A Generic Static Analysis Framework for Model Transformation Programs

Zoltán Ujhelyi
uz602@hszk.bme.hu

Ákos Horváth
ahorvath@mit.bme.hu

Dániel Varró
varro@mit.bme.hu

June 26, 2009

Abstract

To ensure the correctness of complex model transformations tools that can find errors are necessitated. The goal of this paper is to define a static analysis framework which can detect some common errors in transformation programs.

The proposed tool is based on a graph model of the transformation program and defines the analysis criteria as abstract analysis problems.

1 Introduction

In modern software development processes model transformations have a crucial role. Such transformations can be assembled by a series of graph transformation [17, 8] steps that can be described by model transformation programs.

As the programs grow in size ensuring that their correctness becomes more and more difficult, nonetheless it is required as errors in the transformation program can propagate into the developed application.

Methods for ensuring correctness of computer programs such as *static analysis* are applicable for transformation programs as well. The main promise of static analysis is to detect a fixed set of errors without executing the program itself. Although static analysis cannot detect all kind of errors, in practice it can point out some of the most common ones in an early phase of development.

1.1 Objectives

The aim of this paper is to provide a static analysis framework for analyzing model transformation programs. To achieve this goal we first propose a graph model to represent those properties of the transformation program that are relevant to the analysis.

The most important goal of the research is to ensure that the analysis is *complete*: that means the framework should not omit errors (at least not the types looked for). On the other hand it is not so critical to ensure no error reporting happens when errors are reported that cannot occur during the execution of the transformation as this way to correctness of the transformation program (regarding the criteria to check) is ensured. This may help to achieve a faster analysis.

Reasonable speed is also a goal as static analysis tools can be integrated into Integrated Development Environments (IDE) where they could generate early feedback when repair is inexpensive.

1.2 Structure of the Report

The rest of the paper is structured as follows. Section 2 gives an introduction to the concepts used throughout the paper such as model transformations, metamodeling and static analysis. Section 3 gives an overview of the proposed analysis framework. Section 4 describes other approaches related to our research, and finally, Section 5 concludes our work and outlines possible directions of future research.

2 Background Concepts

2.1 Models and Transformations

Graph transformation languages such as the VIATRA2 [1] transformation language the ATL [3] describe model transformations as a series of *graph transformation* (GT) [8] rules with *graph patterns* as a condition definition. To define this series some kind of control structure is used (e.g. the abstract state machine [5] formalism).

The used control structure varies between the different graph transformation languages, although its basic goal is the same: the efficient execution of transformations by reducing the number of applicable GT rules.

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. Formally, a metamodel can be defined by a type graph. The nodes of the graph are called *classes*, while the arcs are referred as *associations*. The classes may have *attributes* describing further properties, and between classes *inheritance* relations may be used that means each property and association defined on the parent class is also available for the child class while the child may have additional ones. Associations might have *multiplicity* constraints attached to them, such as the at-most-one (0..1) or the arbitrary (∗) multiplicities.

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

The Petri nets are bipartite graphs with two disjoint sets 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 1.

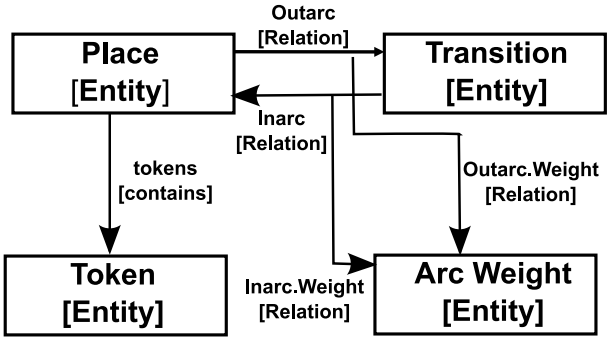


Figure 1: The graphical representation of the Petri net metamodel

Metamodels are used to describe the *instance models*; an instance model can be used in modeling environments and is a well-formed instance of the metamodel.

2.1.2 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 as application criterion, which describes the part of the model to be modified at rule application, while the postcondition describes how the match will look like after the rule application. Changes in the model are calculated as the difference between the precondition and postcondition patterns.

Example 2 *Figure 2 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.*

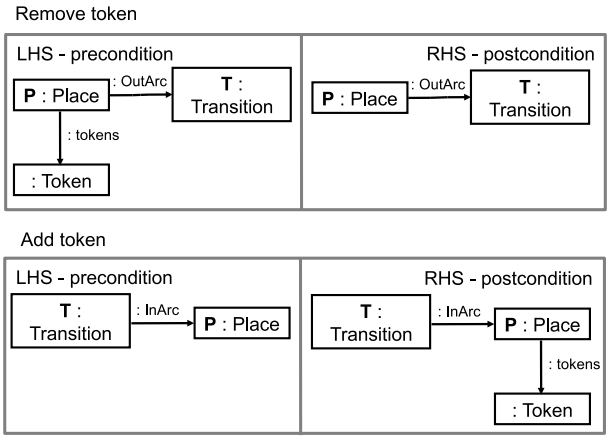


Figure 2: Graphical Representation of Graph Transformation Rules

The LHS graph pattern of the transformation consists of a `Transition` *(called* `T`*) and a* `Place` *(called* `P`*) connected by an* `InArc` *relation while the RHS pattern adds an unnamed* `Token` *element and a* `tokens` *relation which connects the imaged* `Place` `P` *and the newly created* `Token`*. After the execution of the rule a new* `Token` *is created and assigned to a* `Place` *as a consequence.*

2.1.3 Graph Patterns

Graph patterns are the atomic units of graph transformations. They represent a condition (or 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 mapped to a subgraph of the model using a *graph pattern matching*

technique. Basically this means that 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.

It is possible to write both positive and negative patterns: the positive pattern holds if the all conditions hold, but if a negative pattern condition can be satisfied, the pattern will fail. Both positive and negative patterns can be nested in an arbitrary depth thus reaching the expressive power of first order logic [16].

Example 3 *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 3.*

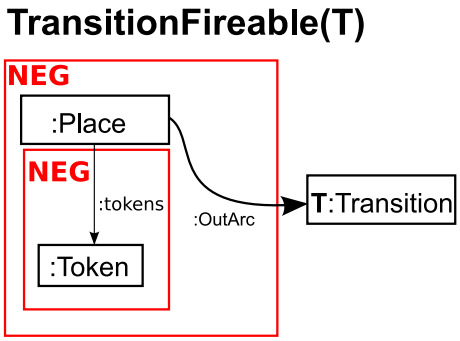


Figure 3: The Transition Fireable Graph Pattern

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

2.2 Static Analysis

It is a known fact in software engineering 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 by an order of magnitude.

The main promise of *static analysis* is to detect a predefined set of errors without running the application. In practice running the static analysis can only detect a limited set of errors, but for this limited 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# already use some kind of static analysis: during compilation they determine the types of variables, watch for uncaught exceptions, etc. These checks are performed during compile time thus facilitating the early identification of some common problems.

A very similar approach is possible for other structures. E.g. in case of Petri netsP- and T-invariants of the net [15] can be checked, which might be used to detect some serious modeling flaws - without the expensive calculation of the state space.

In our approach static analysis is carried out by an *abstract interpretation* [7] of the program, and the description of the computation is checked in this abstract universe. The execution of this abstract computation might offer some information about the actual computation.

Example 4 *A typical example for abstract interpretations is the rule of signs. In this case we are denoting the integers on the abstract universe of* $\{(+), (-), (\pm)\}$. *In this example* $-1517 \cdot 17$ *is abstracted as* $(-) \cdot (+) = (-)$, *and the properties of transformation prove 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 inaccurate.*

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

Static analysis can be powered by a number of abstract analysis techniques such as theorem proving [4] or constraint satisfaction programming [2].

3 Static Analysis of Transformation Programs

The proposed static analysis solution is based on the construction and traversal of a Transformation Program Model (TPM).

3.1 Transformation Program Models

The Transformation Program Model (TPM) is a graph model which is an abstract interpretation of the transformation program. The TPM omits information e.g. the current values of variables. The fact that the attached model space is not tracked allows the analyser to check every possible run path looking for some errors more efficiently.

The main reason to generate this graph model is that it makes the solution more flexible by separating the different tasks of the analysis as described in Figure 4. These tasks include the construction of the TPM model (see Section 3.2), the traversal of the TPM model (see Section 3.3), generating an

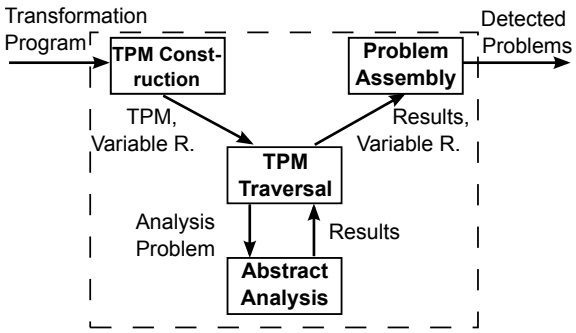


Figure 4: The TPM Based Static Analysis Process

Abstract Analysis Problem based on the TPM (see Section 3.4), and gathering the list of detected problems (see Section 3.5).

For the design of the TPM traversal the *visitor design pattern* [10] is used which allows the use of different traversal algorithms for different analysis criteria.

The variables of the transformation program is transformed into an abstract domain of the selected analysis criteria; these transformed variables will be referred as TPM variables. The value of TPM variables represent one or more parameter of the transformation program variable.

A very important difference between the TPM and the transformation program is that the TPM uses single assignment variables. This means that the value of the variable is set during initialization and bound to it.

The use of single assignment facilitates the generation of the analysis problem as many analysis methods are based on such variables.

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

To achieve full coverage in the analysis all these paths should be investigated. This could also be achieved by representing the TPM variables several times in the analysis problem but to avoid unnecessary memory consumption each branch is described as a separate problem; the analysis result can be calculated independently but the results should be aggregated.

Example 5 *The Conditional ASM Rule depicted in Figure 5 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, whose main

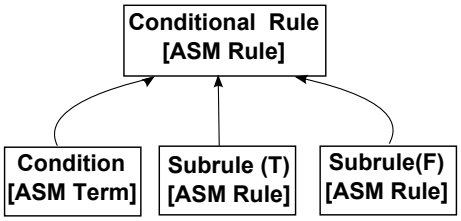


Figure 5: The TPM representation of the Conditional ASM Rule

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 errors that appear only on certain run paths.

3.2 Creating the TPM graph

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

The model is constructed during a traversal of the program 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 which are built from ASM constants, variables and functions.

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

Every TPM Node and variable should be associated to its source element: the element of the transformation program the Node is generated from. This association allows to describe errors simultaneously in the TPM graph and the transformation program. As the semantics of most nodes are exactly the same as its source element in the transformation program in this section only the differences are listed.

- For every potential failure (a concept similar to exceptions) a `Fail` node is explicitly inserted into the TPM. If the failure is conditional,

it should only be inserted into the corresponding branch (or branches). The TPM construction process should not care about failure handling - it is the responsibility of the traversal to find the next node in case of a failure.

- For each `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 are 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 be recursive. At runtime the interpreter can terminate the recursion using the values of variables but in the abstract interpretation this information is not (always) available, so in general it is possible that a recursive call represents an infinite length of calls.

  Currently this problem is handled by defining a universal *depth limit* k to describe a program by 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 an *sentinel node* representing no information is inserted.

  It is important to note that this depth limit 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 limit reduces the amount of available information before any analysis could happen (and thus it is possible that some errors might be unnoticeable) it is conservative: after a limit no information is collected and no false negative error detection can happen related to this limit.

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 building the analysis problem with the information of the 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 and the number of branches initialized in the current node.

According to the visitor pattern [10] the control of the traversal is handled by an external traversal control class, the *visitor*. In order to have the best error detection capabilities the visitor should be able to traverse every node in the TPM, identify and handle branches, build the analysis problem, update the variables in the connected Variable Repository, identify errors and handle fail nodes.

The *traversal of all nodes* is required in order to achieve full coverage of the transformation program. As the analysis tool should be capable of weaving the local information coming from the TPM nodes into a global analysis problem, the type of traversal does not matter (at least in theory). However in practice changing the traversal can help to find problem more efficiently.

Listing 1 introduces the used algorithm.

Listing 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.addNodeInformation(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 `traversal` method is used to manage the different branches, and for each branch a traversal is initiated 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 selected, the concrete traversal begins. The subnodes are grouped to (i) nodes to visit *before* generating the constraints and (ii) nodes to visit *after* extracting the information from the node. The algorithm traverses first the *before* nodes recursively, the information is passed into the abstract analysis tool, then updates the Variable Repository, and finally traverses the *after* nodes.

There are two cases which break that flow:

- If the analysis tool reports problems, 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 for running every possible branch one by one, and starting a clear analysis problem for each branch. Branch handling is based on a simple *backtracking algorithm*: when it reaches a branching point, it saves the current position as a 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 analysis tool reports a problem. After evaluating the results, the traversal of the next branch is started. In the new iteration 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 traversal, 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 object-oriented 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).

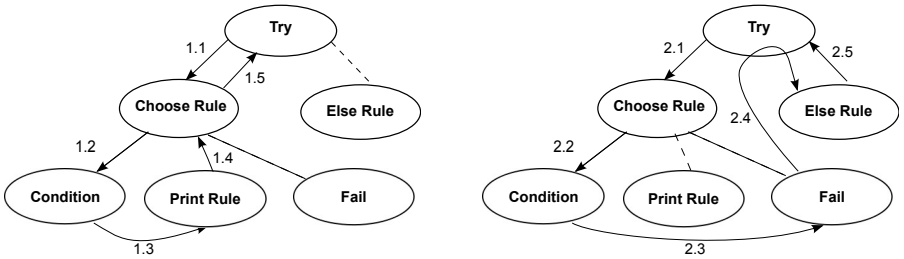
This jump is an alternate continuation of the program, so it has to be represented by a new branch. There are two rules which can fail: (1) the `fail` rule represents an automatic failure (because the failure is automatic, only a single branch is used, which jumps to the failure handler), and (2) the `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 its main rule, and executes its else rule if any failure is found, while (2) the rule `iterate` finishes iteration if a failure happens.

Example 6 *Listing 2 describes a single* ***`try`*** *rule. Figure 6 displays the potential execution paths of the structure*

Listing 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");
```



(a) No Failure (b) With Failure

Figure 6: The Execution of the Try Rule

The main rule to test contains a single ***`choose`*** *rule, while the failure handling rule (named Else in the Figure) is not shown in details. Solid arrows represent the control flow, while dashed arrows connects the nodes not present in the run path to their container.*

The first path in Figure 6a 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 terminates, and the control is returned to the caller, so the* ***`try`*** *block also terminates.*

On the other hand the second path in Figure 6b 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 is finished, the control is returned to the caller, and the `try` block also finishes.

This error handling mechanism can break the normal flow of the traversal. If no error handling node is found the traversal terminates.

3.3.3 Updating the Variable Repository

If a TPM node represents a change of a variable, when the traversal reaches the node the corresponding variables 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 by 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 that updates it.

The Variable Repository component besides storing the TPM variables supports create, update and query operations.

After a variable change is executed there are some properties that have not changed. As in the Variable Repository a new TPM variable is created, the unchanged properties has to be copied from the existing one. This copy has to be stated explicitly.

3.3.4 Enhancing the Performance of the Traversal

As described before for every branch a new analysis problem is initialized, and as described in Section 3.3.1 the TPM is traversed again to generate all the constraints. The speed of the analysis could be improved by avoiding the regeneration (and solution) of the same subproblems when the same nodes are repeatedly traversed in a different branch.

To achieve this at each 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 analysis tool. This means if the analysis tool does not support saving and restoring previously calculated elements this performance enhancement cannot be applied.

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 analysis tool and variable repository should be initialized from the saved state.

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

3.4 Analysis Problems

In order to get a static checker that is independent of any concrete analysis tool, an abstraction layer is implemented using the *Bridge* design pattern[10].

Based on the Bridge pattern a general interface is defined for handling the analysis tools: the interface allows for filling some predefined set of elements that will be used to build an analysis problem and the status of the analysis also needs to be queried.

The required set of elements depends on both the property to check and the analysis tool – the interface should be revisited each time a new criteria is added to the analysis or the analysis tool is changed for any reason.

The queries allow the traversal to check both the overall state – whether the current problem can represent a bug-free transformation program or not – and the detailed results. The specific queries are used to provide detailed analysis results after finishing the traversal, and can also be used to fine-tune the traversal in order to be more effective.

The filled elements are parameterized with TPM variables, the implementor has to read the analyzed properties and match them to variables of the analysis tool. This matching is bidirectional thus enables the Handler to return the TPM variable as a cause of failure.

As some analysis methods did not give details about the cause of problems, the analysis problem is built and checked incrementally: when a TPM node is reached during the traversal, the node generates the elements using TPM variables queried from the Variable Repository and asks the analyser whether the problem is correct.

When the problem cannot represent any bug-free transformation program, it is reported, and the traversal of the current branch is stopped. This method gives a hint where the problem has occurred. On the other hand this approach limits the number of errors that can be detected in a single iteration, but does not limit completeness as every error can be found one by one.

As of now we experimented with various constraint satisfaction problem [2] solvers, more specifically the Gecode/J [11] and the clpfd module of SICStus Prolog [6]. These experiments are detailed in [19].

3.5 Problem Identification

The static analyser system use three mechanisms to detect failures:

Analysis Problem It is recommended to try to identify errors during the traversal as during the traversal extra context information is available which can be used for more efficient detection of its cause. This context information is available naturally if the analysis tool reports an error, that happens only if that is related to a single TPM variable.

Inconsistencies It is possible that the error 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 checks such properties after a branch is traversed, and the results are saved for future branches.

Traversal Problem A third kind of errors to be identified is directly related to the traversal: if a traversal cannot successfully terminate, it also indicates an error. Such an error occurs when a failure node is found without a fail handling node. This kind of error terminates the analysis (similar to the execution of the transformation program).

To detect the different kinds of problems the system uses bug pattern detectors. Each of these detectors is a specialized pattern description, when the pattern is matched to the data calculated during the traversal, a problem is found.

It is also important to differentiate between problems by severity. Our model uses two severity categories: *error* and *warning*. Error means a serious bug, e.g., contradictory values conditions are detected on a TPM variable. This severity is used to describe bugs which cause problems during execution. Most analysis problems can be considered as errors. On the other hand, warnings are used to indicate potential problems that may or may not appear during runtime, e.g. a termination caused by an unhandled fail node may be the expected outcome, but this is not the recommended use of the structure.

4 Related Work

There are several static analysers used for different languages with a different set of capabilities based on different approaches. We now introduce two conceptually different approaches.

By annotating the program with a machine-readable specification, it is possible to check whether the code fulfills it. This concept is the basis of the *EFC/Java* (Extended Static Checker for Java) [9] tool, that uses the *Java Modeling Language* (JML) [13]. The JML is based on the "design

by contract" [14] notation. The EFC tool can check the preconditions, postconditions and invariants described by these contracts without executing the program itself. When the contracts are detailed enough the analysis can find violations but the analyser does not work with an unmodified program.

The FindBugs [12] tool handles the problem differently. It introduces the concept of *bug patterns*: possibly incorrect usage of the language can be described by a small pattern that should be searched for. By detecting such patterns it is possible to catch a few common errors while keeping the number of warnings relatively low[18]. 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.

5 Conclusion and Future Plans

In this paper 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.

It is possible to achieve full coverage (and thus a sound static analysis method) by traversing all branches of the TPM. The main limitation of this approach is that even a simple program can contain a lot of branches, and traversing all takes a lot of time.

The handling of cycles in the TPM graph could prevent covering every possible error: by limiting the depth it is possible that some errors remain undetected.

Another way to handle recursion is to follow the recursion until the result of the current iteration is exactly the same as was at the previous level. This state is called a *fixpoint*. The fact that the TPM is a truncated model of the run paths it is possible that the fixpoint is also truncated. As the fixpoint calculation can be at the same time more efficient (as sometimes it is not required to traverse as deep as the depth limit) and more precise (as the fixpoint can always be reached), in the future the depth limiting will be removed from the TPM building process, and depending on the analysis it will be moved to the traversal or replaced with fixpoint calculation.

As the branches following each other are similar (until the last branching point they are the same) it makes sense to *save the state* of the analyser at branching point and reuse them as needed, but in practice the execution time of the analysis increased using this optimalization.

In the future we plan to *modularize the traversal*: every pattern and rule can be evaluated one by one, and the result can be used to generate a contract

(or some kind of specification) for the element. After the contract has been generated, each call to the pattern or rule can be replaced by this contract.

As of today we implemented a static type checker [19] tool, but in the future other analysis methods could be added such as:

Reachability Analysis By altering the TPM traversal to travel backwards from a selected node the result can be interpreted as the condition to reach the starting point. If the conditions are contradictory the node cannot be reached.

Property Checking There are some constructs in the transformation program that provide some additional information not present in the program. E.g. after the execution of an `iterate` rule containing a single `choose` rule the condition of the `choose` rule must not hold. These properties could be used as additional information for other analysis methods.

References

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

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

[3] ATLAS Group. *The ATLAS Transformation Language*. Available from `http://www.eclipse.org/gmt`.

[4] Bibel, W., Korn, D. S., Kreitz, C., and Schmitt, S. Problem-oriented applications of automated theorem proving. In *DISCO* (1996), pp. 1–21.

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

[6] 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.

[7] 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.

[8] 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.

[9] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. *SIGPLAN Not. 37*, 5 (2002), 234–245.

[10] 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.

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

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

[13] Leavens, G. T., Baker, A. L., and Ruby, C. JML: a java modeling language. *In Formal Underpinnings of Java Workshop (AT OOPSLA'98* (1998).

[14] Meyer, B. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 1997.

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

[16] 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.

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

[18] 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.

[19] Ujhelyi, Z., Horváth, A., and Varró, D. Static type checking of model transformations by constraint satisfaction programmings. Technical report, Budapest University of Technology and Economics, June 2009.