DEF-USE ANALYSIS OF MODEL TRANSFORMATION PROGRAMS WITH PROGRAM SLICING

Zoltán UJHELYI
Advisor: Dániel VARRÓ

I. Introduction

Model transformations, used 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 grow, ensuring their correctness 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, such as model checking [1] or static analysis [2] based approaches. However, there are further methods supporting the development and validation of traditional programming languages, and their application to model transformation programs can raise the maturity of the technology.

Program slicing [3] techniques are used to calculate parts (called *slices*) of a program that (potentially) affect the values computed at a selected point of the program. The generated slices can be used in several ways: (1) they can be simply displayed to the developer as a filtered version of the program; (2) as the slices represent a small, cohesive unit of the program, various types of analysis can be executed on them instead of the entire program, resulting in performance and/or precision gain; and (3) as the dependencies between program statements are explored to generate the slices, this information can be re-used in different analysis techniques.

The current paper presents a slicing approach for model transformation programs based on *dependence graphs*. The generated slices are then used to perform *def-use analysis* to check for *uses of undefined variables* and *unused variables* (approach 3). We demonstrate this approach using the transformation language of the VIATRA2 model transformation framework [4] over a Petri net simulator transformation, however it is applicable to other languages and transformations as well.

Petri nets are bipartite graphs with two disjoint set of nodes: *Places* and *Transitions*. *Places* can contain an arbitrary number of *Tokens*, that represents the state of the net (marking). The simulator changes this state by a process called *firing*: 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.

II. Overview of the Approach

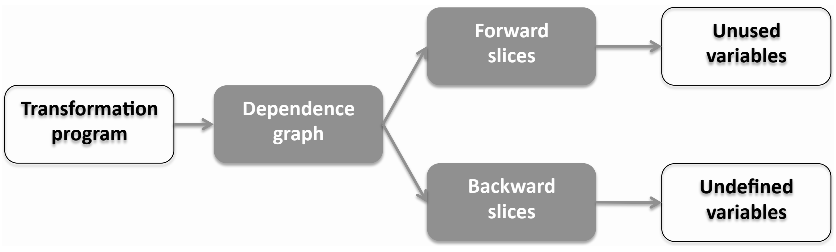

Figure 1: Overview of the Approach

Our analysis process is depicted in Figure 1. At first, the transformation program is read, and the dependencies between the program statements are collected into a *system dependence graph* (SDG) [5].

These graphs are labeled graphs, their nodes represent the program statements, while the edges describe the various dependency relations between them, such as control, data or call dependencies. The calculation of these graphs is detailed in Section III.

The SDG can be used to calculate two different kind of slices: forward and backward slices.

The *backward slicing problem* of a program can be summarized as follows: for a selected program statement we are looking for the (superset of the) statements the selected statement depends on. Such slices can be calculated by traversing the SDG: starting from the node of the selected statement every statement has to be reached, that the selected statement depends on (both directly or indirectly).

Backward slices can be used to detect uses of *undefined variables*, that are read before having been defined. If the backward slice of a statement using a variable does not contain a definition (or assignment) for the used variable, it should be reported as error. When the slice consists of multiple execution paths, every execution path is checked for a definition. A special case of these undefined variables is the *reading of output parameters*: a parameter of a called procedure that is supposed to be returned is undefined before it is set.

Similarly, the *forward slicing problem* can be summarized as follows: for a selected program statement we are looking for the (superset of the) statements depending on the selected statement. These slices can also be calculated by the traversal of the SDG: the dependencies have to be traversed in the other direction then in case of backward slices.

Unused variables can be detected by looking for the forward slices of variable definitions. In case the slice does not contain at least a single statement reading the variable, the variable is unused. In case of multiple execution paths we check whether the variable is used in at least a single path. The *writing of input parameters* is a special case of unused variables: changes to an input parameter are lost after rule termination.

III. Calculating Dependence Graphs in Model Transformation Programs

The transformation language of VIATRA2 [4] uses the formalism of *graph transformation* (GT) to describe elementary transformation steps between graph models, and uses abstract state machines (ASM) as control structure to build complex transformation programs. The dependence graphs of the GT and ASM rules have to be calculated differently, as GT rules offer a declarative description of transformation steps with loosely defined control flow, while ASM rules define a rich, precisely specified one. In the following we detail the calculation of such dependence graphs for both parts.

A. The Dependence Graph of ASM Rules

ASM rules provide similar control structure as imperative programming languages, making their dependence graph basically the same as the one described in [5]: the statements of the transformation program are represented as nodes, while control, data and call dependencies are depicted as arcs between them.

ASM rules may have directed parameters: incoming parameters are initialized when the rule is called, while outgoing parameters are expected to be initialized by the rule. These dependencies are presented in the dependence graph by incoming/outgoing data dependency arcs.

Block rules, such as `forall` or `let` rules define local variables that have to be considered as dependencies as well. As the language permits having the same variable name to be defined in multiple blocks, the analysis must distinguish between these instances - for this reason the dependence graph uses different variable instances for these elements.

Example 1 *Figure 2 displays an ASM rule of the Petri net simulation program together with its dependence graph. The rule first calls the* isTransitionFireable *pattern as a condition, then applies the* removeToken *GT rule for all places that are results of the* inputPlace *pattern. Then this is repeated similarly for the* outputPlace *pattern and the* addToken *GT rule.*

In the dependence graph solid lines represent call and control dependencies, while jagged lines represent data dependencies - their labels describe the corresponding variable.

The dependence graph describes how the variables are connected. It is important to note, that although the Pl *variables in the two* `forall` *rules have the same name, but describe different variables - these variables are called* Pl_1 *and* Pl_2 *respectively.*

The called patterns and GT rules are omitted from the graph for readability reasons - only the calls and entry points are displayed. In order to calculate the required slices, these dependence graphs are also to be calculated.

```
rule fireTransition(in T) = seq {
 // graph pattern call in a condition
 if (find isTransitionFireable(T))
 seq {
  forall Pl
   // graph pattern call
   with find inputPlace(T, Pl)
   // GT rule application
   do apply removeToken(T, Pl);
  forall Place
  // graph pattern call
   with find outputPlace(T, Pl)
  // GT rule application
   do apply addToken(T, Pl);
 }
}
```

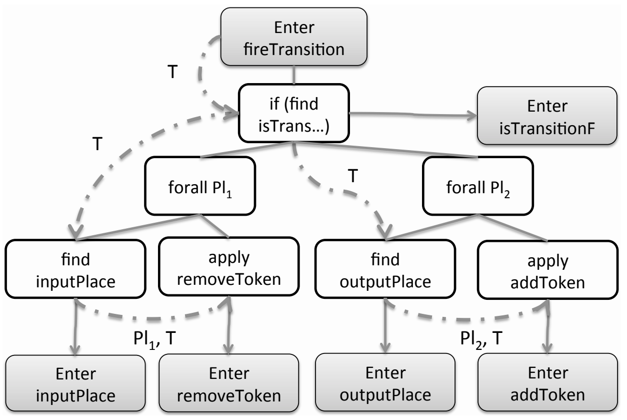

Figure 2: The Dependence Graph of the fireTransition ASM Rule

B. The Dependence Graph of Graph Transformation Rules

Graph transformation provides a high-level, declarative rule and pattern-based manipulation language for graph models. GT rules can be specified using a precondition graph (pattern) to decide applicability, and a postcondition graph (pattern) to specify the result model after rule application. To achieve this, the rule application removes all elements, that are only present in the precondition graph, creates all elements, that are only present in the postcondition, and leaves every other element unchanged.

Data dependencies are easy to calculate inside graph patterns, as the pattern graph is described with a series of pattern variable definitions. If such a definition uses another pattern variable, a data dependency is identified. However, when a pattern is called, its incoming parameters might or might not be initialized. Initialized pattern variables depend on the pattern call, but are not modified, while uninitialized parameters are calculated during the pattern matching process. This means, unlike ASM rules, data dependencies of a graph pattern cannot be evaluated completely without looking at its callers.

This external information is available when calculating the dependencies of the caller GT rule: pattern variables of the precondition pattern are initialized, if they are incoming parameters of the rule (explicitely specified in the syntax), while pattern variables of the postcondition pattern are initialized if they were initialized in the precondition.

Example 2 *Figure 3 displays a token manager GT rule of the Petri net simulator with its dependence graph. Its precondition pattern (called* LHS*) consists of a single* Place*, while its postcondition (called* RHS*) consists of a* Place *and a* Token *connected by a* tokens *relation.*

The LHS *pattern does not have any internal dependency, while the* tokens *relation in the* RHS *pattern depends on both variables* Pl *and* To*. As the* Pl *variable is an incoming parameter of the rule, it is an external dependency of the* LHS *pattern. Similarly the* RHS *pattern depends on the* Pl *variable from the* place *pattern, however, the* To *variable is initialized inside the pattern.*

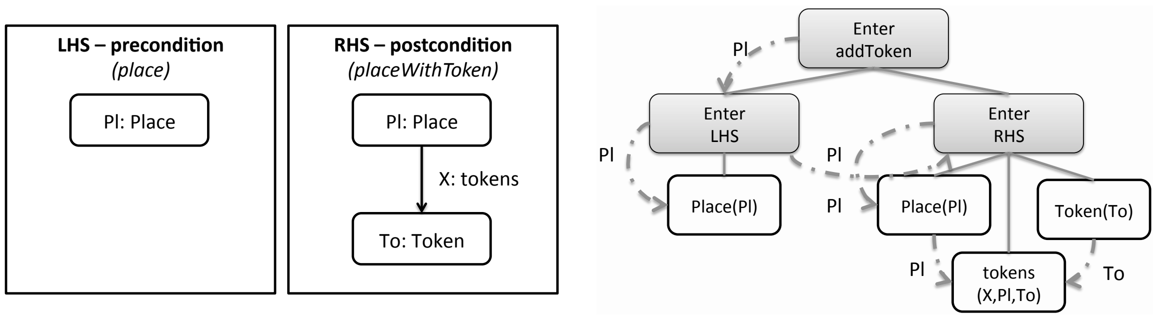

Figure 3: The Dependence Graph of the addToken GT Rule

In case of graph patterns and GT rules it is possible that their application creates modifications in the (input) model, that are not directly visible in transformation program variables. Next to the internal and external data dependencies mentioned before, it is also possible to establish indirect dependencies between graph patterns and GT rules. As the rules look for a match for their input models, and alter them, the applicability of other rules might change as well without any explicit connection between the rules in the transformation program.

A way to detect these dependencies is to consider each model element of the (input) models as a variable of the transformation program. However, as the models can be complex, abstractions are needed for this approach to be feasible. The use of type system provides such an abstraction [6]. This way, each model element can be represented by a type variable, and this type variable could be used to represent possible dependencies between model elements.

IV. Conclusion

In this paper we presented an approach to create slices of model transformation programs based on dependence graphs, and used the slices to perform a def-use analysis: detect undefined variables and unused variable definitions statically.

For the future we plan to find other uses of the generated slices. By displaying the slices to the transformation developer it is possible to get a better understanding of the structure of the transformation program. Similarly, the slices may be used to calculate the set of statements relevant to some condition (e.g., expressed as a graph pattern).

In order to produce smaller slices we plan to use a more precise dependency calculation. To achieve this we will investigate further model abstractions and compare their slice sizes to the one generated using the type system abstraction. The use of dynamic slicing also promises reduced slice sizes since next to the program structure it also uses the input variables to filter out impossible dependencies.

References

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

[2] J. Bauer and R. Wilhelm, “Static Analysis of Dynamic Communication Systems by Partner Abstraction,” in *Static Analysis*, pp. 249–264. Springer Berlin / Heidelberg, 2007.

[3] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Ákos Kiss, and B. Korel, “A formalisation of the relationship between forms of program slicing,” *Science of Computer Programming*, 62(3):228–252, Oct. 2006.

[4] D. Varró and A. Balogh, “The Model Transformation Language of the VIATRA2 Framework,” *Science of Computer Programming*, 68(3):214–234, October 2007.

[5] S. Horwitz and T. Reps, “The use of program dependence graphs in software engineering,” in *Proceedings of the 14th international conference on Software engineering*, pp. 392–411, Melbourne, Australia, 1992. ACM.

[6] Z. Ujhelyi, “Static type checking of model transformation programs,” in *Graph Transformations*, vol. 6372 of *Lecture Notes in Computer Science*, pp. 413–415. Springer Berlin / Heidelberg, 2010.