Static Type Checking of Model Transformations by Constraint Satisfaction Programming

Zoltán Ujhelyi
uz602@hszk.bme.hu

Ákos Horváth
ahorvath@mit.bme.hu

Dániel Varró
varro@mit.bme.hu

June 26, 2009

Abstract

The control structure of the VIATRA2 transformation programs is untyped making it easier to misuse the type system. The aim of this paper is to provide a static type checker tool to these transformation programs that can detect those errors.

We use a generic static analysis framework for analysis, and as an underlying engine constraint satisfaction problem solver is used. For this reason it is required to translate the metamodel (that acts as the type system) and the type safety properties of the transformation program to constraints.

1 Introduction

As more and more complex model transformations are used during development early error detection is becoming a key question as errors of the transformations can even propagate into the developed application. The fact that the specification of transformations is similar to a high level computer programs means verification methods researched for computer programs are also applicable for model transformation programs.

It is a relatively common error (especially in dynamically typed languages) that the developer uses the type system incorrectly. They most often lead to a misleading output rather than a runtime exception making them hard to trace. On the other hand using static type analyser tools could help the developer to detect such problems.

In [25] a generic static analysis framework for model transformation programs was introduced. In this paper we describe a static type checker system for the VIATRA2 framework based on that generic method.

The framework needs a graph model of the transformation program, and creates constraints from it. To describe type checking we need to populate the graph model and define the constraint mapping process.

1.1 Main Goals

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 main goal of the framework are the following:

Type Inference The types of variables have to be determined regardless if they are available in the transformation program or has to be inferred;

Type Safety Checking The incorrect use of the type system should be detected, e.g. the use of a model element instead of a boolean variable.

Resource Usage execution time and memory consumption of the analysis is an important concern during the implementation, as it shall complete within a reasonable amount of time on an average developer computer. This helps to identify the potential problems as early as possible.

1.2 The Structure of the Report

The report is structured as the following: Section 2 details the important concepts used, Section 3 describes the ideas of the generic static analyser solution. Then Section 4 details the main type checking ideas while Section 5 shows how to handle the language elements. The error detection is detailed in Section 6. Finally some related work is introduced in Section 7, and Section 8 concludes our work and details future enhancements.

2 Background Concepts

2.1 Models and Transformations in Viatra2

A possible way to describe complex transformations is the use of *graph transformation* (GT) [12] rules for local model manipulations and *abstract state machine* (ASM) [5] 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) [3] 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.

The framework uses the *Viatra Textual Command Language* (VTCL) for defining transformations. 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 [2].

2.1.1 Metamodeling

Metamodeling provides a structural definition (ie. abstract syntax) of modeling languages. Such a definition is needed to define the input and output of model transformations.

The VIATRA2 framework uses the VPM (Visual and Precise Metamodeling) [26] concept to describe metamodels. The basic elements of the metamodels are *Entities* that represent the basic concept of the (modeling) domain, and *Relations* that represent general relationships between the Entities. Both the Entities and the Relations are arranged into a strict containment hierarchy described by the `containment` relation.

There are other built-in relations: 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 (thus allowing multilevel metamodeling in the same model space), while the `supertypeOf` relation represents a binary superclass-subclass relation (similar to the concept of generalization in UML).

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

Petri nets 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.

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.*

The Petri net metamodel is depicted in Figure 1.

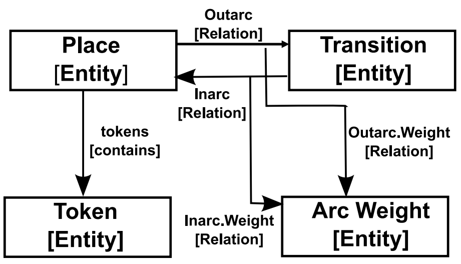


Figure 1: The Graphical Representation of the Petri net Metamodel

2.1.2 Graph Patterns

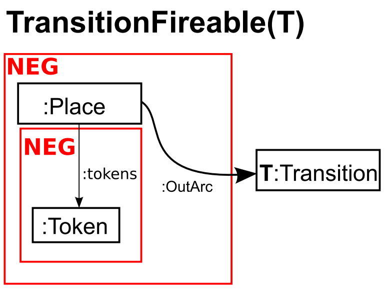
Graph patterns can be considered as the atomic units of graph transformations. They represent a condition (or constraint) on (a part of) the model space. Graph patterns are used in transformation rules as conditions and as a description of the result pattern.

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 *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 reaching the expressive power of First Order Logic [22].

Example 2: *As an example of graph patterns we describe the pattern of the fireable transitions over the metamodel defined before. A graphical and the*

VTCL representation of the pattern can be seen in Figure 2.



(a) Graphic Description

```
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);
    }
  }
}
```

(b) VTCL Description

Figure 2: The Transition Fireable Graph Pattern

The pattern represents, that a Transition is fireable if it is not connected *to a Place by an* Outarc, *where the Place* contains no *tokens.*

Patterns are described using a typed graph, where the nodes are elements of the metamodel, while the relations are depicted as arcs. E.g., the node ***T:Transition*** *(or the line* ***'PetriNet'.'Transition'(T);****) means that there is a model element (called T) with the type of* ***Transition****. Similarly the types of every element in the pattern graph can be determined.*

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).

To increase the expressiveness of the language two constructs are used together with patterns. (i) It is possible to *call* other patterns (with the `find` keyword). The match condition of the caller pattern is fulfilled if all the called subpatterns are fulfilled. (ii) The other construct is the *alternate patterns*: a pattern can have multiple bodies. When several alternative patterns are defined, the pattern is fulfilled if any of the bodies can be fulfilled.

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 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 3: *Figure 3 shows the graphical representation a transformation rule named `addToken` related to firing the transition.*

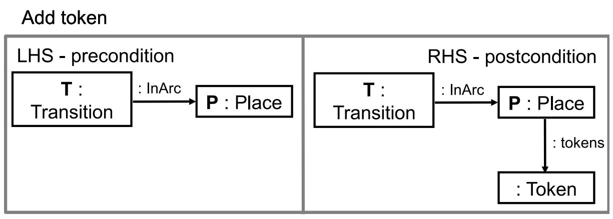


Figure 3: Graphical Representation of Graph Transformation Rules

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`.

As in common transformation rules the LHS and RHS graphs are nearly identical they can be described as a single graph with `new` and `delete` annotations (as in FUJABA [19] notation). The LHS and RHS graphs can be created from this notation: elements tagged with the `new` keyword, are only elements of the RHS graph, elements tagged with the `delete` keyword, are only part of the LHS side and the other elements appear in both graphs.

The VTCL language uses three elements to describe GT Rules: a `precondition` and at least one of a `postcondition` pattern or `action` ASM Rule. When using only the postcondition pattern, the two graphs are used to describe the rule; the action can be used to express the annotations, while the typical usage of both elements are debugging or code generation.

Example 4: *To illustrate the the GT Rules description, Listing 1 describes the `addToken` rule in both notations. The called `place` and `placeWithToken` patterns are not detailed here.*

Listing 1 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)
}

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

In the `addToken` GT Rule the `precondition` and `postcondition` graph patterns are described by a call of the `place` and `placeWithToken` patterns respectively.

The `addTokenFUJABA` rule also has a precondition pattern, but instead of a postcondition graph pattern it uses an `action` ASM rule, that creates a new Token entity and a tokens relation.

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 (ASM) [5] rules to describe the control structure.

The basic elements of the ASM programs are: *ASM Rules* represent a set of operations (like methods in OO languages); *ASM Variables* hold values (model element references, constants, etc.); while *ASM Functions* store variables in arrays (similar to associative arrays in other languages).

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).

Three ASM Rules can be 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* and the *sequential* rules are used to creating compound rules. The *random* rule executes one of its nested rules, 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: *Listing 2 contains the `fireTransition` rule that matches all source places of the input `Transition` parameter with the first `forall` rule and deletes a Token from each place (calling the GT Rule `removeToken`, while the second `forall` rule generates a `Token` for all target places by calling the GT Rule `addToken`.*

Listing 2 A Simple ASM Rule Describing the Firing of Transitions

```
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
      call removeToken(Place,Token);
  //adds the new tokens to the output places
  forall Place with find targetPlace(Transition, Place) do
    call addToken(Place);
}
```

It is important to note that the ASM Rules are untyped: there is no type information is explicitly described, only some basic type checking happens in the interpreter during execution.

2.2 Static Analysis and Type Inference

It is a known fact in computer programming that the repair cost of an error increase by an order of magnitude if it remains undetected during a design phase. *Static analysis* tools help to discover errors early - before running the application - but only for a predefined set of errors. For this fixed set static analysis can prove that none of them are present in the program.

Compilers of statically bound languages, like Java or C# include some kind of static analysis: during compilation they determine the types of variables, watch for uncaught exceptions, etc.

A static analysis may be carried out by an *abstract interpretation* [10] 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.

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.

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 metamodel element can be added). On the other hand the type analysis needs a fixed set of possible types, so before the 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 general any abstract analysis tool can become the basis of the static analysis, 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

Constraint Satisfaction Problems(CSP) [4] are mathematical problems defined as a set of objects (*variables* with a fixed *domain*) whose state (*value*) must fulfill a *set of constraints*. The CSPs are categorized by the domain of their variables: e.g. finite domain variables or real numbers can also be used. In this report we are only using finite domain CSPs.

The CSP can be represented as a (hyper)graph [11], where the nodes are the variables, and the arcs (and hyperarcs) represent the constraints.

Example 6: *A hypergraph visualization of a CSP problem can be seen in Figure 4.*

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. Some ways to find solutions:

Propagation is used to modify the problem to make it easier to solver. By some reasoning about the constraints it is often possible to reduce the domain of one of more constraint variables without excluding any

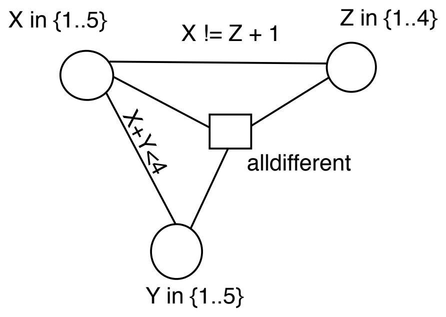


Figure 4: A Simple Graph Visualization of a Constraint Satisfaction Problem

possible solution. The biggest drawback is that there are cases when the satisfiability cannot be decided by propagation.

Search algorithms (typically backtracking or backjumping) can be used to find a solution: as the domains of the constraint variables are finite, in theory it is possible to systematically try out all possible combinations. However in practice the number of combinations is exponential in the size of variables, so this is not a feasible solution.

Combined Propagation and Search algorithms appear to be effective in practice: propagation is used to reduce the domains as it is possible, otherwise search is used to find a solution.

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 may remove symmetries, or help the solver to choose a constraint which reduces the domains of the variables more effectively.

Example 7: *To illustrate constraint propagation lets consider the problem defined in Figure 4. It is possible to reduce the domains of the variable X and Y to $[1;2]$, as otherwise the $X + Y \leq 4$ constraint could not be fulfilled.*

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.

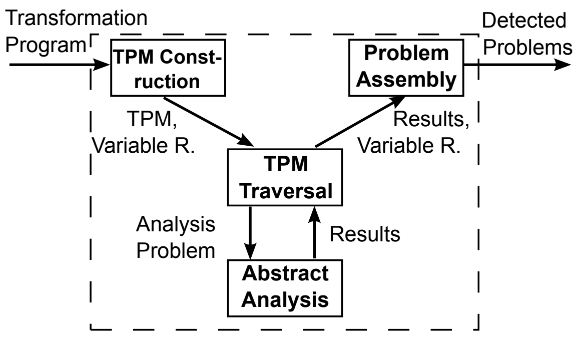


Figure 5: The TPM Based Static Analysis Process

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++) [15], the clp modules of SICStus Prolog [7] or the Gecode library (for C++ and Java) [14], etc.

3 Static Analysis of Transformation Programs

The static analysis solution described in [25] 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 programs different run paths: it omits information e.g. the current values of the variables. The fact that the model space that the transformation is executed on is not tracked allows the analyser to look for problems 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 5. These tasks are the following:

Constructing the TPM The TPM graph represents the entire transformation program. The program variables are converted to single assignment variables in the corresponding abstract domain (the domain depends on the analysis criteria) and are stored in a variable repository component (although the values are only put there during traversal).

As the TPM graph represents all the potentially infinite run paths, a *k depth limit* is introduced for recursive calls: if there are at least *k* occurrences of the same called element, the called node is replaced by a sentinel node representing no available information (and no children).

Traversing the TPM The traversal of the TPM graph is following the *visitor design pattern* [13], thus the traversal algorithm is independent of the constraint generation process. The different run paths of the transformation program are represented by *branches* in the TPM graph; the traversal algorithm solves this by traversing every branch separately, and initializes a new CSP for every branch.

Generating and solving a CSP For every node of the TPM model a set of constraints could be generated, these constraints are used to build a constraint satisfaction problem. The satisfiability of this CSP is checked after every constraint is generated thus allowing early error detection.

To be able to integrate different CSP solvers into the analyser framework, according to the *bridge design pattern* [13] an interface is created that handles a predefined set of constraints. The list of constraints have to be evaluated for every analysis criteria. Using this interface allows the inspection of various solver engines to select the one with the best runtime characteristics.

To measure the performance of the different solver engines the average time needed to solve the problem of a branch and the overall memory consumption is measured. Because the analysis complexity depends on both the transformation program and the metamodel, several test programs on different metamodels should be evaluated.

Listing the detected problems After the traversal finishes (either successfully or unsuccessfully) the result should be processed to help the user to interpret the errors. For this reason bug pattern detectors are used: when a pattern is matched in the results, then a bug is caught. There are three categories of patterns used (based on the source of the bug):

Analysis Problems represent that the CSP is unsatisfiable.

Inconsistencies represent that the CSP is solvable but the values of the different TPM variables describe a bug pattern.

Traversal Problems describe unexpected events during the traversal (not related to the CSP).

4 Type Checking of the VTCL Language

The general static analysis framework can be based on any kind of abstract analysis. For static type checking we have chosen a CSP solver engine as the type safety of a program can be easily described as a constraint satisfaction problem.

4.1 Integrating the Analyser

The VIATRA2 framework is a set of Eclipse [1] plugins. This plugin-based architecture enables extending the framework in a well defined way.

Figure 6 shows the main components that the static type checker is connected.

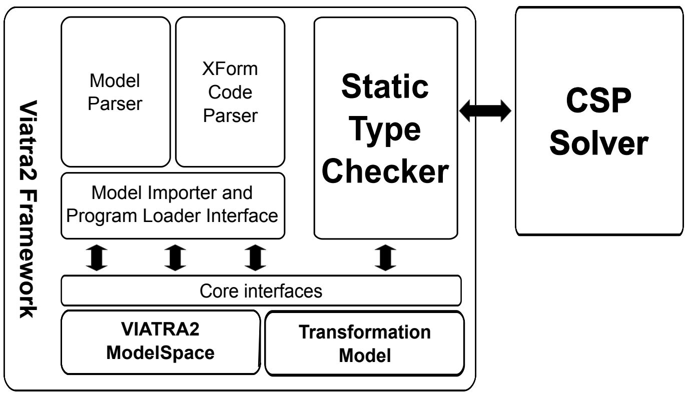


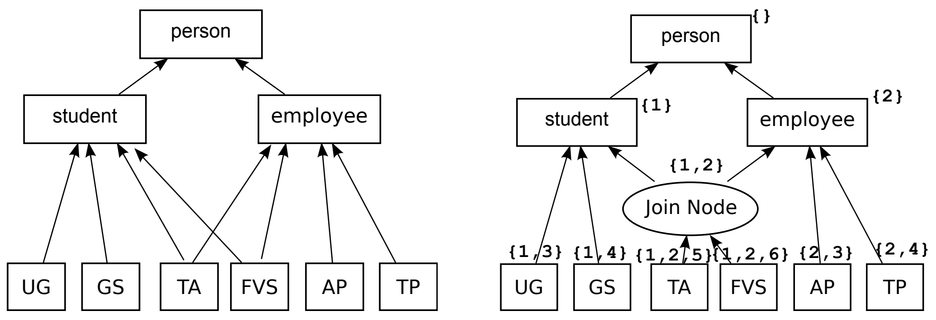
Figure 6: The Static Type Checker in the VIATRA2 Framework

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 transformation program, instead it relies on the framework parsers. The analyser only communicates through the core interfaces with the model space and the program model store.

4.2 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 type hierarchies with the inheritance as the connecting relationship, and then we discuss how to use this representation in constraints to be evaluated by the CSP solver.



(a) The Hierarchy (b) The Gene Assignment

Figure 7: The Gene Assignment for the University Member Hierarchy

4.2.1 Representing Type 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 [17], 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 its 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 [8], 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 8: *As the Petri net metamodel does not contain inheritence first another gene assignment is used to describe these capabilities. In Figure 7 (the example is taken from [8]).*

The hierarchy describes members of a university. Every member is a *`person`, and they can be `students` or `employees`. **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 two 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.*

The fact that the algorithm tries to use as few genes as possible helps us to use describe the metamodel in a compact way thus reducing the memory usage of our algorithm.

To represent the metamodels with these sets first three special nodes are created: 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 (e.g. related to the ASM Term function `target` uses the `TopLevelRelation` node, see Section 5.1).

Example 9: *To represent the metamodel hierarchy of the Petri nets defined in Section 2.1.1 with this algorithm (together with the special nodes), the algorithm assigned genes between the numbers of* 1 *and* 14*. The resulting gene sets are present in Figure 8.*

Although the metamodel of the Petri nets does not contain any inheritance relations at all, some are present in this representation: everything is the descendant of the `TopLevelNode`, every entity is descendant of `TopLevelEntity` and every relation is descendant of `TopLevelRelation`.

4.2.2 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 its 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.

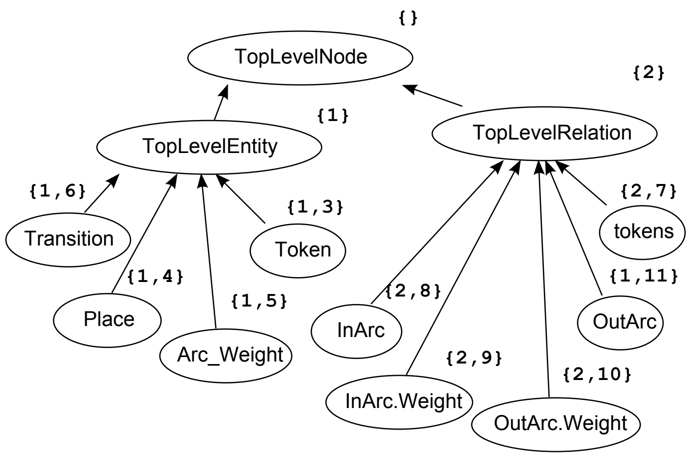


Figure 8: The Gene Assignment for the Petri net Metamodel

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, two relations were defined, 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 relation can be used to precisely describe the type of a TPM variable. The type equality relation holds between two elements if and only if 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 states that the type of an element is either the same or the descendant of the other elements type. The relation is used to represent the type constraint of a variable assignment. The substitutability relation holds between two elements if and only if the assignees gene set is the subset of the assigned ones.

The substitutability relation is more general than the type equality, but for those types, which do not have any descendants (such as the built-in types), the two relations are the same. 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 (both by reducing the number of sets created and by the fact that the smaller set could be stored more efficiently).

As the relation properties are independent from the hierarchy they have to be described another way. As the relation type determines the type of its parameters these properties can be described by a set of conditional constraints. The constraints are all of the following scheme:

$$(Relation\ is\ \text{``}relationtype'') \Rightarrow (Parameter\ is\ \text{``}parametertype'')$$

In these constraints *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 10: *Over the domain of Petri nets (the metamodel is depicted in Figure 1) to express that the variable R is a relation, and the variable F is the "from" parameter of R, the following constraints are needed:*

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

4.2.3 The Constraint Handler API for the Traversal

In order to describe the type safety as a constraint satisfaction problem, the following kind of constraints are used (the interface of the constraint solver should be able to handle these constraints):

Type Equals Constraint represents the type equality of two elements. This constraint represents a substitutable relation (as defined in Section 4.2.2). 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. 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 they have to be translated (often several) to constraints over CSP variables. As CSP variables represent the type of the TPM variables it is quite straightforward to create this translation for any concrete CSP solver engine.

4.2.4 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 our needs better.

During the development the Gecode/J library [14] 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 optimalization to overcome this aspect.

The Gecode library had a set representation, but it did not work well with our specialized sets; on the other hand the SICStus module did not support neither sets nor incremental problem building. These problems caused that neither solver were capable of handling any but the smallest transformation programs.

These problems led to create our own solver implementation. The algorithm creates a graph model from the constraint satisfaction problem and uses this model for propagation: if the domain of a variable changes the related constraints are notified. The notified constraint checks whether these changes can be used to reduce the domains of the other variables described by the same constraint.

To have a both effective and memory-efficient solution sets are described as integers, and bitwise operations are used to express set operations. These sets are used to describe the domain of integer variables; to optimize memory consumption instead of supersets an approximate storage solution is used. The effects of these bitwise operations is that our implementation is working more efficiently in smaller problems (up to about several thousand variables) while the SICStus implementation is optimized for much larger problems, where our implementation will not scale well.

By measuring the execution time and memory consumption we examined medium-sized transformation programs, and found that the simple solver implementation could handle the analysis well, as the average execution time of a branch were less than 60 ms, and the memory usage was about 15 MB RAM. These results are acceptable for the analysis when the number of branches is not too high (up to about 1000 branches).

5 Type Checking of the TPM Nodes

Using the TPM traversal based static analysis framework allows that the relevant parameters of the language elements can be described separately to the traversal. For every language element (represented by a TPM Node) a list of constraints have to be produced together with a list of child nodes and branching information.

This data is enough to run the static analysis.

5.1 Traversing ASM Term Nodes

ASM Term nodes are closely related to variables so 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).

5.1.1 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 4.2, and the type information has to be filled (if available) using the type equality relation.

5.1.2 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 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 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.

5.1.3 Conversion Operators

Conversion operators are used to transform its operand to another type. The conversion operators are available only for the built-in types, not the `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 2.

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`	{any possible type}	{Multiplicity}

Table 2: The Type Constraints of the Conversion Operators

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

5.1.4 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 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}
Not Equals	{any possible type}	{Boolean}
More or Equals	{String, Integer, Double}	{Boolean}
More	{String, Integer, Double}	{Boolean}
Logical Operations	{Boolean}	{Boolean}

Table 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 its 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.

5.1.5 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 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).

Relation parameter constraint sets (as described in Section 4.2.2) are used at the `inverse`, `source` and `target` queries to increase precision.

5.1.6 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.

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: The Type Constraints of the Model Element Query Operators

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.

5.2 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.

5.2.1 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, a *depth limit* 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.

5.2.2 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.

5.2.3 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.

5.2.4 Nested Rules

The Nested Rules (`sequential` 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 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 5 contains a short summary of the handling of the nested rules.

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

Table 5: The Analysis of Nested Rules

5.2.5 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.

5.2.6 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 constraints: (1) the created model element (and thus the variable) will have the type given as parameter, (2) the type parameter have to be a *model element type* (either entity or relation); (3) a variable with the type of the type parameter is used to store the new element; (4) if filling in a relation, relation parameter constraint sets can also be generated.

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. To handle this, new TPM variables have to be created (and thus CSP variables) that are connected via constraints. There all several update rules, all with two ASM Terms as parameters. Without further explanation the type parameters are the following:

- `rename(Model Element, String)`
- `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 6 contains a short summary of the handling of model manipulation rules.

5.2.7 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 5.1, while in case of GT Rules Section 5.3.2 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

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 6: The Constraint Mapping of Model Manipulation Rules

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 non-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 11: *To illustrate the handling of the* `choose` *rule, let's consider the rule presented in Listing 3:*

Listing 3 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 5.3.2. 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.*

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 7: The Analysis of Iteration Rules

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 problem types.

Table 7 contains the handling of the collection iterator nodes.

5.3 Traversing GT Rule and Pattern Nodes

5.3.1 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, a *depth limit* 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 5.2.1 or GT 5.3.3 Rule calls).

5.3.2 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.

Example 12: *The called pattern of Listing 3 is described in Listing 4.*

Listing 4 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.*

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 8: The Elements of GT Patterns

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 8 summarizes the handling of the elements of the GT Patterns.

5.3.3 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 a *depth limit* is 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.

5.3.4 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.

Listing 5 The addToken GT Rule

```
gtrule addToken(in Place) =
{
  precondition find place(Place)
  postcondition find placeWithToken(Place, Token)
  action{
    print(Place);
    print(Token);
  }
}
```

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

Table 9: The Analysis of GT Rules

Example 13: *Listing 5 displays the* `addToken` *GT Rule introduced in Section 5.3.4 with a slight alteration: both the postcondition and action part has been defined in order to demonstrate the traversal.*

When investigating this GT Rule node, first the precondition, then the postcondition pattern is traversed as described in Section 5.3.2, then the rule is also executed as described in Section 5.2.

Table 9 displays the parameters of the GT Rule nodes.

6 The Detected Type Handling Problems

After the TPM traversal finishes the results have to be searched for bug patterns. The analysis framework defines three category of bugs to check for (see Section 3). Currently the type checker looks for three patterns (one of each category):

1. The *analysis problems* detected by the CSP solver are reporting at least one of the CSP variables with 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 its lifecycle.
3. A common *traversal problem* 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.

By using this analyser it is easier to spot type safety errors as if some happens (e.g. accidentally calling another graph pattern than intended) errors (or warnings) are issued. Sometimes a problem is not detected where it happened, but if the transformation developer knows some context information it is easier to track down the cause.

7 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.

The transformation of XML [6] documents (via the XSLT [9] transformation language) involves similar concepts to type checking. XML documents can be interpreted as a hierarchical data structure, their type specification is written in a DTD (Document Type Definition) (or XSD (XML Schema)). By calculating the DTD of the input document from the transformation and the output DTD it is possible to decide whether the input document (written by the user) would be transformated to the output DTD without running the transformation. This problem can be treated as *backward type inference* [24]. 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.

The well known Hindley-Milner [23] algorithm for lambda calculus reduces the typing problem to a unification problem of equations. The algorithm is used in the functional languages Haskell [20] and Erlang [18]. Our approach was influenced by the work started in [21], which translates the typing problem to a set of constraints. As lambda calculus does 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 multilevel metamodeling.

8 Conclusion and Future Work

In this paper we introduced a static type checker tool for the Viatra2 VTCL language. The created tool is capable of detecting some cases of incorrect use of the type system (including pattern or rule calls with invalid parameters).

The tool is capable of inferring the type of most variables in transformation programs, but there are some cases where it is not possible to calculate it. Such are variables defined with an undefined value (`undef`), as it can become any type (although if the variable changes the type might become detectable). It is also possible that variables are only used in constructs that does work on multiple types - in such cases the precise type cannot be calculated only approximated.

Such result happens when a parameters only use is a comparison with an integer variable: in this case the type of the parameter is one of string, integer or double.

The main limitation of the analysis is the granularity of the error reports: if the error is a *constraint failure* the CSP solver returns only the fact that no solution is available but no information about the cause of the constraint violation. This could cause that only the fact of an error can be reported but not its location.

In the future the analyser tool could be extended by

- the addition of further *bug patterns*. Such possible pattern would be to compare the name of the variable with the metamodel, and if the name is the name of a metamodel element, the type of the variable should be the same element (e.g. the variable `place` should have a type of `Place`).
- the CSP solver could be extended to provide *explanations* [16]. Explanations are a set of contradictory constraints, that could help to identify the cause of the constraint violation and thus more specific error reporting.

References

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

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

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

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

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

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

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

[18] 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.

[19] 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.

[20] 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).

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

[22] 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.

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

[24] 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.

[25] UJHELYI, Z., HORVÁTH, A., AND VARRÓ, D. A generic static analysis framework for model transformation programs. Technical report, Budapest University of Technology and Economics, June 2009.

[26] 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.