# Static Type Checking of Model Transformation Programs

Zoltán Ujhelyi

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2
ujhelyiz@mit.bme.hu

## 1 Introduction

Model transformations, utilized for various tasks, such as formal model analysis or code generation are key elements of model-driven development processes. As the complexity of developed model transformations grows, ensuring the correctness of transformation programs becomes increasingly difficult. Nonetheless, error detection is critical as errors can propagate into the target application.

Various analysis methods are being researched for the validation of model transformations. Theorem proving based approaches, such as [5] show the possibility to prove statement validity over graph-based models. For the verification of dynamic properties model checking seems promising, but abstractions are needed to overcome the challenge of infinite state spaces [6].

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 [1], or using a two-layered abstract interpretation introduced in [2].

The current paper presents a static analysis approach for early detection of typing errors in partially typed model transformation programs. Transformation languages, such as the one of VIATRA2 [9], are often partially typed, e.g. it is common to use statically typed (checked at compile time) graph transformation rules with a dynamically typed (checked during execution) control structure.

The lack of static type enforcements in dynamically typed parts makes typing errors common and hard to trace, on the other hand the static parts allow efficient type inference. Our type checker approach uses constraint satisfaction problems (CSP) to propagate information between the different parts of the transformation program. Error messages are generated by a dedicated back-annotation method from the constraint domain.

## 2 Overview of the Approach

### 2.1 Type Safety as Constraint Satisfaction Problems

To evaluate type safety of transformation programs as CSPs, we create a CSP variable for each potential use of every transformation program variable with

the domain of the elements of the type system. The constraints are created from the statements of the transformation program, expressing the type information inferrable from the language specification (e.g. conditions have a boolean type). For details about the constraint generation process see [8].

After the CSPs are evaluated, we check for two kinds of errors: (1) typing errors appear as *CSP violations*, (2) while variable type changes can be identified by comparing the different CSP variable representations of transformation program variables, looking for *inconsistencies* (although this may be valid, it is often erroneous, thus a warning is issued).

### 2.2 The Analysis Process

Our constraint-based type checking process is depicted in Fig. 1. The input of the static analysis is the transformation program and the metamodel(s) used by the program, while its output is a list of found errors. It is important to note that the instance models (that form the input of the transformation program) are not used at all in the static analysis process.

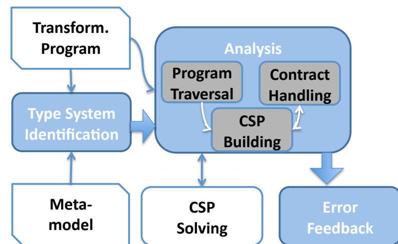


**Fig. 1.** Overview of the Approach

At first, we collect every possible type used in the transformation program (*Type System Identification*). The type system consists of the metamodel elements and built-in types (e.g. string, integer). To reduce the size of the type system, we prune the metamodel to provide a superset of the types used in the transformation program as described in [7]. The type system contains all elements referenced directly from the transformation program, all their parents, and in case of relations their endpoints and inverses.

To prepare the type system for the CSP based analysis, a unique integer set is assigned to each element of the type system as proposed in the algorithm in [3] to allow efficient calculation of subtype relationships.

Then in the *Analysis* step we traverse the transformation program, building and evaluating constraint satisfaction problems. Multiple traversal iterations are used to cover the different execution paths of the transformation problem.

For performance considerations a modular traversal is used: graph transformation rules and graph patterns are traversed separately. The partial analysis results of the rules (or patterns) are described and stored as pre- and postconditions based on the "design by contract" [4] methodology. After a contract is created for every reference to its corresponding rule (or pattern) the contract is used to generate the relevant constraints instead of re-traversing the referenced rule.

This modular approach is also used for the traversal of the control structure: it is similarly divided into smaller parts that are traversed and contracted separately.

The created CSPs can be solved using arbitrary finite domain CSP solver tools. The results are both used to build type contracts and to provide error

feedback. The type contract of a rule (or pattern) holds the calculated types of the parameters at the beginning and the end of the method (the difference in the pre- and postcondition implies a type change in the parameter variable).

Finally, we look for typing errors and type changes to back-annotate them to the transformation program (*Error Feedback*). CSPs are evaluated together with the traversal to make context information also available to associate related code segment(s) to the found errors.

## 3 Implementation 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 inital evaluation the type checker seems useful for early error identification as it identified errors related to swapped variables or pattern calls.

As for the future, we plan to evaluate the possible usage of static program slicing methods 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. The generated slices are also usable to extend the system with additional validation options such as *dead code analysis* to detect unreachable code segments or *use-definition analysis* to detect the use of uninitialized or deleted variables.

## References


1. Baldan, P., Corradini, A., Heindel, T., König, B., Sobociński, P.: Unfolding Grammars in Adhesive Categories. In: Proc. of CALCO '09 (Algebra and Coalgebra in Computer Science). p. 350–366. Springer (2009), LNCS 5728
2. Bauer, J., Wilhelm, R.: Static Analysis of Dynamic Communication Systems by Partner Abstraction. In: Static Analysis, pp. 249–264. Springer Berlin / Heidelberg (2007)
3. 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. pp. 271–287. ACM, New York, NY, USA (1993)
4. Meyer, B.: Applying 'design by contract'. Computer 25(10), 40–51 (1992)
5. Pennemann, K.: Resolution-Like theorem proving for High-Level conditions. In: Graph Transformations, pp. 289–304. Springer Berlin / Heidelberg (2008)
6. Rensink, A., Distefano, D.: Abstract Graph Transformation. Electronic Notes in Theoretical Computer Science 157(1), 39–59 (May 2006)
7. Sen, S., Moha, N., Baudry, B., Jézéquel, J.: Meta-model Pruning. In: Model Driven Engineering Languages and Systems, pp. 32–46. Springer Berlin / Heidelberg (2009)
8. Ujhelyi, Z., Horváth, A., Varró, D.: Static Type Checking of Model Transformations by Constraint Satisfaction Programming. Technical Report TUB-TR-09-EE20, Budapest University of Technology and Economics (Jun 2009)
9. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. 68(3), 214–234 (2007)