STATIC ANALYSIS OF MODEL TRANSFORMATIONS

Zoltán UJHELYI
Advisor: Dániel VARRÓ

I. Introduction

Model transformations have a crucial role in the model-driven development [1] processes as they form the basis to derive source code from high level descriptions. Usually complex model transformations are captured by transformation programs.

As these programs grow in size ensuring their correctness becomes increasingly difficult, nonetheless it is required as errors in these programs can propagate into the developed application.

Methods for ensuring correctness of computer programs such as *static analysis* are applicable for transformation programs as well. Static analysis represents a set of techniques for computing different properties of programs without their execution. It is extensively used both in compiler optimization and program verification.

As computing the properties of the program can be infeasible to calculate - or even undecidable, - static analysis tries to calculate an approximation of these properties. In case of using an over-approximation, it can be guaranteed that no *bugs are missed*, similarly using an underapproximation may imply that no *spurious warnings* (an error message about a bug not present in the application) are emitted.

A widely used static analysis methods is abstract interpretation [2], that is based on the concept of abstract domains. The elements of the abstract domains categorize the concrete values (as present in the code) by a selected property. Using abstract interpretation the analyzer evaluates the program on these abstract domains to provide an approximation of the behavior.

II. Overview of the Approach

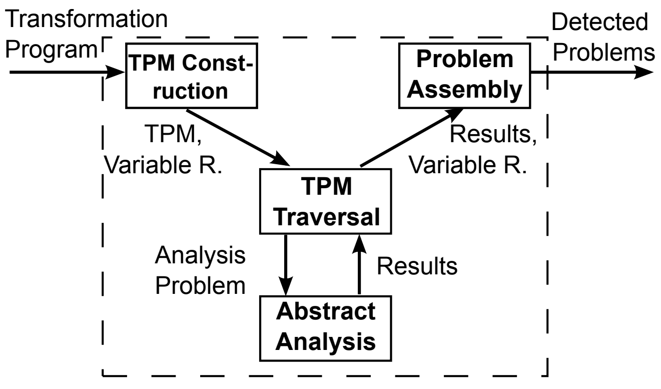

Figure 1: Overview of the Analysis

We implemented a static type checker component for the VIATRA2 model transformation system [3] that maps the variables of the transformation programs to their types, and validates it. The component is implemented in a generic way in order to support the analysis of other properties. For type checking every variable of the program has been mapped to the abstract domain of types, and the type information have been inferred following every possible execution path.

The static analysis process consists of three separate steps: (1) a Transformation Program Model (TPM) is built for storing an abstract representation of the program, (2) the model is analyzed using an abstract analysis tool and (3) the problems are connected. Figure 1 shows an overview of the static analysis process.

The TPM model is created to represent the transformation program in abstract domains - e.g. in case of type checking every variable is replaced with its type.

The main part of the static analysis consists of traversing the TPM model, and incrementally building and evaluating an analysis problem. The analysis problem of the type checking task is implemented as a constraint satisfaction problem [4], where program variables are mapped to constraint variables, and type information is represented in the domains of CSP variables.

In order to give useful error messages to the transformation developer, the analysis results have to be back-annotated to the transformation program. This is achieved by the incremental building of the CSP from the TPM, which allows to pin-point the faulty variable. The back-annotated error messages fit into the following categories: (1) *analysis problems* mean the abstract analysis fails, (2) *inconsistencies* happen when multiple analysis iterations report contradictory values and (3) *traversal problems* indicate that the traversal could not finish the analysis properly.

The whole analysis approach is described in more details in [5].

III. Enhancing the Static Analysis Framework

As with all static analysis technique the applicability of the static analysis framework to real life scenarios is a crucial question, and mainly relates to the performance aspect. My framework traverses every execution path separately, and the number of execution paths can increase exponentially with the size of the transformation programs.

Therefore significant performance increase can be achieved by a *modularized traversal*, that divides the transformation program into smaller, verifiable parts.

As the parts of the transformation program interact with each other, the relation between the parts have to be described. For this reason a *contract* [6] is attached to every validated part that describes a part of the static analysis results, that are relevant to the interaction between the different parts - e.g. in case of type checking the inferred types of the parameters are added to the contract. After the contract is created, every use of the program part can be replaced by its contract.

IV. Evaluation and Future Plans

Type errors are hard to detect manually as transformation programs can be executed without runtime errors, only the output differs from the expected outcome. The created static type checker tool could detect these type errors that makes it a usable addition to the VIATRA2 framework.

The performance of the tool using the modularized traversal is also acceptable: it was capable of type checking of a real-world transformation program [7] in a reasonable amount of time (a few minutes).

As for the future we are planning to extend the system with other validations such as *dead code analysis* to detect code segments that cannot be reached and *use-definition analysis* to detect use of uninitialized or deleted variables. Additionally we plan to evaluate the performance of the analysis using SAT solvers as the underlying abstract analysis tool.

References

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

[2] P. Cousot and R. Cousot, “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*, pp. 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[3] D. Varró and A. Balogh, “The model transformation language of the VIATRA2 framework,” *Science of Computer Programming*, 68(3):214–234, October 2007.

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

[5] Z. Ujhelyi, A. Horváth, and D. Varró, “A generic static analysis framework for model transformation programs,” Technical Report TUB-TR-09-EE19, Budapest University of Technology and Economics, June 2009.

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

[7] M. Kovács, D. Varró, and L. Gönczy, “Formal Analysis of BPEL Workflows with Compensation by Model Checking,” *IJCSSE*, 23(5), November 2008.