Electronic Communications of the EASST
Volume 38 (2010)

Proceedings of the Fifth International Conference on Graph Transformation - Doctoral Symposium (ICGT-DS 2010)

Static Type Checking of Model Transformation Programs

Zoltán Ujhelyi, Ákos Horváth and Dániel Varró

16 pages

Guest Editor: Andrea Corradini
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Static Type Checking of Model Transformation Programs

Zoltán Ujhelyi, Ákos Horváth and Dániel Varró *

(ujhelyiz,ahorvath,varro)@mit.bme.hu, http://www.inf.mit.bme.hu/
Department of Measurement and Information Systems (MIT)
Budapest University of Technology and Economics (BME), Budapest, Hungary

Abstract: Model transformation is seen as a promising approach to automate software development and verification, thus improving quality and reducing production costs significantly. However, errors of transformation programs can propagate into the generated artifacts complicating the detection of errors. The current paper proposes a static type checking approach for early detection of typing errors of partially typed transformation programs. The approach describes type safety as constraint satisfaction problems, and uses a dedicated back-annotation mechanism for error feedback.

Keywords: model transformations, type checking, constraint satisfaction problems

1 Introduction

Model-driven development (MDD) has become a key technique in system and software engineering. MDD facilitates the systematic use of models from a very early phase of the design procedure using high-level, engineering models (such as UML, SysML or AADL - Architecture Analysis & Design Language). During development model transformations are used to generate appropriate mathematical models for formal analysis, deployment descriptors and source code.

Validating these model transformations is critical, as errors can invalidate the analysis results or might propagate into the target application. However, as the complexity of developed model transformations grows, ensuring the correctness of transformation programs becomes increasingly difficult. Several different methods and frameworks are already available for the verification of model transformation programs, including testing strategies [LZG05, KGZ09], model checking [RD06, LBA10] and static analysis [BW07, BCH+09]. However, there are many more formal techniques used to support the development and validation of traditional programming languages, and their application to model transformation programs can raise the maturity of the technology.

Albeit that static type checking is a common method to avoid or fix type errors (e.g. mismatched parameters, invalid variable assignments) in traditional programming languages, current model transformation frameworks using partially typed languages - having a statically typed (that check type safety at compile time) graph transformation rules with a dynamically typed (checked at runtime) control structure - have only very limited support for indentifying such errors.

The current paper presents a static analysis approach for the early detection of type errors in partially typed model transformation programs. Type safety is described using finite domain constraint satisfaction problems (CSP), where type constraints are created from every statement

* This work was partially supported by the SecureChange (ICT-FET-231101) and CERTIMOT (ERC_HU_09) projects.

of the transformation program based on the language specification. If these CSPs are unsatisfiable, a type error is detected and back-annotated to the transformation program. As the type constraints use concepts from the transformation language, the report is simply a compact representation of the contradicting type constraints.

The rest of the paper is structured as follows. Section 2 gives a brief overview of the technologies used in the paper. Section 3 details our approach with regards to identifying the type system, and creating constraint satisfaction problems that represent the type safety of the transformation programs. In Section 4 we demonstrate and evaluate the static type checking approach using an implementation in the VIATRA2 framework. Section 5 assesses the related work, and finally, Section 6 concludes our paper by evaluating the presented analysis approach and suggesting possible future research directions.

2 Preliminaries

2.1 Running Example: Simulation of Petri nets

In the current paper we will use the simulation of Petri nets as a model transformation problem to demonstrate the technicalities of our approach.

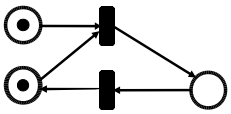


Figure 1: A Sample Petri net

Petri nets are bipartite graphs with two disjoint set of nodes: *Places* and *Transitions*. *Places* can contain an arbitrary number of *Tokens*, that represent the state of the net (marking). The process called *firing* changes this state: from every input *Place* of a *Transition* a *Token* is removed (if there is none to remove, the *Transition* must not fire), then to every output *Place* a *Token* is added. A sample Petri net model is depicted in Figure 1.

2.2 Foundations of Metamodeling

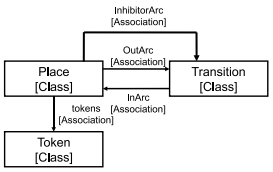


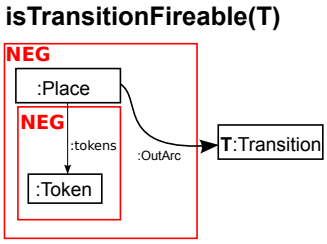
Figure 2: The Petri net metamodel

Metamodeling provides a structural definition (e.g. abstract syntax) of modeling languages. Formally, a *metamodel* can be represented by a type graph. Nodes of the type graph are called *classes*. *Associations* define connections between classes with possible *multiplicity* constraints on both ends declaring the number of objects that may participate in the association. A metamodel for Petri net models is depicted in Figure 2.

2.3 Graph Patterns and Graph Transformations

Graph patterns are often considered as atomic units of model transformations [VB07]. They represent conditions (or constraints) to be fulfilled by a part of the (input) models, and are used in model manipulation steps.

A basic graph pattern consists of graph nodes and edges corresponding to a metamodel. *Negative application conditions* (NAC) are an extension to this formalism, defining a negative subpattern to forbid contextual conditions for the original pattern in case of a successful match, or *alternate patterns* where the pattern if fulfilled if any alternate pattern matches. Existing patterns

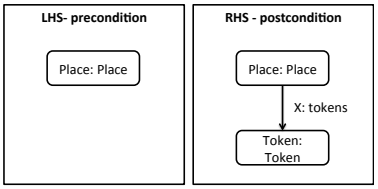


(a) Graphical Notation

```
pattern isTransitionFireable(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) Textual Notation

Figure 3: The Firing Condition of Petri nets



(a) Graphical Notation

```
gtrule addToken(in Place) = {
 precondition pattern place(Place) = {
  PetriNet.Place(Place);
 }
 postcondition pattern placeWithToken(Place, Token) = {
  PetriNet.Place(Place);
  PetriNet.Place.Token(Token);
  PetriNet.Place.tokens(X, Place, Token);
 }
}
```

(b) Textual Notation

Figure 4: The Add Token GT Rule

can be reused with *pattern composition*: the caller pattern holds only if the called pattern also holds with the specific graph nodes.

Example 1 *As an example, the firing enabled condition of the Petri net simulator program is displayed in Figure 3 as a graph pattern in the notation used in* VIATRA2. *The pattern uses nested negative application conditions to express that a* Transition *is enabled if every input* Place *instance collected to the* Transition *instance has at least one* Token *instance associated. In the example, the universal quantification is expressed with double negation of existence.*

Graph Transformation (GT) [Roz97] provides a high-level rule and pattern-based manipulation language for graph models. GT rules can be specified using a left-hand side (LHS or precondition) graph (pattern) to decide the applicability of the rule, and a right-hand side (RHS or postcondition) graph (pattern) which declaratively specifies the result model after the rule application. To achieve this, the rule application removes all elements only present in the LHS, creates all elements only present in the RHS, and leaves every other element unchanged.

Example 2 *Figure 4 shows a GT rule that specifies how to add a token to a place. The LHS pattern represents a single place, while the RHS pattern describes a* Token *connected to the* Place. *When this rule is applied, a* Place *is found, and a new* Token *with an association is created.*

```
rule fireTransition(in T) = seq {
 /* perform a check to confirm that the transition is fireable */
 if (find isTransitionFireable(T))
 seq {/* remove tokens from all input places */
  forall Place with find inputPlace(T, Place)
   do apply removeToken(T, Place); // GT rule invocation
  /* add tokens to all output places */
  forall Place with find outputPlace(T, Place)
   do apply addToken(T, Place);
 }}
```

Figure 5: The Fire Transition ASM rule

Control Language Complex model transformation programs can be assembled from elementary graph transformation rules using some kind of control language. In our examples, we use abstract state machine (ASM) [BS03] for this purpose as available in the VIATRA2 framework.

ASMs provide complex model transformations with all the necessary control structures including the sequencing operator (`seq`), ASM rule invocation (`call`), variable declarations and updates (`let` and `update` constructs), `if-then-else` structures, the simultaneous application at all possible matches (`forall`) and single rule application on a single match (`choose`).

Example 3 *The source code in Figure 5 demonstrates how is firing described in* VIATRA2*. At first the code determines whether the input parameter is a fireable* `Transition` *using the previously defined* `isTransitionFireable` *pattern. Then in a sequence the* `removeToken` *GT rule for each* `inputPlace`*, then the* `addToken` *GT Rule for every* `outputPlace` *are called.*

2.4 Type Checking and Type Inference

A type system of a language can be defined as "a tractable syntactic framework for proving the absence of certain program behaviours by classifying phrases according to the kinds of value they compute" [Pie02]. In other terms, the type system defines a categorization for statements of the program (typically for variables and terms).

During the execution of a statement the types of all used variables are ensured to fulfill selected *type constraints*. These constraints are simple logical expressions created using the specification of the executed statement and the current execution path. If the type constraints are insatisfiable, a runtime error is issued, as the selected statement cannot be executed.

In *statically typed* languages type information is either available, or easily inferred during compile-time, allowing precise type checking. *Dynamically typed* languages on the other hand postpone constraint validation until runtime. In these languages type constraints are often kept simple to avoid costly runtime validation, allowing some type errors to give incorrect output instead of an error message.

This can make such errors hard to identify, while they can be introduced easily by writing a statement with incorrect parameters, or correct parameters in an invalid order. To avoid such issues an efficient compile-time verification technique is needed.

A *static type checker* framework defines a compile-time approximation of the constraints, and evaluates it before execution. By creating an over-approximation it is possible to detect all

type errors by forbidding constructs that are often (but not every time) problematic. Similarly under-approximations can be used, in these cases all reported errors are valid, however some type errors might remain undetected.

Statically typed languages often require the developer the annotate variable definitions with type information, resulting in a simple *type checking* for analysis. However, type constraints are useful for both validating the usage of user-defined type information (called *type checking*) and calculating the proper types of variables based on their uses and definitions (*type inference*).

The type system of transformation programs consists of (*i*) built-in types of the transformation language (e.g. `string`, `integer`, `double` and `boolean` in VIATRA2 [VB07]) and (*ii*) the set of metamodel classes and associations used in the transformation program.

Type constraints can be constrain variables, terms containing variables and pattern conditions. The remaining language elements are not constrained directly, but are used to combine the existing constraints to detect insatisfiabilities.

Graph patterns or graph transformation rules are statically bound with user-defined metamodel types, while the control structure might use dynamic typing (such as ASM rules in VIATRA2).

The dynamic control structure makes it easy to call the GT rules and graph patterns with incorrect parameters that results in an empty match set (not a runtime error). E.g. switching the parameters of the `inputPlace` pattern call in Figure 5 causes the pattern to return with an empty match set, and the `removeToken` GT rule will not be called.

```
forall Place with find inputPlace(Place, T) // parameters in invalid order
 do apply removeToken(T, Place); // GT rule invocation
```

This and similar errors (e.g. invalid variable assignments) can be detected automatically using a type checker component. This type checking is possible because types can be inferred from the statically typed graph patterns, and propagated through the control structure.

2.5 Constraint Satisfaction Problems for Variables over Finite Domains

A CSP(FD) is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. In a more precise way a constraint satisfaction problem is a triple: (Z,D,C) where Z is a finite set of variables $x_1,x_2,...,x_n$; D is a function which maps every variable in Z to a set of objects of arbitrary type; and C is a finite (possibly empty) set of constraints on an arbitrary subset of the values of variables in Z. The task is to assign a value to each variable satisfying all the constraints. Solutions to CSPs are usually found by (i) *constraint propagation*: a reasoning technique to explicitly forbid values or domains for variables by predicting future subsequent constraint violations and (ii) *variable labeling*: searching through the possible assignments of values to variables already restricted by the (propagated) constraints.

3 Type Checking of Model Transformation Programs

3.1 Overview of the Approach

Our constraint-based type checking process is depicted in Figure 6. The inputs of the static analysis are the transformation program and the metamodel(s) used by the program, while its

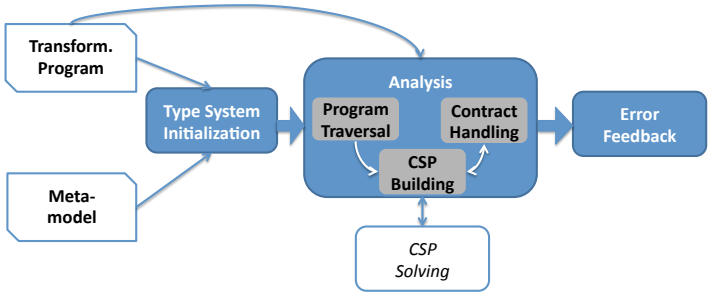


Figure 6: Overview of the Approach

output is a list of found errors, such as mismatched parameters. It is important to note that the instance models (input of the transformation program) are not used at all in the static analysis process. Our analysis process consists of the following steps:

Type System Initialization creates a representation of the type system of the transformation program, and makes it available for the analysis step, as described in Section 3.2.

Analysis The analysis phase can be split into three co-operating tasks:

Transformation Program Traversal processes every statement in every possible execution path of the transformation program. This task is carried out by a simple tree traversal algorithm operating on the abstract syntax tree of the transformation program.

CSP Building creates type constraints from every program statement, and calls a CSP solver to evaluate the generated problem. This step is detailed in Section 3.3.

Contract Handling is used to store (and return) partial analysis results as contracts. This task is described in Section 3.4.

Error Feedback back-annotates the analysis results to the transformation program to provide error messages to the transformation developer. For details see Section 3.5.

3.2 Type System Initialization

The first step of the analysis process is the identification of the type system (TS) of the transformation, and initializing it for the CSP solver library. As transformations often deal only with a selected aspect of models, such a type system consists only of a (potentially very small) subset of the source and target metamodels and some built-in types.

In order to calculate the used subset of the metamodels, we collect every type directly referenced from the transformation program, then for each type we add all their supertypes, and in case of relations type, their endpoint and inverse associations as well. This metamodel pruning method was described in [SMBJ09], and was proven to calculate a superset of the metamodel elements used in the transformation, providing a reduced metamodel with all needed elements for analysis.

After the type system is collected, to each remaining type a unique integer set was assigned in a way, that the set-subset relation between the integer sets represent the inheritance hierarchy in the type system. Informally, we define a mapping function $m : type \mapsto 2^{\mathbb{N}}$, guaranteeing

$\forall T_1, T_2 \in TS : supertypeOf(T_1, T_2) \Leftrightarrow m(T_1) \subset m(T_2)$. An algorithm for creating such a mapping from multiple inheritance hierarchies is proposed in [Cas93].

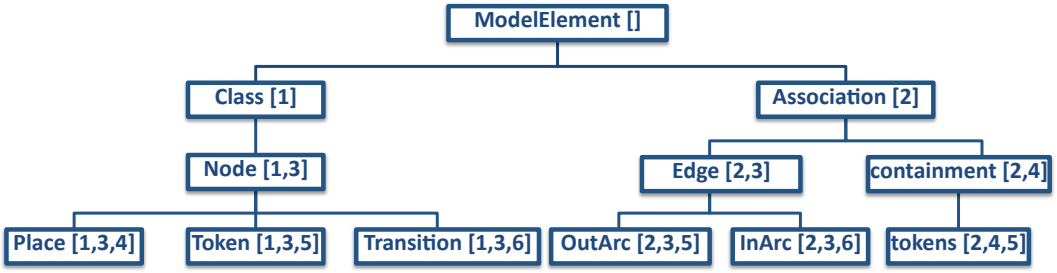


Figure 7: The Type System of the Petri net Simulator (without built-in types)

Example 4 *Figure 7 displays the type system of the Petri net simulator transformation program. The built-in types are omitted for the sake of readability, but they should be included in the top level of the tree, without child elements. The hierarchy also includes three additional types referring to every* model element, class *or* association *respectively - this is useful for expressing certain type constraints.*

For every type the associated integer set is also displayed on the figure. For types that are in supertype relation (such as the Node *and* Transition*) the sets are in subset relation ($m(Node) = \{1,3\} \subset \{1,3,6\} = m(Transition)$), while for those that are not (such as* Edge *and* Transition*) they are not ($m(Edge) = \{2,3\} \not\subseteq \{1,3,6\} = m(Transition)$).*

3.3 Constraint Generation

The type safety of transformation programs can be described as a finite domain CSP as follows. For each use of every transformation program variable a CSP variable is created with the domain of the integer sets defined in the type system (TS). These CSP variables represent *the type of a variable* of the transformation program, that will be matched with constraints representing the various uses of the variable.

These constraints are created from the *statements of the transformation program*, expressing type information available from the *language specification*. This information consists of the parameters and return values of the statement, e.g. conditions have the (built-in) `boolean` type.

By adjusting the constraint generation step it is possible to change whether the analysis creates an over- or under-approximation of the runtime type constraints. Less restrictive constraints can be created by creating type constraints directly from the language specification. On the other hand more restrictive constraints might be part of coding style policies aimed at the reduction of potentially (but not always) erroneous code.

In this paper we show how to create type constraints from graph patterns, type constraints from other language elements can be calculated similarly. All other type constraints derived from the transformation language of VIATRA2 are described in detail in [UHV09].

According to the body of graph patterns consists of a set of pattern variables with a type taken from the metamodel. These can be translated to constraints as follows: as the type of the pattern

variable has to be the same as (or a subtype of) the type defined in the pattern, a constraint representing the correct set-subset relation has to be created.

Pattern Graph Element	Type Information	Constraint
`Place(Pl)`	$typeOf(Pl) = Place$	$m(typeOf(Pl)) \subset \{1,3,4\}$
`Token(To)`	$typeOf(To) = Token$	$m(typeOf(To)) \subset \{1,3,5\}$
`tokens(X,Pl,To)`	$typeOf(Pl) = Place$	$m(typeOf(Pl)) \subset \{1,3,4\} \wedge$
	$typeOf(To) = Token$	$m(typeOf(To)) \subset \{1,3,5\} \wedge$
	$typeOf(X) = tokens$	$m(typeOf(X)) \subset \{2,3,5\}$

Figure 8: Constraint Generation from the *PlaceToken* Graph Pattern

Example 5 *Figure 8 displays the constraint generated from the* PlaceToken *NAC pattern (see in Figure 3b). The pattern defines a* `Place` *(called* `Pl`*) and a* `Token` *(*`To`*) pattern variables representing classes from the metamodel connected with a* `tokens` *association (*`X`*). The first column contains the different pattern graph elements, while the second one displays the derived type information. Finally, the third column shows the constraint that can be filled into CSP solvers.*

3.4 Contract Handling

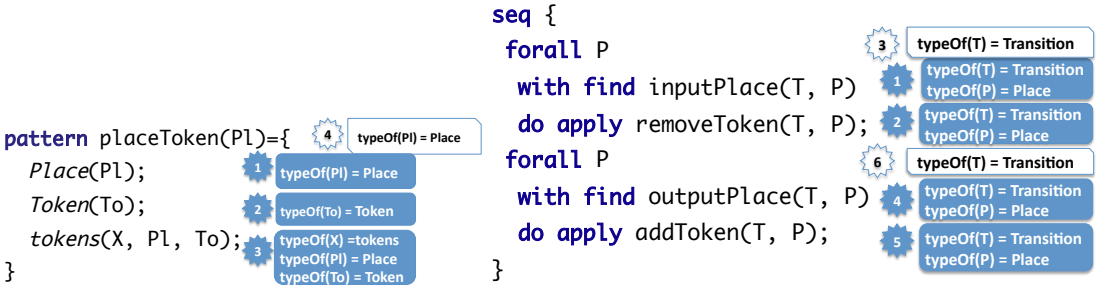
For performance considerations, the transformation program is traversed in a modular way: the transformation programs are split into smaller segments based on the natural structure of the program, that are traversed and analyzed separately. The partial analysis results of the segments are described and stored as pre- and postconditions. After being assigned to a segment, the contract is used to generate the relevant constraints instead of re-traversing the referenced segment.

Such a contract is basically a constraint that stores the externally visible properties of the contracted code segment. In case of type contracts, the type information of the externally visible variables are stored before (precondition) and after (postcondition) the execution of the code segment. This dual storage is only needed if the contract has to represent variable updates, otherwise it is enough to store a single unit of type information.

It is possible to assign a set of contracts to every code segment allowing to represent different behaviour in different execution paths. This construct allows reducing the number of execution paths to consider by filtering out execution paths providing the same type information, but retaining the different results. When reaching a segment with a set of contracts assigned, the segment has to be considered as a statement with multiple possible executions, thus maintaining the exhaustive nature of the traversal.

Call Contracts Callable elements, such as graph patterns queries, GT and ASM rules give a natural modularization of the transformation program: they are independent blocks running with their own set of variables. The type contracts of patterns and rules (*call contract*) have to store type information for its parameters, other variables used inside it are not visible externally. After that, every call to that pattern or rule can be replaced by the application of the call contract.

As the analysis of a callable element depends on other call contracts, it is important to calculate



(a) Call Contracts (b) Block Contracts

Figure 9: The Use of Contracts

the contracts in inverse call order: first the contract of the called element, then the callers. In case of recursive calls, no such order can be created.

To overcome this challenge, we propose to use a queue of callable elements, initially sorted by inverse call order (circular dependencies are broken). The queue is used to determine the order of contract calculation. After its contract is created or updated, we ensure that every caller of the rule is present in the queue by adding the missing elements, maintaining the inverse call order.

Example 6 *Figure 9a shows the generation of the call contract of the* placeToken *graph pattern. Next to each line in the pattern body the associated type information is presented* (1–3)*, in the order it is collected. After the constraints are evaluated,* (4) *the contract of the pattern is created. The pattern has a single parameter* Pl*, that is not changed, so the call contract contains a single type information:* $typeOf(Pl) = Place$*. This contract is displayed next to the pattern header.*

Block Contracts The analysis of control structures can also be modularized: blocks defining local variables (such as `let`, `choose` or `forall` rules in ASM) may also offer a compact contract. These *block contracts* store type information for the variables defined outside the block.

Blocks may be embedded into each other, but no circular embedding is possible, thus the ordering of block contract calculation is much simpler: first the contract of the innermost block has to be calculated, then the outer ones. This way, no recalculation of contracts is needed.

Example 7 *Figure 9b shows an example for generating block contracts in the* fireTransition *ASM rule. The rule contains two* `forall` *blocks to create contracts from. To calculate the type of the first rule,* (1) *we read the contracts from the* `inputPlace` *pattern and* (2) *the* `removeToken` *GT rule - both state, that* $typeOf(P) = Place \wedge typeOf(T) = Transition$*.* (3) *The contract of the block does not contain locally defined variables, so the only variable present in the contract is* T*. The contract of the other* `forall` *rule can be calculated similarly* $(4-6)$*.*

The performance gains of both call and block contracts over the naive, non-modular traversal approach are detailed in Section 4.3.

Statement	Type Information	Constraints
`with`	$typeOf(T) = Transition$	$m(typeOf(T)) \subset \{1,3,6\}$
`find inputPlace(T,P)`	$typeOf(P) = Place$	$m(typeOf(P)) \subset \{1,3,4\}$
`do`	$typeOf(P) = Transition$	$m(typeOf(P)) \subset \{1,3,4\} \wedge$
`apply removeToken(P,T)`		$m(typeOf(P)) \subset \{1,3,6\}$
	$typeOf(T) = Token$	$m(typeOf(T)) \subset \{1,3,6\} \wedge$
		$m(typeOf(T)) \subset \{1,3,4\}$

Figure 10: Error Detection

3.5 Error Reporting

Most CSP solvers do not give detailed output in case of an unsatisfiable constraint set, only the unsatisfiable variables are reported. In order to obtain context information, we evaluate the constraints in parallel with the traversal. This way, in case an error is found, both the variable and the last processed statement are known, allowing to associate the error to this segment.

Additionally, every CSP variable and constraint is linked with its source variable or program statement, so it is possible to find the related elements.

If the CSP solver reports an unsatisfiable set of constraints, it means that the various uses of a variable (e.g. its definition and its use) expect *incompatible types*, indicating a type error, that can be presented to the transformation developer as an error. The most common way such errors manifest are parameter mismatching in calls.

Example 8 *The first column of Figure 10 shows two modified lines from the previously introduced* fireTransition *ASM rule: the parameters of the* removeToken *call are switched, which is a common error committed by transformation developers. In this case the analysis first uses the contract of the* inputPlace *pattern to determine that* $typeOf(T) = Transition$ *and* $typeOf(P) = Place$*. After that, it applies the contract of the* removeToken *GT rule. At this point, the solver has the following constraints for the variable P:* $m(typeOf(P)) \subset \{1,3,4\} \wedge m(typeOf(P)) \subset \{1,3,6\} \Leftrightarrow m(typeOf(P) \subset \{1,3,4,6\}$*, that is not a subset of any allowed set from the type system. Finally, a failure is reported, that no type can satisfy the constraints for variable P.*

As values (and possibly types) of the transformation program variables can change during execution, multiple CSP variables might be needed to represent them. After the constraints are evaluated, it also needs to be checked whether the CSP variables return a single, consistent type from the transformation program variable. Inconsistency indicates that the *type of a transformation program variable changes* during execution. Such changes are almost always unintended, but as dynamic languages allow it, only a warning is displayed to the developer.

4 Implementation and Evaluation

4.1 Implementation

The proposed static type checker framework was implemented and evaluated in the VIATRA2 model transformation framework and integrated into its Eclipse-based user interface.

The implementation defines an under-approximation of the type constraints, and evaluates them with various available CSP solvers for Java, most notably the Gecode/J[1] and the `clpfd` module of SICStus Prolog[2]. A preliminary performance comparison between the different solvers is described in [Ujh09]. In our experience neither of them supported our needs well: Gecode/J became slow with larger sets (about 50 metamodel elements), while SICStus Prolog did not support incremental CSP building required for error reporting.

Thus, we created a simple CSP solver tailored to our requirements. Incremental evaluation is supported by maintaining a constraint graph with variables as nodes and constraints as arcs and propagating information from new type constraints using optimized propagation rules.

4.2 Case Studies

To demonstrate the analysis capabilities, we evaluated larger transformation programs with more complex metamodels next to the Petri net firing example. As examples we tried to select different kind of transformation programs: (1) the Petri net simulator and generator are very simple basic transformation programs, (2) the AntWorld case study present a larger simulation case study, (3) while the BPEL2SAL case study represents and industrial-sized model transformation.

The Petri net generator program In addition to the Petri net simulator program a generator transformation is also defined in [BHRV08]. This transformation is used to generate Petri nets with an approximately equal number of *places* and *transitions* as test cases for the firing program.

The nets are created using the inverse of six reduction operators that preserve safety and liveness properties of the net. The operators are selected using a weighted random function.

The generator program consists of 9 patterns, 5 GT and 9 ASM rules with a control structure using more elaborate ASM rules.

The AntWorld Case Study The AntWorld case study [Zü08] is a model transformation benchmark featured at GraBaTs 2008. It is based on the ant colony optimization problem and simulates the life of a simple ant colony searching and collecting food to spawn more ants in a dynamically growing world. The ant collective forms a swarm intelligence, as ants discovering food sources leave a pheromone trail on their way back so that the food will be found again by other ants.

The case study uses a turn-based simulation, with each turn divided into seven different phases for ant simulation (e.g. grad, search) and world management (e.g. create ants, boundary breached).

The metamodel of the case study is somewhat larger than the Petri net metamodel: it consists of 7 different classes with complex associations between them. The transformation program consists of 17 graph patterns and 13 relatively simple ASM rules as control structure.

The BPEL2SAL Transformation Business processes implemented in BPEL (Business Process Execution Language) are often used to create business-to-business collaborations and complex web services. Their quality is critical to the organization and any malfunction may have a significant negative impact on financial aspects. To minimize the possibility of failures, designers and

[1] http://www.gecode.org/gecodej/
[2] http://www.sics.se/isl/sicstuswww/site/index.html

	LOC	Calls (P/G/A)	Time (naive)
Petri net simulator	120	12/2/3	0,1s
Petri net generator	94	9/5/9	2s
Antworld	300	17/0/13	24min
BPEL2SAL	8339	177/0/102	--

(a) Execution Time with the Naive Traversal Algorithm

		# of contracts	# of execution paths	Avg # of paths in a contract	Max # of paths in a contract	Analysis time
Petri net simulator	Call contract	17	26	1,53	4	0,1s
	Block contract	40	49	1,23	3	0,1s
Petri net generator	Call contract	23	62	2,70	33	1,2s
	Block contract	57	97	1,70	33	0,9s
Antworld	Call contract	30	47	1,57	4	0,2s
	Block contract	56	79	1,41	4	0,3s
BPEL2SAL	Call contract	279	1291	4,63	576	--
	Block contract	1248	1568	1,26	12	69s

(b) Execution Times with the Modular Traversal Approaches

Figure 11: Runtime results

analysts need powerful tools to guarantee the correctness of business workflows. The BPEL2SAL transformation program [GHV10] is used inside a tool for such analysis.

Both the BPEL and SAL metamodels used in the transformation are much more complex than the AntWorld case study, together consisting of 150 classes and associations between them. To express this transformation, 177 different graph patterns have been defined together with 102 ASM rules, some of them are really complex (over 100 lines of code).

4.3 Performance Assessment

In order to evaluate the performance we measured the execution time of the analysis on the various case studies. We compared the execution time of a *naive traversal* (without contracts) to the modular traversals, using either *call contracts* or both *call and block contracts*. During the measurements we used error-free programs as erroneous execution paths are only analyzed until the first unsatisfiability found, thus shortening analysis time. For measurements we used a developer notebook with a 2.4 GHz Core2Duo processor and 4 GB RAM with a 64 bit Java SE runtime. Measurements were repeated several times, and the average of the analysis time is used.

We tested memory consumption by limiting the available heap size to 500 MB - the analysis could handle every tested transformation. A more detailed evaluation is planned for the future.

Figure 11a summarizes the size of the different transformation programs with the analysis time using the naive traversal approach (without modularization). The first column displays size of the program (number of code lines), while the second the number of graph patterns, GT rules and ASM rules respectively. The third column shows the analysis time. The BPEL2SAL transformation did not terminate in several hours, so its result was omitted.

Figure 11b consists of our measurement results related to the modular traversal approaches: (1) when using only *call contracts* and (2) using both *call and block contracts*. The first column displays the number of contracts created, in case of block contracts holding both call and block contracts. The second column then describes the total number of partial execution paths the analysis must traverse: the total number of different execution paths inside contracted elements.

The third column shows the average number of execution paths per contract. We believe, the performance of the analysis correlates with this number, as it shows how many times a call or block has to be evaluated before its contract can be created. Even worse, a large number of execution paths usually suggests a complex call or block resulting in large CSPs to solve.

The fourth column displays the maximum number of execution paths per contract. In most

cases it is similar to the average value, with the notable exception of the BPEL2SAL program. When only call contracts are used, there exists a complex ASM rule responsible for 576 paths, almost half of the total number. We believe, that the use of block contracts reduce this maximum to 12 causes that the analysis becomes possible. Similarly, if the code would be refactored in a way that every call contains only a single block, the performance would increase similarly.

The last column displays the execution time of the analysis. These results show, that even the use of call contracts can reduce the execution time significantly: the AntWorld transformation can be analyzed in 0.2 seconds instead of 24 minutes. Similarly, the use of block contracts allowed to analyze the BPEL2SAL transformation program in about 1 minute.

It is important to note, that the use of block contracts can also increase the analysis time: in case of the AntWorld case study the increased administrative overhead of maintaining contracts outweighed the benefits of the contract generation. As Figure 11b shows, the average number of execution paths is only slightly smaller using block contracts, however, the total number of paths to evaluate grows significantly.

The type checking is performed before the transformation is executed, so it does not cause slowdowns during runtime. However, as the parsing of various transformation programs happens fast (in case of the BPEL2SAL transformation it needs about 5 seconds, otherwise it happens instantly), the type checking should be executed in a similar time frame in order to maintain the speed of the framework.

In case of smaller programs the analysis time is comparable with the parse time, and the combined parsing and type checking could be executed every time when the transformation is saved. However, the BPEL2SAL transformation is analyzed much slower than its parse time, so further optimizations are needed to provide a fast, integrated analysis. This issue is partly mitigated by the fact that the analysis is executed in the background, thus it does not block the development environment, allowing the developer to work during the analysis.

5 Related work

In this section we give a brief introduction to various approaches to the verification of model transformations, and also compare our system with existing type checking solutions.

Verification of Model Transformations Various analysis methods are being researched for the verification and validation of model transformations. Testing methods are very common in the field of traditional software engineering, and its applications to model transformation programs are actively researched [LZG05, KGZ09], but early results show that testing or comparing the output of transformation programs (usually models) can be very time consuming.

Formal methods, such as theorem proving based approaches [Pen08] show the possibility to prove statement validity over graph-based models. For the verification of dynamic properties model checking seems promising [LBA10, Ren04], but the challenge of infinite state spaces needs to be overcome, e.g. by creating an abstraction [RD06] or by statically computing a Petri graph model [KK06]. However, these techniques usually do not scale well and are cumbersome to apply to industrial size problems.

In addition to model checking, static analysis techniques have been used in the verification

of static properties. They provide efficiently calculable approximations of error-free behaviour, such as unfolding graph transformation systems into Petri nets [BCH+09], creating and validating OCL constraints derived from GT rules [CCGL08], or using a two-layered abstract interpretation introduced in [BW07]. Our approach works similarly: we transform the model transformation program into constraint satisfaction problems, and verify it statically.

The PROGRES environment allows the definition of static semantic rules on the model [EJS95], that are evaluated incrementally when the model changes. These rules are more expressive than our type constraints, and are evaluated with an algorithm similar to the arc consistency-based propagation algorithm in CSP solvers, but the basic goal is different. Semantic rules are used to describe well-formedness constraints over the model, while type constraints are generated from transformation programs.

Type Checking Approaches As the number of type checking frameworks and algorithms is extremely large, we focus only on the approaches for dynamically typed functional and logical programming languages solve because of their similarities with model transformations.

The well-known Hindley-Milner algorithm [Mil78] for lambda calculus reduces the typing problem to a unification problem of equations, and is widely used in the functional programming community. [JVWS07] extends to this algorithm, supporting higher-order functions.

Our approach was influenced by the work started in [Pot05] which translates the typing problem to a set of constraints. As the type mapping of lambda calculus does not support inheritance hierarchies required for the transformation languages, we designed a different mapping and evaluation approach that fits better the graph based data structures and multi-level metamodeling.

Type checking of Prolog programs works differently: the basic task of type checking is to infer the concrete type, represented as a hierarchical tree structure from its basic uses. A typical approach is to calculate with different kinds of widening [VB02, Lin96] steps.

Other approaches exist as well for the type checking of logical programming languages: [HCC95] creates type graphs to represent the various Prolog structures, and uses abstract interpretation techniques to validate the program, while [HM04] traces back type safety of Datalog languages to the consistency of ontologies.

6 Conclusion and Future Work

We have presented a static type checker approach for model transformation programs. It was implemented for the VIATRA2 transformation framework, and evaluated using transformation programs of various size. In our initial evaluation the type checker seemed useful for early error detection as it identified typing errors related to swapped variables or erroneous pattern calls.

As for the future, we plan to enhance the expressiveness of the analysis by identifying the subset of well-formedness constraints that can be included for type checking. We'd also like to enhance the performance by various optimizations, such as the merging of similar traces or using a single-pass algorithm to handle recursive calls to avoid possible recalculations and ordering.

In addition, the inferred types can be used to detect indirect dependencies between parts of the transformation program (readers and writers of a selected type). Identifying these dependencies allow the creation of static program slicing solutions for model transformation programs.

This would allow to generate meaningful traces for reaching possibly erroneous parts of the transformation programs, thus helping more precise error identification.

Bibliography

[BCH+09] P. Baldan, A. Corradini, T. Heindel, B. König, P. Sobociński. Unfolding Grammars in Adhesive Categories. In *Proc. of CALCO '09 (Algebra and Coalgebra in Computer Science)*. P. 350–366. Springer, 2009. LNCS 5728.

[BHRV08] G. Bergmann, Á. Horváth, I. Ráth, D. Varró. A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In *Proc. 4th International Conference on Graph Transformations, ICGT 2008*. Pp. 396–410. Springer, 2008.

[BS03] E. Börger, R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[BW07] J. Bauer, R. Wilhelm. Static Analysis of Dynamic Communication Systems by Partner Abstraction. In *Static Analysis*. Pp. 249–264. Springer Berlin / Heidelberg, 2007.

[Cas93] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. Pp. 271–287. ACM, New York, NY, USA, 1993.

[CCGL08] J. Cabot, R. Clarisó, E. Guerra, J. Lara. An Invariant-Based Method for the Analysis of Declarative Model-to-Model Transformations. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. MoDELS '08, pp. 37–52. Springer-Verlag, Berlin, Heidelberg, 2008.

[EJS95] W. Emmerich, J.-H. Jahnke, W. Schäfer. Object Oriented Specification and Incremental Evaluation of Static Semantic Constraints. Technical report, Universität Paderborn, 1995.

[GHV10] L. Gönczy, Á. Hegedüs, D. Varró. Methodologies for Model-Driven Development and Deployment: an Overview. In *Results of the* SENSORIA *project on Software Engineering for Service-Oriented Computing*. Springer-Verlag, 2010. To appear.

[HCC95] P. V. Hentenryck, A. Cortesi, B. L. Charlier. Type analysis of Prolog using type graphs. *The Journal of Logic Programming* 22(3):179–209, Mar. 1995.

[HM04] J. Henriksson, J. Małuszyński. Static Type-Checking of Datalog with Ontologies. In *Principles and Practice of Semantic Web Reasoning*. Pp. 76–89. LNCS 3208, 2004.

[JVWS07] S. P. Jones, D. Vytiniotis, S. Weirich, M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17(01):1–82, 2007.

[KGZ09] J. Küster, T. Gschwind, O. Zimmermann. Incremental Development of Model Transformation Chains Using Automated Testing. In *Model Driven Engineering Languages and Systems*. Pp. 733–747. Springer Berlin / Heidelberg, 2009.

[KK06] B. König, V. Kozioura. Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In *TACAS '06: Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 197–211. 2006.

[LBA10] L. Lúcio, B. Barroca, V. Amaral. A Technique for Automatic Validation of Model Transformations. In *13th Model Driven Engineering Languages and Systems*. Pp. 136–150. LNCS 6394, 2010.

[Lin96] T. Lindgren. The impact of structure analysis on Prolog compilation. Technical report UPMAIL Technical Report No. 142, Uppsala University, 1996.

[LZG05] Y. Lin, J. Zhang, J. Gray. A Testing Framework for Model Transformations. In *Model-Driven Software Development*. Pp. 219–236. Springer Berlin Heidelberg, 2005.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17(3):348–375, Dec. 1978.

[Pen08] K. Pennemann. Resolution-Like Theorem Proving for High-Level Conditions. In *Graph Transformations*. Pp. 289–304. Springer Berlin / Heidelberg, 2008.

[Pie02] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, USA, 2002.

[Pot05] F. Pottier. A modern eye on ML type inference. 2005. In Proc. of the International Summer School On Applied Semantics (APPSEM '05).

[RD06] A. Rensink, D. Distefano. Abstract Graph Transformation. *Electronic Notes in Theoretical Computer Science* 157(1):39–59, May 2006.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *Applications of Graph Transformations with Industrial Relevance*. Pp. 479–485. 2004.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.

[SMBJ09] S. Sen, N. Moha, B. Baudry, J. Jézéquel. Meta-model Pruning. In *Model Driven Engineering Languages and Systems*. Pp. 32–46. Springer Berlin / Heidelberg, 2009.

[UHV09] Z. Ujhelyi, A. Horváth, D. Varró. Static Type Checking of Model Transformations by Constraint Satisfaction Programming. Technical report TUB-TR-09-EE20, Budapest University of Technology and Economics, June 2009.

[Ujh09] Z. Ujhelyi. Static Analysis of Model Transformations. Masters thesis, Budapest University of Technology and Economics, 2009.

[VB02] C. Vaucheret, F. Bueno. More Precise Yet Efficient Type Inference for Logic Programs. In *9th International Symposium on Static Analysis*. Pp. 102–116. Springer, 2002.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.* 68(3):214–234, 2007.

[Zü08] A. Zündorf. AntWorld benchmark specification, GraBaTs. 2008.