parameter variables represent a selection of variables which have to be matched, while the local variables are used as internal variables, they are helpful for describing more complex patterns. During the processing of graph patterns it is not needed to differentiate between the two variable types, they can be handled the same way.

It is possible to define alternate bodies for a graph pattern: these bodies define

disjunctive conditions: the pattern matches if at least one of its bodies matches. To handle these bodies, a new branch should be created for every body.

Type definition states that a variable is an instance of a metamodel element. It can be translated into *substitutable* relation.

Checking of a boolean formula states, the a boolean formula holds. The formula is an ASM Term, it has to be evaluated, and its type should be `Boolean`.

Pattern Calls are used to define subpatterns. These subpatterns can be handled as additional conditions and constraints. Together with the alternate body construct pattern calls are used to write recursive patterns (similar to the recursive clauses in Prolog).

A pattern call (both inside or outside the pattern) is responsible for parameter matching: from the call node we are able to extract the variables known to the caller, and it has to generate constraints stating the type equality of every parameter of the callee and the variable from the caller.

For this simple parameter matching it is required to copy the subgraph representing the pattern to every call. This solution also allows the handling of the branches of the same way as every other node. If instead of the copying it would be only referenced, it should be ensured that the different invocations could choose different branches (the branches are calculated for nodes, if there is only a single node representing the pattern there will be only a single branching point).

Example 16 *The called pattern of Listing 4.1 is described in Listing 4.2.*

Listing 4.2 A Simple Graph Pattern

```
pattern placeWithToken(PlaceVar, TokenVar) =
{
       'PetriNet'.'Place'(PlaceVar);
       'PetriNet'.'Place'.'Token'(TokenVar);
       'PetriNet'.'Place'.'tokens'(X, PlaceVar, TokenVar);
}
```

The pattern describes a relation between two elements, called PlaceVar *and* TokenVar*. The lines of the pattern describe in order, that (1)* PlaceVar *is an element of* `Place` *(from the metamodel), (2)* TokenVar *is an element of* `Token`*, and (3) there is a variable called* X*, which represents a* `tokens` *relation between PlaceVar and TokenVar.*

Although the type of the variable X *is not used outside the pattern the static checker calculates it, the node-based constraint extraction process is not capable of detecting these redundancies.*

Table 4.8 summarizes the handling of the elements of the GT Patterns.

Element	Subnodes	Constraints	Branches
GT Pattern	Arbitrary number of bodies	No additional constraints	One for every body
Pattern Body	Arbitrary number of called patterns, Arbitrary number of Pattern Elements	Type Equals Constraint for every pattern element	One

Table 4.8: The Elements of GT Patterns

4.6.3 Calling Graph Transformation Rules

The `GT Rule Invocation` rule is used to call Graph Transformation Rules in the VTCL language. These calls can be recursive, because Graph Transformation rules may contain ASM rules, so the *depth limit* (see Section 3.2) should be applied to the called rules.

The `GT Rule Invocation` rule's responsibility is to match the called rules symbolic parameters with the actual parameters given in the call node. This parameter matching must happen both before and after the call takes place, because only this way is it possible to handle the changes of the variables inside the called rules. These parameter matching should consider the parameters direction (`in`, `out`, `inout`) to update only those variables that can be changed.

4.6.4 Graph Transformation Rules

A GT Rule describes a single graph transformation step. The description include the graph patterns: a pattern describing the LHS graph (`precondition`) and another for the RHS graph (`postcondition`). The rule may also have directed (`in`, `out`, `inout`) parameters.

It is also possible that the GT Rule contains an optional ASM Rule `action` which is applied to the matched precondition pattern.

Both the graph patterns and the action have to be traversed, because the parameters (and thus the type of the parameters) of the graph transformation are constrained by the patterns. The analyser first traverses the patterns and only then the action as the graph patterns provide full type information that is useful during the analysis of the action.

Example 17 *Listing 4.3 displays the* ***`addToken`*** *GT Rule introduced in Section 4.6.4 with a slight alteration: both the postcondition and action part has been defined in order to demonstrate the traversal.*

Listing 4.3 The addToken GT Rule

```
// Adds a token to the place 'Place'.
gtrule addToken(in Place) =
{
  precondition find place(Place)
  postcondition find placeWithToken(Place, Token)
  action{
    print(Place);
    print(Token);
  }
}
```

Rule	Subnodes	Constraints	Branches
GT Rule (with RHS)	Precondition pattern, Postcondition pattern	No additional constraints	One
GT Rule (with action)	Precondition pattern, Action	No additional constraints	One
GT Rule (with both)	Precondition pattern, Postcondition pattern, Action	No additional constraints	One

Table 4.9: The Analysis of GT Rules

When investigating this GT Rule node, first the precondition, then the postcondition pattern is traversed as described in Section 4.6.2, then the rule is also executed as described in Section 4.5.

Table 4.9 displays the parameters of the GT Rule nodes.

4.7 The Detected Type Handling Problems

By using the terminology introduced in Section3.5 the analyser is capable of detecting three kind of (possibly) invalid type handling:

1. In case of *constraint failures* one of the CSP variables has an empty domain, that means the type constraints connected to that variable are inconsistent. This can translated to inconsistent type handling in the VTCL code. The severity of this problem is *error*.

2. If there are no constraint failures, the analyser looks for*inconsistencies*: for every VTCL variable to representing TPM variables are assembled, and their calculated types are compared. If there is a change of types between the

types, a *warning* is issued, because in most cases it is not recommended to use a variable with multiple types during it's lifecycle.

3. A common *traversal fault* is an unhandled failure node. When finding one, a *warning* is issued, as it is considered as a bad practice to leave unhandled possible failures in the code.

4.8 Related Work

While there is already a large set of static type checking concepts in the literature, below we focus on providing a brief overview with two different application areas that show conceptual similarities with our approach.

There are similar problems involving XML [12] and XSLT [15]. XML is a hierarchical data structure, which can be though of as a data structure, while XSLT is a transformation language, with rules given to apply to various XML values. A type specification of an XML document is written in a DTD (Document Type Definition) (or XSD (XML Schema)), and can express types such as a node of type HTML contains a head followed by a body. The question is: Given a DTD for an output document, and an XSLT transformation, what is the DTD for the input document? The advantage of this knowledge is that a document can be checked to meet an output DTD without the cost of transformation first, and the errors can be determined in the input (or source) document, which the user wrote not a document generated by a transformation. One of the most widely known *answer* [42] treats this as a question of backward type inference. A type is synthesized as a finite tree automaton, and is deduced compositionally. We adopted the tree based structural traversal from the approach that works on our models.

As for the functional language community, type systems have played significant roles in guaranteeing better software safety. Most notably the well known Hindley-Milner [39] algorithm for lambda calculus that reduces the typing problem to a unification problem of equations. It serves as the underlying algorithm [35, 30] for many approaches developed for Haskell [5] and Erlang [3], todays most widely used functional programming languages. Our approach was influenced by the work started in [36], which translates the typing problem to a set of constraints. As Hindley-Milner is designed to lambda calculus which is not conform to the VIATRA2 transformation language, we designed a different mapping and evaluation approach that fitted better with the graph based data structures and multi-level metamodeling.

4.9 Summary

In this chapter we have discussed the possibility how to use our proposed static analysis method to implement and integrate a type checker component into the VIATRA2 model transformation framework. The created framework is capable of checking the type safety of the VIATRA2 VTCL language by inferring the type information not present and checking for contradictions.

To create this type checker, we used a gene algorithm by Yves Caseau to map the multiple inheritance hierarchy into integer sets, thus making it representable in a finite domain CSP solver.

Then we described, how to handle the node types during the traversal, and finally gave an overview of the problem types analysed by the method.

Chapter 5

Evaluating the Type Checker

In this chapter we demonstrate the fault detection capabilities of the static analyser tool on a larger example, and then we measure the execution time and memory usage.

As examples we use two transformation programs over the domain of Petri nets and the implementation of the AntWorld case study. In this chapter we give an introduction to these transformation programs, then test the failure detection capabilities of our type checker, finally the performance of the tool is evaluated.

5.1 The Used Transformation Programs

5.1.1 Petri net Transformation Programs

These transformation programs are defined in [9] over the domain of Petri nets, they work well as examples because the control structure of the program contains a lot of elements of the language. The programs are:

The Simulator program alters existing Petri net models by enabling firing steps that move existing tokens between the places of the net. The firing is managed by two steps: (i) by the usage of the pattern `isTransitionFireable` a fireable transition is selected non-deterministically then (ii) the GT Rules `addToken` and `removeToken` calls are used to manage the changes of `Token`s in the model.

During the firing process the structure of the model (the `Place` and `Transition` entities, and the `Inarc` and `Outarc` relations) remains unchanged.

The source code of the transformation program is described in Appendix A.1.

The Generator program creates new Petri net models. The creation starts from a small, living Petri net, then the inverse of six reduction operations [31]

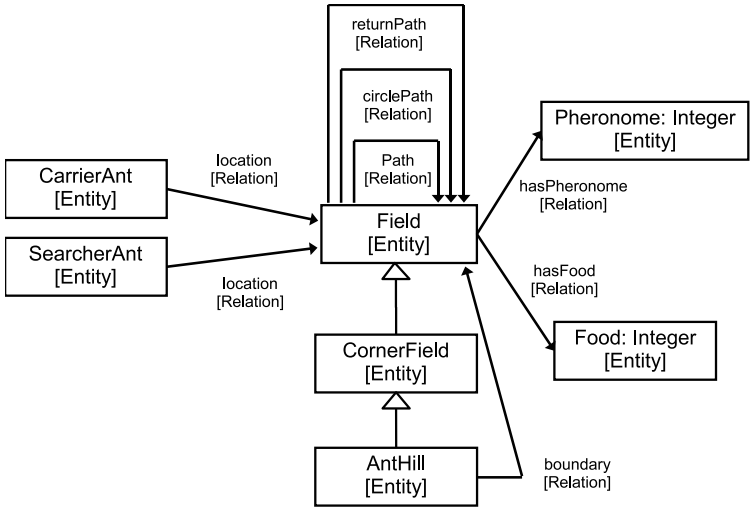


Figure 5.1: The Simplified Metamodel of the AntWorld Case Study

that preserve the liveliness and safety properties of the net together with a weighted random operation selection. The generation is parametrized to produce approximately equal number of places and transitions.

The output is available in PNML format [26]. The source code of the transformation is described in Appendix A.2.

5.1.2 The AntWorld Case Study

To demonstrate the analysis capabilities together with the Petri net examples a larger transformation program with a more complex metamodel is also evaluated. The AntWorld case study [44] is a model transformation benchmark featured at GraBaTs 2008 [4]. AntWorld, probably inspired by Ant Colony Optimalization [19], simulates the life of a simple ant colony searching and collecting food to spawn more ants on a dinamically growing rectangular world. The ant collective forms a swarm intelligence, as ants discovering food sources leave a pheromone trail on their way back so that the food will be found again by other ants.

Figure 5.1 contains the metamodel of the AntWorld case study. The *Field* element represents a field of AntWorld (*grid node* in the original specification, but is renamed in the implementation to avoid confusion), they are connected by *Path*s, and a set of field created in a single step are connected via *circlePath* relations.

Fields may associated with an Integer number of *Pheromone* and *Food*. Finally they may contain two types of ants: *searcherAnts* that do not carry food, instead are looking for a food bundle and *carrierAnts* that carry food back to the anthill.

Listing 5.1 describes how a round is managed in the simulation. A round consists

Listing 5.1 A Round in the AntWorld case study

```
236     rule doRound() = let Hill = antHill() in seq {
237             //Ant actions
238             iterate choose Ant, LocationEdge, Food, Field with find canGrab(Ant, LocationEdge,
                    Food, Field) do call grab(Ant,LocationEdge,Food,Field);
239             forall Ant, LocationEdge with find hasCarrierAnt(LocationEdge, Hill, Ant) do call
                    deposit(Hill,Ant,LocationEdge);
240             forall Ant, FromField, HA1 with find hasCarrierAnt(HA1, FromField, Ant) do
241                     choose NewField with find alongReturnPath(FromField, NewField) do seq {
242                             call moveAnt(HA1, NewField);
243                             call leavePheromone(FromField);
244                     }
245             forall Ant with find searcher(Ant) do call search(Ant); // two kinds of search
246             // only searchers can breach the boundary!
247
248             //Field action
249             forall Pheromone with find pheromone(Pheromone) do call evaporate(Pheromone);
250
251             iterate
252                     if(toInteger(value(Hill)) > 0)
253                             call consume(Hill);
254                     else
255                             fail;
256             if (find boundaryBreachedBySearcher()) call growGrid();
257
258     }
```

of seven phases, four for the ant simulation and three for managing the world. All phases are captured as a series of `forall` and `choose` rules guarded by graph patterns. The phases are the following:

Grab phase for every *searcher ant* standing on a food bundle the `grab` rule is called that collects as many food from that source as the ant can carry, also managing the size of the food bundle. If the food bundles size becomes negative, it is deleted from the model. The ant becomes a *carrier ant*.

Deposit phase for every *carrier ant* standing in the anthill the `deposit` rule is called that increases the food reserves of the hill by the food carried by the ant, and the ant becomes a *searcher ant*. This rule is the inverse of the Grab phase.

Return phase for every carrier ant not in the anthill the `returnPath` relation is used to determine the next location while going back to the hill. The `moveAnt` rule is called that moves the ant to the new location after the `leavePheromone` call is used to increase the pheromone level of the current position.

Search phase for every *searcher ant* the `search` rule is called. That rule selects a next location; if there is high Pheromone level on a neighboring field, it becomes the new target, else a random location is selected.

Evaporate Pheromon phase for every *field* with pheromone the `evaporate` rule is called that reduces the amount of pheromone available on the field. If the amount of pheromone becomes negative, it is removed from the model.

Create Ants phase as long as the *hill* contains food (its integer value is greater than zero), the `consume` rule is called that reduces the value by one, and creates a new *searcher ant* to the hill.

Boundary Breached phase if the `boundaryBreachedBySearcher` pattern matches, the `growGrid` rule is called that creates new fields. A circular based traversal of the boundary fields is used the generate the new fields using the *circlePath* relations. Along the new fields also new food bundles are created on every tenth created field.

The full source code of the transformation program is available in Appendix A.3.

5.2 Evaluation of the Static Type Checker

To evaluate the fault detection capabilities of the static type checker, some errors are injected into the *Simulator* and *Generator* programs. The source listings in these section show only the modified code segments, the entire fault-free source code can be found in Appendix A.1 and Appendix A.2 accordingly.

It is important to note that the parser of the VIATRA2 framework contains a built-in static analyser. That analyser is capable of detecting faults not limited to the type safety, but only a small subset of the typical faults: all the transformation programs described in this section pass the analysis of the parser. All together the goal of our analysis solution is not to replace error checking of the parser but to increase the fault coverage when using both.

Listing 5.2 shows a small modified block from the *Simulator* program - the only difference is that in line 94 the `iterate` rule is missing before the `seq` rule, so there is no failure handling rule for the elements in the block.

When analysing this example three messages are filled into the *Problems View* of the IDE, as can be seen in Figure 5.2. These messages state, that the (potential) failures in the `fail` rule in line 96, the `choose` rule in line 97, and another `choose` rule inside the `fireTransition` call are not handled.

By restoring the deleted `iterate` rule as seen in the second part of Listing 5.2, these messages are removed.

Our second example also comes from the *Simulator* program. The second creation rule of the `addToken` rule (see Listing 5.3) has been altered to create a variable with the same name as the first one.

Listing 5.2 A Missing Fail Handler Rule

The Context of the Fault:

```
94  let PN = ref(NetFQN) in seq {
95       update counter("iterations") = counter("iterations") + 1;
96       if (counter("iterations") > Iterations) fail;
97       choose T with find fireable(PN,T) do seq {
98         call fireTransition(T);
99       }
100    }
```

The Fix of the Problem:

```
94  let PN = ref(NetFQN) in iterate seq {
```

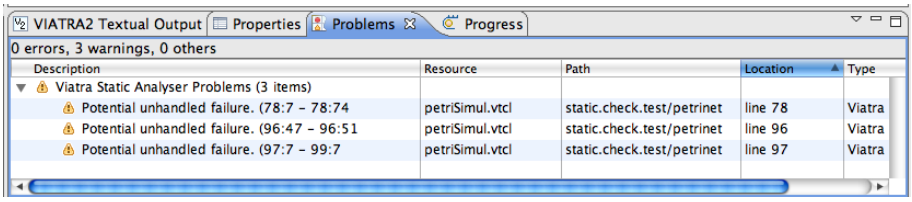


Figure 5.2: The Detected Faults in the Problems View

The analyser detects this kind of misuse, because the calculated type of the variable `Token` changes from the Entity `Token` to the Relation `tokens`. The correct create rule is described in the second part of Listing 5.3.

The last example comes from real development experience: during the development of the *generator* program once an invalid pattern call was used: at line 313 a `petriPlace` pattern is called instead of the `petriTransition` as described in Listing 5.4.

When running the type checker on the program it reported an error related to the variable `P`, because in the pattern call `petriPlace` it must have been a `Place`, while in the rule call `inverseSerialTransitionRedaction` a `Transition`. These conditions cannot hold both. A more careful examination showed that the pattern call has to be replaced as described in the second part of Listing 5.4.

We also used the static type checker to analyse the implementation of the AntWorld case study. We only got the final version of the implementation, and the analysis did not find type errors.

Listing 5.3 An Invalid Creation Rule

The Context of the Fault:

```
70 rule addToken(in Place) = let Token = undef, X = undef in seq {
71     new('PetriNet'.'Place'.'Token'(Token) in Place);
72     new('PetriNet'.'Place'.'tokens'(Token, Place, Token));
73   }
```

The Fixed Problem:

```
72     new('PetriNet'.'Place'.'tokens'(Token, Place, Token));
```

Listing 5.4 An Invalid Pattern Call

The Context of the Fault:

```
313 choose P below PN with find petriPlace(PN,P) do call inverseSerialTransitionRedaction(P, PN);
```

The Fixed Problem:

```
313 choose P below PN with find petriTransition(PN, P) do call inverseSerialTransitionRedaction(P, PN);
```

5.3 Benchmarking the Static Type Checker

5.3.1 The Measurement Environment

The runtime performance of the type checker is approximated by the execution time of the traversal: the omission of the TPM building process is intentional, because its execution is much faster then the several traversal iterations. The execution time of running each branch and the total time is measured several times, and the result are averaged.

The memory consumption of the analysis process is also measured. This is done by watching the heap size of the Java Virtual Machine (JVM); the heap also contains the VIATRA2 framework, so before the benchmark the heap size should be remembered, and they have to be deducted from the measured value.

It is important to note that the SICStus Prolog engine uses a different Virtual Machine so its memory usage is not measured by the JVM heap size. In this benchmark the memory usage of the Prolog VM is not collected.

The measurements have been carried out by on a MacBook notebook computer with a 2 GHz Intel Core 2 Duo processor and 3 GB of system RAM available, running Mac OSX 10.5 version 1.5.0_16 of the 32 bit Java SE runtime (for the VIATRA2 framework), and version 4.0.4 of the SICStus Prolog, and version 2.2 of the Gecode/J engine is used in the corresponding measurements.

Execution times were measured with millisecond precision as allowed by the related system calls; the memory consumption is measured with megabyte precision.

5.3.2 Benchmarking the Simulator Program

The Petri net Firing transformation program is very simple: in total 4 branches are needed during traversal, and at most 61 variables are identified in a branch. That was represented in the memory consumption of the analysis: in every iteration it consumed 1–2 MB RAM (the relative error of the measurement can be very high, no conclusion should be taken from these values), except the Gecode-based solver that used about 15–16 MB.

The execution times of the Petri net firing program has been measured for all three solvers. The results are shown in a chart in Figure 5.3. It is important to note that the scale of the X axis is logarithmic in order to display the results of the solvers side by side.

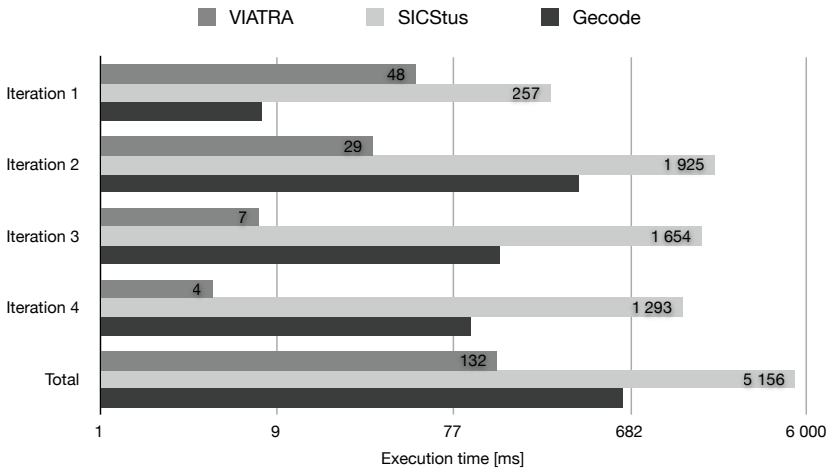


Figure 5.3: The Execution Time of the Analysis of the Firing Program

It may come as surprise that the SICStus solver is slower by an order of magnitude, but there are several differences which may explain it: the SICStus solver does not run in the JVM, and it does not support the iterative building of the constraint satisfaction problem - both of these factors might cause performance issues.

The Gecode engine performed between the other solvers, but used much more memory. The engine supports iterative building, but the used set implementation is very naive thus uses an enormous amount of memory.

5.3.3 Benchmarking the Generator Program

The generator program is larger than the firing program: it is described by 39 branches with at most 221 variables.

When benchmarking the SICStus solver with this program it was not capable of evaluating the first branch in half an hour, so it is excluded from this test.

The Gecode solver was also uncapable of analysing this generator program: although 18 branches were evaluated in an average of 60 s, then it was not able to allocate enough memory, and failed. The analyser used about 700 MB RAM at this point.

Our solver implementation needed less than two seconds (about 47 ms per branch) for the complete evaluation, and about 14 MB of RAM during execution.

The generator program seems large enough to also test the performance of the *state saving and restore* enhancement introduced in Section 3.3.4. This enhanced method needs about 35 MB memory, but surprisingly takes more time as the simple traversal: more than 5600 ms (about 144 ms per branch, almost three times as much time).

To illustrate the runtime characteristics, 6 groups were created from the branches based on the time the analyser spent on them. The boundaries of the groups were choosen in a way to cover the values of the runtimes evenly. Figure 5.4 shows how many branches belong to each group for both traversal methods.

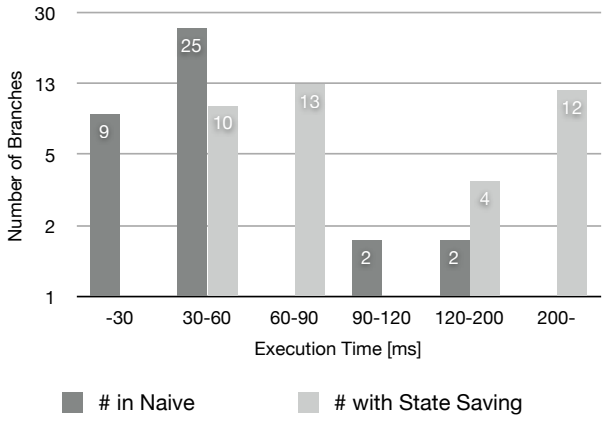


Figure 5.4: The Effect of State Saving on the Execution Time

As the chart displays, that there are only 4 branches using the naive method needing more than 90 ms to execute, while there are 12 using state saving that need more than 200 ms. By looking the groups of smaller runtimes, the result is similar. A basic interpretation of the phenomenon is that the cloning of the constraint space is a more expensive operation that building another one from scratch.

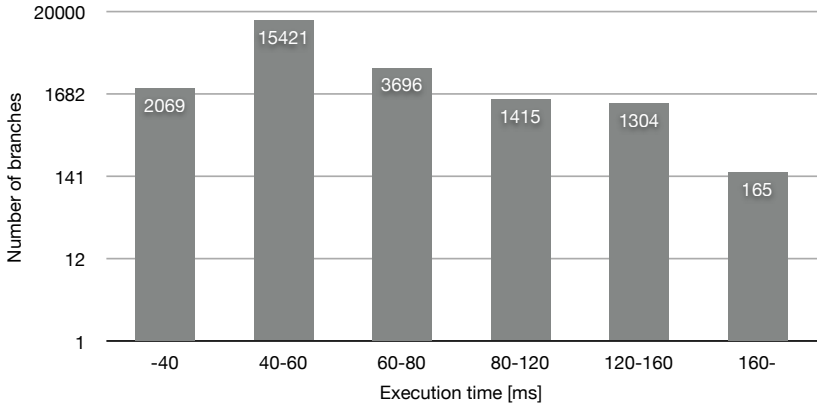


Figure 5.5: The Execution Times of the Different Branches

5.3.4 Benchmarking with the Antworld Program

The Antworld program is the largest example: 24270 branches with an average of 531 variables (with a maximum of 670) are needed to traverse every possible path.

As this example is that large, only the our solver implementation was tested without the state saving. The analyser used about 15 MB memory during the analysis.

The entire analysis needed about 24 minutes, a branch was evaluated in about 60 ms. A more detailed analysis of the time needed for running the branches is depicted in Figure 5.5.

This chart displays a similar branch grouping as described at the benchmark of the *generator* program.

When comparing the results of the two programs, it is interesting, that the memory usage of the two program analysis is about the same. We believe this is caused by the fact that more CSP problems can be stored in this amount of memory, and after the execution of a branch the related problem can be garbage collected, but that happens only when the heap is about to get full. We could not verify this theory by a further reduction of the heap size, because the VIATRA2 framework needs a similar amount of memory the load the transformation programs, so the heap space is already reserved at the start of the analysis.

On the other hand, a small increase in the analysis time is observable (from 47 ms to 60 ms per branch); this increase is similar in size to the increase of number of constraints, but much smaller than the increase of variables.

The overall runtime of the Antworld program is much higher as it contains 600 times as many branches that has to be evaluated. The conclusion of this result is that the number of branches can become a performance bottleneck, so a way to

reduce the number of branches in large programs is an important research task.

5.4 Summary

In this chapter we evaluated the capabilities and performance of the implemented static type checker component, and demonstrated its fault handling capabilities on the example of two transformation programs.

As we noted during our research the different parts of the transformation language acts differently with regards of the type handling as type information is present in the GT rules and patterns while the control structure used in VIATRA2 is untyped. As the testing of the component showed the type information coming from the GT part could ease the type inference in the control structure.

The performance evaluation shows that the main bottleneck could be the number of branches to check: the checking of a branch can happen in a reasonably short time, but in case of a large number of branches needed, this solution would be incapable of finishing the analysis fast enough for a wide usage. This means a way has to be found to reduce the number of branches to check.

Chapter 6

Results and future plans

6.1 Main Results

I designed and developed a static analysis technique using Constraint Satisfaction Programming to provide type checking for the (partially) untyped transformation language of the VIATRA2 framework.

- I specified a method for mapping ASM-driven model transformation control languages (both the GT Rules and the ASM Control Structure) into constraint satisfaction problems. This mapping can be easily adapted to other languages, and is able to manage the potential changes in the language specification.

- I used the defined mapping as a basis of a general static analysis method for transformation programs.

- I adapted the gene algorithm of Yves Caseau to the multi-level metamodeling hierarchy to represent the type system. The result of the algorithm is a representation of the metamodel elements as integer sets. This allowed the description of the hierarchy for the finite domain CSP solver.

- I implemented a simple finite domain CSP solver capable of handling integer and integer set variables and the constraints needed for the analysis.

- I defined and implemented a static type checker component based on the static analysis method for the VIATRA2 framework. The component is able to detect some faults that can be hard to detect by reading the source code.

For the implementation parts a total of 10000 lines of Java code has been written.

6.2 The Limitations of the Technology

The usage of the CSP technology brings forth some limitations which must be taken care of.

One of such limitation is the lack of error cause detection. The problem is very hard to solve over the general domain, because any possible subset of the entered constraints can be the root cause of the failure. But using knowledge about the transformation programs it is possible to do more specific detection.

It is also important to understand that CSP solvers stop execution when the first error is found, because they assume no solution might be found from this point. This is a sound assumption, as the constraints are monotonic: there are no constraints that increase the allowed domain of a constraint variable. This means for the static analysis process that it is not guaranteed to find every fault in the program in a single run. On the other hand, when fixing the faults one by one, the analysis can be re-run to check for other faults; and it is possible that multiple errors are detected, if the detected error does not cause the solver to fail (e.g. the results of multiple CSP variables is inconsistent).

The use of finite domain sets can also be problematic: in order to reduce memory consumption, the implementations might use some optimization's - which may or may not be sufficient. If the implementation is not accurate enough for the model (which was the case with the Gecode solver), a naive replacement implementations may require an enormous amount of memory. This was the main reason for implementing my own CSP solver.

6.3 Future plans

This master thesis shows a proof-of-concept solution for building a static analyser for ASM-driven MT system based on a CSP solver. As the first implementation version I built a type checker using this technology. The method can be further enhanced several ways, some are listed below.

6.3.1 New Analysis Methods

Reachability Analysis As of now the traversal algorithm always start from the root of the TPM, and the direction of the traversal always points to the direction of leaves in the order as the TPM nodes suggest them (according to the run paths). The semantics of traversing in the opposite direction is a search for conditions of reaching the current node. This semantic could be used for the detection of a dead path (if the conditions are contradictory the node cannot be reached) to support dead path elimination.

Invariant Matching There can be some invariants detected during the traversal, e.g. after a successful execution of a single `choose` in an `iterate` rule the condition of the rule must not hold for any element in the model space. These invariants can be used as constraints. These extra constraints could improve both the accuracy of the analysis and performance of the tool.

6.3.2 Increasing Performance

Partial Evaluation In order to avoid a much higher memory consumption, we used a branching strategy. To reduce the number of branches, some kind of partial evaluation can be used at several cases. One of these cases are disjunctive graph patterns. If we could determine that the constraints of a branch always hold, it is unnecessary to traverse the other branches as well. Another usage scenario of partial evaluation is the mapping of parallel rules: by detecting the conflicting subrules the number of needed branches can be reduced.

Modularizing the Traversal The traversal could be splitted by the call nodes, evaluating the called subtree, and replacing it with some kind of *contract* that contains all the aggregated constraints from the called subtree which are relevant to the caller. This enhancement could increase the performance of the analysis, because the same subtrees do not need to be traversed several times, and the number of branches might decrease. On the other hand replacing a subtree with a contract could decrease the fault-detecting capabilities of the static analyser by reducing the amount of available information.

6.3.3 More Specific Error Detection

Reimport Constraints in case of Failure In order to allow the better identification of the cause of an already detected fault, it might be possible to reimport some already imported constraints (into a new problem space). The analysis of some specific subsets could help to identify a set of related constraints that are unsatisfiable. This concept needs further research, especially the selection of the constraints to reimport.

Explanation Calculation It is also possible to locate the cause of a fault by calculating explanations [27] in the constraint solver. The explanations are a set of contradictory constraints. The calculated explanation can be showed to the transformation developer (after interpreting it in the problem domain) to help him/her to find the fault.

Appendix A

The Analysed Transformation Programs

A.1 The Petri Net Simulator Program

```
1  namespace DSM.machines.PetriNet;
2
3  import DSM.metamodel.PetriNet.PetriNetEditor;
4
5  @incremental
6  machine 'PetriNetSimulator' {
7    // 'Transition' is a transition of the petri net 'PN'.
8    pattern petriTransition(PN, Transition) = {
9      'PetriNet'(PN);
10     'PetriNet'.'Transition'(Transition);
11     'PetriNet'.'transitions'(X, PN, Transition);
12   }
13
14   // 'Place' is a source place for transition 'Transition'.
15   pattern sourcePlace(Transition, Place) = {
16     'PetriNet'(PN);
17     'PetriNet'.'Transition'(Transition);
18     'PetriNet'.'transitions'(X1, PN, Transition);
19     'PetriNet'.'Place'(Place);
20     'PetriNet'.'places'(X2, PN, Place);
21     'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
22   }
23
24   // 'Place' is a target place for transition 'Transition'.
25   pattern targetPlace(Transition, Place) = {
26     'PetriNet'(PN);
27     'PetriNet'.'Transition'(Transition);
28     'PetriNet'.'transitions'(X1, PN, Transition);
29     'PetriNet'.'Place'(Place);
30     'PetriNet'.'places'(X2, PN, Place);
31     'PetriNet'.'Transition'.'InArc'(InArc, Transition, Place);
32   }
33
34   // 'Place' contains a token 'Token' linked to it
35   pattern placeWithToken(Place, Token) = {
```

```
36      'PetriNet'.'Place'(Place);
37      'PetriNet'.'Place'.'Token'(Token) in Place;
38      'PetriNet'.'Place'.'tokens'(X, Place, Token);
39    }
40
41    // Transition is fireable
42    pattern isTransitionFireable_flattened(Transition) = {
43      'PetriNet'.'Transition'(Transition);
44      neg pattern notFireable_flattened(Transition) = {
45        'PetriNet'.'Place'(Place);
46        'PetriNet'.'Transition'(Transition);
47        'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
48        neg pattern placeToken(Place) = {
49          'PetriNet'.'Place'(Place);
50          'PetriNet'.'Place'.'Token'(Token);
51          'PetriNet'.'Place'.'tokens'(X, Place, Token);
52        }
53      } or {
54        'PetriNet'.'Place'(Place);
55        'PetriNet'.'Transition'(Transition);
56        'PetriNet'.'Place'.'InhibitorArc'(OutArc, Place, Transition);
57        'PetriNet'.'Place'.'Token'(Token);
58        'PetriNet'.'Place'.'tokens'(X, Place, Token);
59      }
60    }
61
62    // Transition is fireable in PetriNet
63    pattern fireable(PetriNet, Transition) = {
64      find petriTransition(PetriNet, Transition);
65      find isTransitionFireable_flattened(Transition);
66      'PetriNet'(PetriNet);
67      'PetriNet'.'Transition'(Transition);
68    }
69
70    rule addToken(in Place) = let Token = undef, X = undef in seq {
71      new('PetriNet'.'Place'.'Token'(Token) in Place);
72      new('PetriNet'.'Place'.'tokens'(X, Place, Token));
73    }
74
75
76    rule fireTransition(in Transition) = seq {
77      forall Place with find sourcePlace(Transition, Place) do
78        choose Token with find placeWithToken(Place,Token) do delete(Token);
79      forall Place with find targetPlace(Transition, Place) do
80        call addToken(Place);
81      update counter("firings") = counter("firings") + 1;
82    }
83
84    asmfunction counter/1;
85
86    // entry point
87    rule main(
88          in NetFQN, // fully qualified name of the entity representing the Petri-net model
89          in Iterations // number of firings to be executed
90    ) = let Start = 0 in seq {
91      update counter("iterations") = 0;
92      update counter("firings") = 0;
93      update Start = systime();
94      let PN = ref(NetFQN) in iterate seq {
95        update counter("iterations") = counter("iterations") + 1;
96        if (counter("iterations") > Iterations) fail;
97        choose T with find fireable(PN,T) do seq {
```

```
98          call fireTransition(T);
99        }
100     }
101     println("Simulation ended, fired " +
102       counter("firings") + " transitions in " +
103       (counter("iterations")-1) + " iterations in "+
104       (systime()-Start)+ " msec.");
105   }
106
107 }
```

A.2 The Petri Net Generator Program

```
1  namespace DSM.machine.PetriNet;
2
3  import DSM.metamodel.PetriNet.PetriNetEditor;
4
5  @incremental
6  // All transformation for Petri Net simulation
7  machine 'PetriNetGenerator'
8  {
9          // 'Transition' is a transition of the petri net 'PN'.
10         pattern petriTransition(PN, Transition) = {
11                 'PetriNet'(PN);
12                 'PetriNet'.'Transition'(Transition);
13                 'PetriNet'.'transitions'(X, PN, Transition);
14         }
15
16         @Random
17         pattern petriPlace(PN,Place) = {
18                 'PetriNet'(PN);
19                 'PetriNet'.'Place'(Place);
20                 'PetriNet'.'places'(X2, PN, Place);
21         }
22
23         // 'Place' is a source place for transition 'Transition'.
24         pattern sourcePlace(Transition, Place, OutArc) = {
25                 'PetriNet'(PN);
26                 'PetriNet'.'Transition'(Transition);
27                 'PetriNet'.'transitions'(X1, PN, Transition);
28                 'PetriNet'.'Place'(Place);
29                 'PetriNet'.'places'(X2, PN, Place);
30                 'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
31         }
32
33         // 'Place' is a target place for transition 'Transition'.
34         @Random
35         pattern targetPlace(Transition, Place, InArc) = {
36                 'PetriNet'(PN);
37                 'PetriNet'.'Transition'(Transition);
38                 'PetriNet'.'transitions'(X1, PN, Transition);
39                 'PetriNet'.'Place'(Place);
40                 'PetriNet'.'places'(X2, PN, Place);
41                 'PetriNet'.'Transition'.'InArc'(InArc, Transition, Place);
42         }
43
44         // 'Place' contains a token 'Token' linked to it
45         pattern placeWithToken(Place, Token) = {
46                 'PetriNet'.'Place'(Place);
47                 'PetriNet'.'Place'.'Token'(Token) in Place;
```

```
48              'PetriNet'.'Place'.'tokens'(X, Place, Token);
49      }
50
51      pattern token(Token) = {
52              'PetriNet'.'Place'.'Token'(Token);
53      }
54
55      pattern placeWithToken2(Place) = {
56              'PetriNet'.'Place'(Place);
57              'PetriNet'.'Place'.'Token'(Token) in Place;
58              'PetriNet'.'Place'.'tokens'(X, Place, Token);
59      }
60
61      asmfunction counter/1;
62
63      rule generatePlaceRestriction(in PN) = let Temp = undef in seq {
64      forall Place below PN with find petriPlace(PN,Place) do
65              let NewPlace = undef, NewToken = undef in seq {
66                      new('PetriNet'.'Place'(NewPlace) in PN);
67                      new('PetriNet'.'places'(Temp, PN, NewPlace));
68                      new('PetriNet'.'Place'.'Token'(NewToken) in NewPlace);
69                      new('PetriNet'.'Place'.'tokens'(Temp,NewPlace,NewToken));
70                      forall Transition below PN, OutArc below PN with find sourcePlace(Transition,
                              Place, OutArc) do seq {
71                              new('PetriNet'.'Transition'.'InArc'(Temp,Transition,NewPlace));
72                      }
73                      forall Transition below PN, InArc below PN with find targetPlace(Transition,
                             Place, InArc) do seq {
74                              new('PetriNet'.'Place'.'OutArc'(Temp,NewPlace,Transition));
75                      }
76              }
77      }
78
79      rule inverseSelfPlaceLoop_rule(in Transition, in PN) =
80              let NewPlace = undef, NewOutArc= undef, NewInArc= undef,
81                      Temp = undef, NewToken = undef in seq{
82                              new('PetriNet'.'Place'(NewPlace) in PN);
83                              new('PetriNet'.'Transition'.'InArc'(NewInArc,Transition,NewPlace));
84                              new('PetriNet'.'Place'.'OutArc'(NewOutArc,NewPlace,Transition));
85                              new('PetriNet'.'places'(Temp, PN, NewPlace));
86                              new('PetriNet'.'Place'.'Token'(NewToken) in NewPlace);
87                              new('PetriNet'.'Place'.'tokens'(Temp,NewPlace,NewToken));
88                      }
89
90
91      gtrule inverseSelfPlaceLoop(out Transition, in PN) = {
92        precondition find petriTransition(PN, Transition)
93        action{
94              let NewPlace = undef, NewOutArc= undef, NewInArc= undef,
95               Temp = undef, NewToken = undef in seq{
96                      new('PetriNet'.'Place'(NewPlace) in PN);
97                      new('PetriNet'.'Transition'.'InArc'(NewInArc,Transition,NewPlace));
98                      new('PetriNet'.'Place'.'OutArc'(NewOutArc,NewPlace,Transition));
99                      new('PetriNet'.'places'(Temp, PN, NewPlace));
100                     new('PetriNet'.'Place'.'Token'(NewToken) in NewPlace);
101                     new('PetriNet'.'Place'.'tokens'(Temp,NewPlace,NewToken));
102                     }
103             }
104     }
105
106     rule inverseSelfTransitionLoop_rule(in Place, in PN) =
107             let NewTransition = undef, NewOutArc= undef, NewInArc= undef, Temp = undef in seq{
```

```
108                     new('PetriNet'.'Transition'(NewTransition) in Place);
109                     new('PetriNet'.'Place'.'OutArc'(NewOutArc,Place,NewTransition));
110                     new('PetriNet'.'Transition'.'InArc'(NewInArc,NewTransition,Place));
111                     new('PetriNet'.'transitions'(Temp, PN, NewTransition));
112             }
113
114
115     gtrule inverseSelfTransitionLoop(out Place, in PN) = {
116             precondition find petriPlace(PN, Place)
117             action{
118                     let NewTransition = undef, NewOutArc= undef, NewInArc= undef, Temp = undef in
                              seq{
119                             new('PetriNet'.'Transition'(NewTransition) in Place);
120                             new('PetriNet'.'Place'.'OutArc'(NewOutArc,Place,NewTransition));
121                             new('PetriNet'.'Transition'.'InArc'(NewInArc,NewTransition,Place));
122                             new('PetriNet'.'transitions'(Temp, PN, NewTransition));
123                     }
124             }
125     }
126
127     rule inverseParallelTransitionRedaction_rule(in Place, in PN, in NextPlace) =
128             let NewTransition = undef, NewOutArc= undef, NewInArc= undef, X1 = undef in seq{
129                     new('PetriNet'.'Transition'(NewTransition) in Place);
130                     new('PetriNet'.'Place'.'OutArc'(NewOutArc,Place,NewTransition));
131                     new('PetriNet'.'Transition'.'InArc'(NewInArc,NewTransition,NextPlace));
132                     new('PetriNet'.'transitions'(X1, PN, NewTransition));
133             }
134
135     pattern getPlace(Place, NextPlace, PN) = {
136             'PetriNet'(PN);
137             'PetriNet'.'Transition'(Transition);
138             'PetriNet'.'transitions'(X1, PN, Transition);
139             'PetriNet'.'Place'(Place);
140             'PetriNet'.'places'(X2, PN, Place);
141             'PetriNet'.'Place'(NextPlace);
142             'PetriNet'.'places'(X3, PN, NextPlace);
143             'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
144             'PetriNet'.'Transition'.'InArc'(InArc, Transition, NextPlace);
145     }
146
147     gtrule inverseParallelTransitionRedaction(out Place) = {
148             precondition pattern getPlace(Place, NextPlace, PN) = {
149                     'PetriNet'(PN);
150                     'PetriNet'.'Transition'(Transition);
151                     'PetriNet'.'transitions'(X1, PN, Transition);
152                     'PetriNet'.'Place'(Place);
153                     'PetriNet'.'places'(X2, PN, Place);
154                     'PetriNet'.'Place'(NextPlace);
155                     'PetriNet'.'places'(X3, PN, NextPlace);
156                     'PetriNet'.'Place'.'OutArc'(OutArc, Place, Transition);
157                     'PetriNet'.'Transition'.'InArc'(InArc, Transition, NextPlace);
158             }
159             action{
160                     let NewTransition = undef, NewOutArc= undef, NewInArc= undef, X1 = undef in
                            seq{
161                             new('PetriNet'.'Transition'(NewTransition) in Place);
162                             new('PetriNet'.'Place'.'OutArc'(NewOutArc,Place,NewTransition));
163                             new('PetriNet'.'Transition'.'InArc'(NewInArc,NewTransition,NextPlace))
                                    ;
164                             new('PetriNet'.'transitions'(X1, PN, NewTransition));
165                     }
166             }
```

```
167         }
168
169         rule inverseParallelPlaceRedaction_rule(in Transition, in PN, in NextTransition) =
170                 let NewPlace = undef, NewOutArc= undef, NewInArc= undef, X1 = undef in seq{
171                         new('PetriNet'.'Place'(NewPlace) in PN);
172                         new('PetriNet'.'Transition'.'InArc'(NewInArc,Transition,NewPlace));
173                         new('PetriNet'.'Place'.'OutArc'(NewOutArc,NewPlace,NextTransition));
174                         new('PetriNet'.'places'(X1, PN, NewPlace));
175                 }
176
177         @Random
178         pattern getTransition(Transition, NextTransition, PN) = {
179                 'PetriNet'(PN);
180                 'PetriNet'.'Transition'(Transition);
181                 'PetriNet'.'transitions'(X1, PN, Transition);
182                 'PetriNet'.'Transition'(NextTransition);
183                 'PetriNet'.'transitions'(X2, PN, NextTransition);
184                 'PetriNet'.'Place'(Place);
185                 'PetriNet'.'places'(X3, PN, Place);
186                 'PetriNet'.'Transition'.'InArc'(InArc, Transition, Place);
187                 'PetriNet'.'Place'.'OutArc'(OutArc, Place, NextTransition);
188         }
189
190         gtrule inverseParallelPlaceRedaction(out Transition) = {
191                 precondition pattern getTransition(Transition, NextTransition, PN) = {
192                         'PetriNet'(PN);
193                         'PetriNet'.'Transition'(Transition);
194                         'PetriNet'.'transitions'(X1, PN, Transition);
195                         'PetriNet'.'Transition'(NextTransition);
196                         'PetriNet'.'transitions'(X2, PN, NextTransition);
197                         'PetriNet'.'Place'(Place);
198                         'PetriNet'.'places'(X3, PN, Place);
199                         'PetriNet'.'Transition'.'InArc'(InArc, Transition, Place);
200                         'PetriNet'.'Place'.'OutArc'(OutArc, Place, NextTransition);
201                 }
202                 action{
203                         let NewPlace = undef, NewOutArc= undef, NewInArc= undef, X1 = undef in seq{
204                                 new('PetriNet'.'Place'(NewPlace) in PN);
205                                 new('PetriNet'.'Transition'.'InArc'(NewInArc,Transition,NewPlace));
206                                 new('PetriNet'.'Place'.'OutArc'(NewOutArc,NewPlace,NextTransition));
207                                 new('PetriNet'.'places'(X1, PN, NewPlace));
208                         }
209                 }
210         }
211
212         gtrule addToken(out Place, in PN) = {
213                 precondition find petriPlace(PN,Place)
214                 action {call addToken_Rule(Place);}
215         }
216
217         rule addToken_Rule(in Place) =
218                 let NewToken = undef, Temp = undef in seq{
219                         new('PetriNet'.'Place'.'Token'(NewToken) in Place);
220                         new('PetriNet'.'Place'.'tokens'(Temp,Place,NewToken));
221                 }
222
223         rule inverseSerialPlaceRedaction(in Place, in PN) =
224                 let NewPlace = undef, NewTransition = undef, Temp = undef in seq {
225                         update counter("InArcs") = 0;
226                         update counter("OutArcs") = 0;
227                         new('PetriNet'.'Place'(NewPlace) in PN);
228                         new('PetriNet'.'places'(Temp,PN,NewPlace));
```

```
229                     //sets all target ARCs to the newNode
230                     forall Tr, OutArc with find sourcePlace(Tr, Place, OutArc) do seq{
231                       setFrom(OutArc,NewPlace);
232                     }
233                     //sets all odd source ARCs to the new Place
234                     forall Tr, InArc with find targetPlace(Tr, Place, InArc) do seq{
235                            update counter("InArcs") = counter("InArcs")+1;
236                            if(toInteger(counter("InArcs")) % 2 == 0 ) seq{
237                                   setTo(InArc,NewPlace);
238                            }
239                     }
240                     //creates the additional transition
241                     new('PetriNet'.'Transition'(NewTransition) in NewPlace);
242                     new('PetriNet'.'transitions'(Temp, PN, NewTransition));
243                     new('PetriNet'.'Place'.'OutArc'(Temp,Place,NewTransition));
244                     new('PetriNet'.'Transition'.'InArc'(Temp,NewTransition,NewPlace));
245             }
246
247     rule inverseSerialTransitionRedaction(in Transition, in PN) =
248             let NewPlace = undef, NewTransition = undef, Temp = undef in seq{
249                     update counter("InArcs") = 0;
250                     new('PetriNet'.'Transition'(NewTransition) in PN);
251                     new('PetriNet'.'transitions'(Temp,PN,NewTransition));
252                     forall Place, InArc with find targetPlace(Transition, Place, InArc) do seq{
253                            update counter("InArcs") = counter("InArcs")+1;
254                            if( toInteger(counter("InArcs")) % 2 == 1 ) seq{
255                                   setFrom(InArc,NewTransition);
256                            }
257                     }
258                     //creates the additional transition
259                     new('PetriNet'.'Place'(NewPlace) in PN);
260                     new('PetriNet'.'places'(Temp, PN, NewPlace));
261                     new('PetriNet'.'Transition'.'InArc'(Temp,Transition,NewPlace));
262                     new('PetriNet'.'Place'.'OutArc'(Temp,NewPlace,NewTransition));
263             }
264
265     //Its input parameter is the root container of the newly generated Petri-net
266     rule main(in Where) =
267             let PN = undef, P1= undef, P2 = undef, T1 = undef, T2= undef,
268             Temp = undef , Tk1= undef, Tk2 = undef in seq { //creates the 2 place 2 transition
                    petri net
269                     new('PetriNet'(PN) in ref(Where));
270                     new('PetriNet'.'Place'(P1) in PN); rename(P1,"P1");
271                     new('PetriNet'.'Place'(P2) in PN); rename(P2,"P2");
272                     new('PetriNet'.'places'(Temp,PN,P1));
273                     new('PetriNet'.'places'(Temp,PN,P2));
274                     //transition
275                     new('PetriNet'.'Transition'(T1) in P1); rename(T1,"T1");
276                     new('PetriNet'.'transitions'(Temp, PN, T1));
277                     new('PetriNet'.'Transition'(T2) in P2); rename(T2,"T2");
278                     new('PetriNet'.'transitions'(Temp, PN, T2));
279                     //Arcs
280                     new('PetriNet'.'Place'.'OutArc'(Temp,P1,T1));
281                     new('PetriNet'.'Transition'.'InArc'(Temp,T1,P2));
282                     new('PetriNet'.'Place'.'OutArc'(Temp,P2,T2));
283                     new('PetriNet'.'Transition'.'InArc'(Temp,T2,P1));
284                     //tokens
285                     new('PetriNet'.'Place'.'Token'(Tk1) in P1);
286                     new('PetriNet'.'Place'.'tokens'(Temp,P1,Tk1));
287                     new('PetriNet'.'Place'.'Token'(Tk2) in P2);
288                     new('PetriNet'.'Place'.'tokens'(Temp,P2,Tk2));
289                     //initializes the counters
```

```
290                     update counter("ParPlace") = 0;
291                     update counter("ParTrans") = 0;
292                     update counter("SerPlace") = 0;
293                     update counter("SerTrans") = 0;
294                     update counter("LoopPlace") = 0;
295                     update counter("LoopTrans") = 0;
296                     update counter("Random") = 0;
297                     update counter("Token") = 0;
298                     update counter("Places") = 0;
299                     update counter("Transitions") = 0;
300
301                     //random generation
302                     //name of the PetriNet itself
303                     rename(PN,"Sparse_5000");
304                     iterate seq{
305                             update counter("Random") = counter("Random") + 1;
306                             if (counter("Random") > 5000) fail; //****The number of iteration
307                             random {
308                                     choose Place below PN, NextPlace below PN with find getPlace(
                                            Place, NextPlace, PN)
309                                       do call inverseParallelTransitionRedaction_rule(Place,PN,
                                            NextPlace);
310                                     choose P below PN with find petriPlace(PN,P) do call
                                            inverseSerialPlaceRedaction(P, PN);
311                                     choose NextTransition below PN, Transition below PN with find
                                            getTransition(Transition, NextTransition, PN)
312                                       do call inverseParallelPlaceRedaction_rule(Transition,PN,
                                            NextTransition);
313                                     choose T below PN with find petriTransition(PN,T) do call
                                            inverseSerialTransitionRedaction(T, PN);
314                                     choose P below PN with find petriPlace(PN,P) do call
                                            inverseSelfTransitionLoop_rule(P,PN);
315                             }
316                     }
317                     //Token
318                     iterate seq {
319                             update counter("Token") = counter("Token") + 1;
320                             if (counter("Token") > 8) fail; //The number of Tokens generated into
                                    the Petri-net
321                             choose P with find petriPlace(PN, P) do seq { call addToken_Rule(P);
                                    println("Token added to: "+ fqn(P)); }
322                     }
323
324                     //Counts the size of the generated Petri-net
325                     update counter("Tokens") = 0;
326                     forall Tok below PN with find token(Tok) do update counter("Tokens") =
                            counter("Tokens")+1;
327                     forall P below PN with find petriPlace(PN,P) do update counter("Places") =
                            counter("Places")+1;
328                     forall T below PN with find petriTransition(PN,T) do update counter("
                            Transitions") = counter("Transitions")+1;
329                     println("\nNumber of iterations: "+(counter("Random")-1));
330                     println("*** The generated Petri net contains "+ counter("Places")+" places
                            and "+ counter("Transitions")+" Transitions and " +counter("Tokens")+"
                            Tokens");
331             }
332 }
```

A.3 The Antworld Benchmark Program

```
1  import ants.metamodel;
2
3  @incremental('parallel'='1')
4  machine antMachine_sleek_prehybrid{
5
6          // cache
7          asmfunction model/0;
8          asmfunction antHill/0;
9
10         // statistics
11         asmfunction pheromones/0;
12         asmfunction foodCounter/0;
13         asmfunction foodTotal/0;
14         asmfunction circlesTotal/0;
15         asmfunction antsTotal/0;
16         asmfunction roundCounter/0;
17
18         //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
19         // GRID GROWING
20         //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
21         pattern boundary3(BoundaryField, BoundaryEdge, Hill) = {
22                 antHill(Hill);
23                 field(BoundaryField);
24                 antHill.boundary(BoundaryEdge, Hill, BoundaryField);
25         }
26
27         pattern boundaryBreachedBySearcher() = {
28                 field(Field);
29                 searcherAnt.location(HasAnt, Ant, Field);
30                 searcherAnt(Ant);
31                 find boundary3(Field, BoundaryEdge, Hill);
32         }
33
34         pattern alongReturnPath(OuterNeighbor, InnerNeighbor) = {
35                 field(InnerNeighbor);
36                 field(OuterNeighbor);
37                 field.returnPath(RP, OuterNeighbor, InnerNeighbor);
38         }
39         pattern circled(Field1, Field2) = {
40                 field(Field1);
41                 field(Field2);
42                 field.circlePath(CP, Field1, Field2);
43         }
44         pattern nextBoundaryField(BoundaryField, NextBoundaryField, NextBoundaryEdge) = {
45                 find circled(BoundaryField, NextBoundaryField);
46                 find boundary3(NextBoundaryField, NextBoundaryEdge, Hill);
47         }
48         pattern corner(CornerField) = {
49                 cornerField(CornerField);
50         }
51
52         rule expandBoundary(in BoundaryField, out Back, out Front, in OldBoundaryEdge, in Hill) =
53                 let BRP = undef, Model = model() in seq {
54                         new(field(Back) in Model);
55                         new(field.returnPath(BRP, Back, BoundaryField));
56                         setTo(OldBoundaryEdge, Back);
57                         call newField(Back, Hill);
58                         if (find corner(BoundaryField)) let BE1=undef, BE2=undef, CRP=undef, FRP=
                                undef, CP1=undef, CP2=undef, ExpandedCorner = undef in
```

```
59                      seq {
60                              new(cornerField(ExpandedCorner) in Model);
61                              new (antHill.boundary(BE1, Hill, ExpandedCorner));
62                              new(field.returnPath(CRP, ExpandedCorner, BoundaryField));
63                              new(field.circlePath(CP1, Back, ExpandedCorner));
64                              call newField(ExpandedCorner, Hill);
65
66                              new(field(Front) in Model);
67                              new(antHill.boundary(BE2, Hill, Front));
68                              new(field.returnPath(FRP, Front, BoundaryField));
69                              new(field.circlePath(CP2, ExpandedCorner, Front));
70                              call newField(Front, Hill);
71                      }
72                      else update Front = Back;
73
74              }
75      rule newField(in Field, in Hill) = seq {
76              if (foodCounter() < 9) update foodCounter() = foodCounter() + 1;
77              else let Food = undef, HF=undef in seq {
78                      update foodCounter() = 0;
79                      update foodTotal() = foodTotal() + 1;
80                      new (food(Food) in Field);
81                      new (field.hasFood(HF, Field, Food));
82                      setValue(Food, 100);
83              }
84      }
85
86      rule growGrid() = let Hill=antHill(), CP=undef, FirstExpanded = undef, PreviousExpanded =
            undef, PreviousBoundaryField = undef in
87              choose FirstBoundaryField, FirstBoundaryEdge with find boundary3(FirstBoundaryField,
                    FirstBoundaryEdge, Hill) do seq {
88                      update PreviousBoundaryField = FirstBoundaryField;
89                      call expandBoundary(FirstBoundaryField, FirstExpanded, PreviousExpanded,
                            FirstBoundaryEdge, Hill);
90                      iterate choose NextBoundaryField, NextBoundaryEdge with find
                            nextBoundaryField(PreviousBoundaryField, NextBoundaryField,
                            NextBoundaryEdge) do
91                      let BackExpanded = undef, FrontExpanded = undef in seq {
92                              update PreviousBoundaryField = NextBoundaryField;
93                              call expandBoundary(NextBoundaryField, BackExpanded, FrontExpanded,
                                    NextBoundaryEdge, Hill);
94                              new(field.circlePath(CP, PreviousExpanded, BackExpanded));
95                              update PreviousExpanded = FrontExpanded;
96                      }
97                      new(field.circlePath(CP, PreviousExpanded, FirstExpanded));
98                      update circlesTotal() = circlesTotal() + 1;
99              }
100
101
102
103     //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
104     // ANT ACTIONS
105     //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
106     pattern carrier(Ant) = {
107             carrierAnt(Ant);
108     }
109     pattern searcher(Ant) = {
110             searcherAnt(Ant);
111     }
112     pattern hasSearcherAnt(LocationEdge, Field, Ant) = {
113             searcherAnt(Ant);
114             searcherAnt.location(LocationEdge, Ant, Field);
```

```
115             field(Field);
116         }
117         pattern hasCarrierAnt(LocationEdge, Field, Ant) = {
118             carrierAnt(Ant);
119             carrierAnt.location(LocationEdge, Ant, Field);
120             field(Field);
121         }
122         pattern foodAvailable(Field, Food) = {
123             field(Field);
124             field.hasFood(HF, Field, Food);
125             food(Food);
126         }
127
128         pattern canGrab(Ant, LocationEdge, Food, Field) = {
129                     find searcher(Ant);
130                     find hasSearcherAnt(LocationEdge, Field, Ant);
131                     find foodAvailable(Field, Food);
132         }
133
134         rule grab(in Ant, in LocationEdge, in Food, in Field) =
135          let Rest = toInteger(value(Food)) -1 in seq{
136             if (Rest > 0) setValue(Food, Rest);
137                     else delete(Food);
138             delete(instanceOf(Ant,ants.metamodel.searcherAnt));
139             delete(instanceOf(LocationEdge,ants.metamodel.searcherAnt.location));
140             new(instanceOf(Ant,ants.metamodel.carrierAnt));
141             new(instanceOf(LocationEdge,ants.metamodel.carrierAnt.location));
142          }
143
144         rule deposit(in Hill, inout Ant, in LocationEdge) = seq{//raises the number of elements
145             setValue(Hill, toString( toInteger(value(Hill)) + 1));
146
147             delete(instanceOf(Ant,ants.metamodel.carrierAnt));
148             delete(instanceOf(LocationEdge,ants.metamodel.carrierAnt.location));
149             new(instanceOf(Ant,ants.metamodel.searcherAnt));
150             new(instanceOf(LocationEdge,ants.metamodel.searcherAnt.location));
151
152         }
153
154         rule moveAnt(in OldHasAnt, in NewField) = setTo(OldHasAnt, NewField);
155
156
157         pattern hasPheromone(Field, Pheromone) = {
158                     field(Field);
159                     field.hasPheromone(HF, Field, Pheromone);
160                     pheromone(Pheromone);
161         }
162         rule leavePheromone(in Field) =
163             try choose Pheromone with find hasPheromone(Field, Pheromone) do
164                     setValue(Pheromone, 1024 + toInteger(value(Pheromone)));
165             else let Pheromone = undef, HF = undef in seq {
166                     new (pheromone(Pheromone) in Field);
167                     new (field.hasPheromone(HF, Field, Pheromone));
168                     setValue (Pheromone, 1024);
169                     update pheromones() = pheromones()+1;
170             }
171
172
173         pattern attractingField(Field) = {
174             find hasPheromone(Field, Pheromone);
175             check(toInteger(value(Pheromone)) > 9);
176         }
```

```
177
178         @Random
179         pattern attractingOuterNeighbor(Field1, Field2) = {
180                         find alongReturnPath(Field2, Field1);
181                         find attractingField(Field2);
182                 }
183
184         pattern home(Field) = {
185                 antHill(Field);
186         }
187
188         @Random
189         pattern anyNeighborButHome(Field1, Field2) = {
190                         field(Field1);
191                         field(Field2);
192                         field.path(P, Field1, Field2);
193                         neg find home(Field2);
194                 } or {
195                         field(Field1);
196                         field(Field2);
197                         field.path(P, Field2, Field1); // reverse direction
198                         neg find home(Field2);
199                 }
200
201         rule search(in Ant) =
202                 choose Field1, HA1 with find hasSearcherAnt(HA1, Field1, Ant) do
203                 try choose /*random*/ Field2 with find attractingOuterNeighbor(Field1, Field2) do
                        call moveAnt(HA1, Field2); // WEIRD
204                 else choose /*random*/ Field2 with find anyNeighborButHome(Field1, Field2) do call
                        moveAnt(HA1, Field2); // WEIRD
205
206         //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
207         // WORLD MANAGEMENT
208         //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
209         pattern pheromone(P) = {
210                 pheromone(P);
211         }
212         rule evaporate(in Pheromone) =
213                 let Rest = (19*toInteger(value(Pheromone)))/20 in
214                 if (Rest > 0) setValue(Pheromone, Rest); else seq {
215                         delete(Pheromone);
216                         update pheromones() = pheromones() - 1;
217                 }
218
219         pattern delivered(Hill) = {
220                         antHill(Hill);
221                         check(toInteger(value(Hill)) > 0);
222         }
223
224         rule consume(in Hill) =
225          let Ant = undef, HA = undef in seq{
226                 setValue(Hill,toString(toInteger(value(Hill))-1));
227                 new(searcherAnt(Ant) in Hill);
228                 new(searcherAnt.location(HA, Ant, Hill));
229                 update antsTotal() = antsTotal() + 1;
230          }
231
232
233         //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
234         // MAIN
235         //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
236         rule doRound() = let Hill = antHill() in seq {
```

```
237             //Ant actions
238             iterate choose Ant, LocationEdge, Food, Field with find canGrab(Ant, LocationEdge,
                    Food, Field) do call grab(Ant,LocationEdge,Food,Field);
239             forall Ant, LocationEdge with find hasCarrierAnt(LocationEdge, Hill, Ant) do call
                    deposit(Hill,Ant,LocationEdge);
240             forall Ant, FromField, HA1 with find hasCarrierAnt(HA1, FromField, Ant) do
241                     choose NewField with find alongReturnPath(FromField, NewField) do seq {
242                             call moveAnt(HA1, NewField);
243                             call leavePheromone(FromField);
244                     }
245             forall Ant with find searcher(Ant) do call search(Ant); // two kinds of search
246             // only searchers can breach the boundary!
247
248             //Field action
249             forall Pheromone with find pheromone(Pheromone) do call evaporate(Pheromone);
250
251             iterate
252                     if(toInteger(value(Hill)) > 0)
253                             call consume(Hill);
254                     else
255                             fail;
256             if (find boundaryBreachedBySearcher()) call growGrid();
257
258     }
259
260     rule printStatistics(in Buf, in MemTelemetry, in RoundCounter, in Rounds, in BlockSize, in
            AntAccumulator, in StartTime, inout LastTime) =
261       let CurrentTime = systime() in seq
262       {
263             println(Buf, "\t<round-block finished=\"" + RoundCounter + "\" goal=\"" +
                        Rounds + "\" block-size=\"" + BlockSize+ "\">");
264             println(Buf, "\t\t<elapsed-time>");
265             println(Buf, "\t\t\t<per-block> " + (CurrentTime-LastTime) + " </per-block>")
                        ;
266             println(Buf, "\t\t\t<per-round> " + (CurrentTime-LastTime)/BlockSize + " </
                        per-round>");
267             println(Buf, "\t\t\t<per-round-per-1000ants> " + 1000*(CurrentTime-LastTime)
                        / AntAccumulator + " </per-round-per-1000ants>");
268             println(Buf, "\t\t\t<total> " + (CurrentTime-StartTime) + " </total>");
269             println(Buf, "\t\t</elapsed-time>");
270             println(Buf, "\t\t<circles> " + circlesTotal() + " </circles>");
271             println(Buf, "\t\t<grid-fields> " + circlesTotal() * circlesTotal() * 4 + "
                        </grid-fields><!-- excluding the anthill -->");
272             println(Buf, "\t\t<food-bundles-created> " + foodTotal() + " </food-bundles-
                        created>");
273             println(Buf, "\t\t<pheromone-traces> " + pheromones() + " </pheromone-traces>
                        ");
274             println(Buf, "\t\t<ants> " + antsTotal() + " </ants>");
275             if (MemTelemetry == 1)
276                     println(Buf, "\t\t<memory> " + measureMemoryFootprint(6) + " </memory>
                            ");
277             else
278                     println(Buf, "\t\t<memory> NA </memory>");
279             println(Buf, "\t</round-block>");
280             update LastTime = CurrentTime;
281       }
282
283     rule main(in Rounds, in Variant, in MemTelemetry) = let StartTime = systime(), BlockSize =
            25,
284     Buf = getBuffer("file://output/"+Variant+".out.xml")
285     in seq {
286             update model() = ref("ants.model");
```

```
287             update antHill() = ref("ants.model.hill");
288
289             update foodCounter() =      toInteger(value(ref("ants.statistics.foodCounter")));
290             update foodTotal() =        toInteger(value(ref("ants.statistics.foodTotal")));
291             update circlesTotal() =     toInteger(value(ref("ants.statistics.circlesTotal")));
292             update antsTotal() =        toInteger(value(ref("ants.statistics.antsTotal")));
293             update pheromones() =       toInteger(value(ref("ants.statistics.pheromones")));
294             update roundCounter() =     toInteger(value(ref("ants.statistics.roundCounter")));
295             println(Buf, "<anthill-simulation rounds=\"" + Rounds + "\" up-to=\"" + (Rounds +
                    roundCounter()) + "\">");
296             let BlockCounter = 0, AntAccumulator = 0, RoundMax = Rounds + roundCounter(),
                    LastTime=StartTime in iterate seq {
297                 if (roundCounter() >= RoundMax) fail;
298                 update roundCounter() = roundCounter() + 1;
299                 call doRound();
300                 update BlockCounter = BlockCounter + 1;
301                 update AntAccumulator = AntAccumulator + antsTotal();
302                 if (BlockCounter >= BlockSize) seq {
303                     call printStatistics(Buf, MemTelemetry, roundCounter(), RoundMax,
                            BlockSize, AntAccumulator, StartTime, LastTime);
304                     update BlockCounter = 0;
305                     update AntAccumulator = 0;
306                 }
307             }
308             println(Buf, "\t<final-statistics>");
309             println(Buf, "\t\t<total-elapsed-time> " + (systime()-StartTime) + " </total-elapsed
                    -time>");
310             println(Buf, "\t\t<circles> " + circlesTotal() + " </circles>");
311             println(Buf, "\t\t<grid-fields> " + circlesTotal() * circlesTotal() * 4 + " </grid-
                    fields><!-- excluding the anthill -->");
312             println(Buf, "\t\t<food-bundles-created> " + foodTotal() + " </food-bundles-created>
                    ");
313             println(Buf, "\t\t<pheromone-traces> " + pheromones() + " </pheromone-traces>");
314             println(Buf, "\t\t<ants> " + antsTotal() + " </ants>");
315             println(Buf, "\t</final-statistics>");
316             println(Buf, "</anthill-simulation>");
317             setValue(ref("ants.statistics.foodCounter"), foodCounter());
318             setValue(ref("ants.statistics.foodTotal"), foodTotal());
319             setValue(ref("ants.statistics.circlesTotal"), circlesTotal());
320             setValue(ref("ants.statistics.antsTotal"), antsTotal());
321             setValue(ref("ants.statistics.pheromones"), pheromones());
322             setValue(ref("ants.statistics.roundCounter"), roundCounter());
323         }
324 }
```

Bibliography

[1] The Eclipse Modeling Framework project. `http://www.eclipse.org/modeling/emf/`.

[2] The Eclipse project. `http://www.eclipse.org`.

[3] The Erlang programming language. `http://www.erlang.org/`.

[4] Grabats - graph-based tools: The contest. Official Website (2008): `http://www.fots.ua.ac.be/events/`.

[5] The Haskell programming language. `http://www.haskell.org/`.

[6] Viatra transformation language specification. `http://www.eclipse.org/gmt/VIATRA2/doc/ViatraSpecification.pdf`.

[7] *VIATRA2 Framework*. An Eclipse GMT Subproject (`http://www.eclipse.org/gmt/`).

[8] Apt, K. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[9] Bergmann, G., Horváth, Á., Ráth, I., and Varró, D. A benchmark evaluation of incremental pattern matching in graph transformation. In *In Proc. of ICGT '08 , 4th Intl. Conference on Graph Transformation* (2008), R. Heckel and G. Taentzer, Eds., LNCS 5214, Springer.

[10] Binder, R. V. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] Börger, E., and Stärk, R. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[12] Bray, T. Extensible Markup Language (XML) 1.0 (fourth edition), 2006. Available from `http://www.w3.org/TR/xml/`.

[13] Carlsson, M., Widén, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., and Sjöland, T. *SICStus Prolog User's Manual*, release 4.0.4 ed. Swedish Institute of Computer Science, 2008.

[14] Caseau, Y. Efficient handling of multiple inheritance hierarchies. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 1993), ACM, pp. 271–287.

[15] Clark, J. Xsl Transformations (XSLT), 1999. Available from `http://www.w3.org/TR/xslt`.

[16] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model checking*. MIT, Cambridge, Mass. [u.a.], 2001.

[17] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.

[18] Dechter, R., and Pearl, J. Network-based heuristics for constraint-satisfaction problems. *Artif. Intell. 34*, 1 (1987), 1–38.

[19] Dorigo, M., Maniezzo, V., and Colorni, A. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on 26*, 1 (1996), 29–41.

[20] Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., Eds. *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.

[21] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[22] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[23] Horváth, A. Automated generation of platform specific model transformation. Master's thesis, Budapest University of Technology and Economics, 2006.

[24] Hovemeyer, D., and Pugh, W. Finding bugs is easy. *SIGPLAN Not. 39*, 12 (2004), 92–106.

[25] ILOG S.A. *ILOG Solver 5.0: Reference Manual*. Gentilly, France, 2000.

[26] Jüngel, M., Kindler, E., and Weber, M. The petri net markup language. In *7. Workshop Algorithmen und Werkzeuge fr Petrinetze, pages 4752, Universitt Koblenz-Landau* (2000), p. http://www.informati.

[27] Jussien, N. e-constraints: explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction* (Paphos, Cyprus, 1 Dec. 2001).

[28] Kamfonas, M. J. Recursive hierarchies: The relational taboo! *The Relational Journal* (October/November 1992).

[29] Lin, Y., Zhang, J., and Gray, J. Model comparison: A key challenge for transformation testing and version control in model driven software development. In *Workshop on Best Practices for Model-Driven Software Development, held at OOPSLA '04* (2004).

[30] Lindahl, T., and Sagonas, K. Practical type inference based on success typings. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2006), ACM.

[31] Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (Apr. 1989), 541–580.

[32] Nickel, U., Niere, J., and Zündorf, A. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)* (Limerick, Ireland, 2000), ACM Press.

[33] Object Management Group. *Model Driven Architecture — A Technical Perspective*, September 2001. `http://www.omg.org`.

[34] Object Management Group. *Meta Object Facility Version 2.0*, 2003. `http://www.omg.org`.

[35] Pointon, R., Trinder, P., and Loidl, H.-W. The design and implementation of glasgow distributed Haskell. In *IFL'00, Implementation of Functional Languages* (September 2000).

[36] Pottier, F. A modern eye on ml type inference, 2005. In Proc. of the International Summer School On Applied Semantics (APPSEM '05).

[37] Rensink, A. Representing first-order logic using graphs. In *Proc. 2nd International Conference on Graph Transformation (ICGT 2004), Rome, Italy* (2004), H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, Eds., vol. 3256 of *LNCS*, Springer, pp. 319–335.

[38] Rensink, A., Schmidt, Á., and Varró, D. Model checking graph transformations: A comparison of two approaches. In *Proc. ICGT 2004: Second International Conference on Graph Transformation* (Rome, Italy, 2004), vol. 3256 of *LNCS*, Springer, pp. 226–241.

[39] Robin, M. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (December 1978), 348–375.

[40] Rozenberg, G., Ed. *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.

[41] Rutar, N., Almazan, C. B., and Foster, J. S. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering* (2004), IEEE Computer Society, pp. 245–256.

[42] Tozawa, A. Towards static type checking for XSLT. In *In DocEng '01: Proceedings of the 2001 ACM Symposium on Document engineering* (2001), ACM, pp. 18–27.

[43] Varró, D., and Pataricza, A. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling 2*, 3 (October 2003), 187–210.

[44] Zündorf, A. Antworld benchmark specification, GraBaTs 2008, 2008. Available from `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2008performancecase.pdf`.