Towards Dynamic Backward Slicing of Model Transformations

Zoltán Ujhelyi, Ákos Horváth, Dániel Varró
Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
Email: {ujhelyiz,ahorvath,varro}@mit.bme.hu

***Abstract*—Model transformations are frequently used means for automating software development in various domains to improve quality and reduce production costs. Debugging of model transformations often necessitates identifying parts of the transformation program and the transformed models that have causal dependence on a selected statement. In traditional programming environments, program slicing techniques are widely used to calculate control and data dependencies between the statements of the program. Here we introduce program slicing for model transformations where the main challenge is to simultaneously assess data and control dependencies over the transformation program and the underlying models of the transformation. In this paper, we present a dynamic backward slicing approach for both model transformation programs and their transformed models based on automatically generated execution trace models of transformations.**

***Index Terms*—Program slicing; Model transformations;**

I. Introduction

Model transformations (MT) are frequently used in model-driven design (MDD) to simultaneously improve quality and reduce development costs by providing early model validation and automating various phases of software development including code, test case or configuration generation. Model transformations take one (or more) source model(s) as input and produce one (or more) target model(s) as output (together with detailed traceability information in model synchronization scenarios). In-place transformations may also operate on the same model instance to provide model simulation or refactoring.

Model transformations are captured in the form of a MT program, which is a regular piece of software in a certain sense. However, elementary transformation steps in MT programs are highly data-oriented using declarative rules, while complex transformations are assembled from elementary steps using "regular" control structures. As a consequence of this necessarily hybrid nature of MT languages, rich existing results of software engineering cannot be adapted in a straightforward way for MT programs, especially for designing complicated, industrial scale MTs, where debugging and validation plays a crucial role.

In traditional programming environments, program slicing techniques are widely used to calculate control and data dependencies between the statements of the program. When debugging MTs, transformation experts would require similar support to identify parts of the MT program which have causal dependence on a selected statement of the MT program (called slicing criterion). However, the slicing criterion of a MT program can also depend on an element of the underlying model when a read or write operation on the element causes a causal dependency.

Unfortunately, up to our best knowledge, no slicing techniques are available for MT programs. The adaptation of existing program slicing techniques turns out to be non-trivial as MT programs take models as an additional input. Therefore, slices of a MT program should simultaneously incorporate the causally dependent statements *and* the causally dependent parts of the underlying model.

A further difference to a traditional (dynamic) program slicing setup comes from the fact that MTs are executed mostly at design time, in modern integrated development environments (IDEs). IDEs frequently save additional information when running a transformation in order to provide undo/redo support. Consequently, execution traces of a MT run are readily available to support slicing.

Here, we introduce model transformation slicing by first interpreting traditional program slicing characterisations for model transformation programs. Then the main technical challenge is to detect the mutual causal dependencies of declarative transformation rules and imperative control structures both on the transformation program and on the transformed models. For instance, declarative model queries issued by transformation rules might introduce data dependencies that can only be detected precisely by creating relevant slices of the transformed models as well.

In this paper, we present a dynamic backward slicing approach for simultaneously identifying affected parts of both model transformation programs and illustrate it using a simple simulation transformation program.

II. Slicing of Model Transformations

Program slicing approaches were originally developed for imperative style programming languages, and have been exhaustively surveyed in the past in papers like [1]. They have also been applied to logic programs [2], [3] by augmenting data-flow analysis with control-flow dependencies.

In the context of MDD similar techniques were proposed for model reduction purposes [4]. In case of UML models Lano et. al. also presented [5] the creation of UML slices of

both declarative (like pre- and postconditions of methods) and imperative elements (state machines).

A traditional program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest [1].

The slicing problem of MTs receives three inputs: the *slicing criterion*, the *model transformation program*, and the *models* on which the MT program operates. As an output, slicing algorithms need to produce (1) *transformation slices*, which are statements of the MT program depending or being dependent on the slicing criterion, and (2) *model slices*, which are parts of the model(s) depending (or being dependent) causally on the slicing criterion (due to read or write model access).

The slicing criterion denotes the point of interest, and it is specified by a location (statement) in the MT program in combination with a subset of the MT program variables or an element in the underlying model the MT operates on.

1) Static Slicing vs. Dynamic Slicing: Program slicing approaches frequently distinguish between between a static and a dynamic slice [1]. Static slices are computed without making assumptions regarding a program's input, whereas the computation of dynamic slices relies on a specific execution (test case) of the program.

The direct interpretation of this distinction for calculating *model slices* during transformation slicing is problematic as model transformations always operate on some input models, thus static slicing also have to make some assumptions on the input models at least. In case of dynamic slicing, the actual input models are available to identify the affected model elements wrt. a slicing criterion. While in case of static slicing, one can still rely upon the *well-formedness constraints of the modeling language* (typically captured in some declarative constraint languages like OCL or graph patterns) which need to be fulfilled by all valid models.

Interpreting static and dynamic slicing for *transformation program slices* is more straightforward. In case of dynamic slicing, we calculate the statements of the MT program affected the slicing criterion with respect to a specific execution (run) of the MT program (on a specific model). While in case of static slicing, all statements of a MT program are included which may potentially affect the slicing criterion along at least one run of the MT program.

In case of static slicing, we typically overapproximate the affected model elements or program statements to guarantee that a static slice contains all potentially affected artifacts.

2) Backward Slicing vs. Forward Slicing: Traditional program slicing also distinguishes between backward and forward slicing [1], [6], [7], which definitions can be applied to MTs. A *forward slice* consists of (1) all statements and control predicates of an MT program and (2) elements of the underlying model dependent on the slicing criterion. A statement is dependent on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine whether the statement under consideration is executed or not. A model element is dependent on the slicing criterion if it is touched (read, created or removed) as a direct causal consequence of the slicing criterion. In case of a *backward slice*, we identify MT program statements and model elements which the slicing criterion is dependent on.

TABLE I: Characterization of Model Transformation Slicing

		Model Slicing	Transformation Program Slicing
Criteria		Program Statement and Variable or Model Element	
Dynamic Slicing	Backward	Model Elements the Criteria depends on	Program statements the Criteria depends on
	Forward	Model Elements depending on the Criteria	Program statements depending on the Criteria
Static Slicing	Backward	All model elements the criteria potentially depends on the Criteria	All program statements the criteria potentially depends on
	Forward	All model elements potentially depending on the Criteria	All program statements potentially depending on the Criteria

An overview of the different combinations of cases for model transformation slicing is provided in Table I. In the current paper, we tackle dynamic backward slicing for model transformations, while other cases are left for future work.

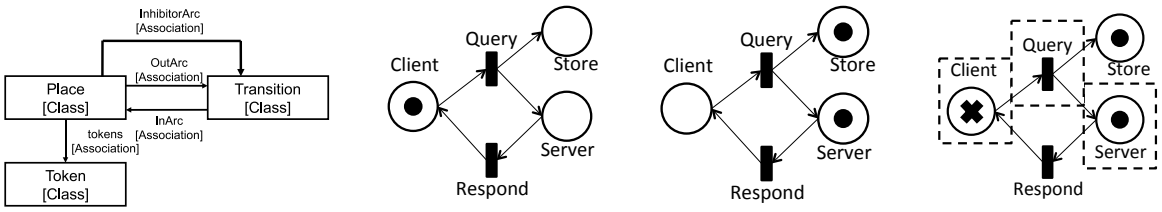
Model Transformation Slicing vs. Program Slicing: The main conceptual difference from existing program and MT slicing problems primarily lies in *simultaneous* slicing of the models and the transformation program. Consequently, the slicing criterion itself may contain an element of the underlying model (instead of a program variable) in addition to a location in the MT program.

Another practical difference from traditional program slicing comes from the fact that model transformations are typically executed at design time in popular integrated development environments (IDEs) like Eclipse where execution traces of a transformation are persisted as undo/redo stacks to support revoking all effects of a transformation (transactional effects). This undo stack may serve as a rich additional input, especially, for dynamic slicing of model transformations (which we also exploit in the current paper).

Furthermore, this definition of model transformation slicing is independent from the actual MT language used for specifying the model transformation. So while we use a dedicated language (namely, the language of VIATRA2 transformation framework [8]) later on to demonstrate the technicalities of our approach, similar slicing problems can be addressed for other MT languages as well.

III. MODEL TRANSFORMATION SLICING: AN EXAMPLE

In this section, we informally present the core technique of dynamic backward slicing of MT programs using a demonstrative example of Petri net simulation formalized by model transformations in the VIATRA2 framework. This example is frequently used to demonstrate how model transformations can be used in model simulation scenarios, furthermore, it already served as a performance benchmark for MTs [9]. Furthermore,



(a) Metamodel for Petri nets

(b) A Simple Petri net

(c) The Model after Firing the Query Transition

(d) The Calculated Model Slices

Fig. 1: A Petri net Example

the choice of the VIATRA2 language was motivated by the fact that it includes both declarative and imperative language elements, thus our slicing results are likely to be extensible for other MT languages as well.

A. Running Example: Simulation of Petri nets

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: a token is removed from every input place of a transition, and then a token is added to every output place of the firing transition. If there are no tokens at an input place of a transition, the *Transition* is disabled, and thus it cannot fire. The structure of the modeling language of Petri nets is formalized by a corresponding metamodel in Fig. 1a.

Example 1: Fig. 1b depicts a simple Petri net. The net consists of three *places*, representing a Client, a Server and a Store and two *transitions*. If the Client issues a query (the Query transition fires), the query is saved in the store (a token is created), the Server gets the control (another token is created), and the client waits for a response (the Respond transition fires and a token is removed).

B. The VIATRA2 Transformation Language

1) Graph Patterns: Graph patterns are often considered as atomic units of MTs [8]. They represent complex structural conditions (or constraints) that are to be fulfilled by a part of the (input) models. Graph patterns are also used to declaratively define model manipulation steps.

Example 2: The `sourcePlace` (Line 1) pattern in Figure 2 is used to identify the source places of a transition. The pattern consists of a `Transition` node `Tr` and a `Place` node `Pl` connected with an edge of type of `OA`. It is important to note that as only variables `Tr` and `Pl` appear in the header of the pattern, they can be received from the caller of the pattern as input parameters, or passed back to the caller as output parameters. The variable `OA` is an internal pattern variable not available outside the pattern.

2) Graph Transformation: Graph transformation (GT) 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. This is achieved by removing all elements only present in the LHS, creating all elements only present in the RHS, and leaving every other element unchanged.

Example 3: Figure 2 presents two simple GT rules that are (respectively) used to add a token to or remove a token from a place. The LHS pattern of the `addToken` (Line 19) pattern consists of a single *place*, while its RHS extends it with a *token* and an edge. This means, applying the rule creates a token and connects it to the *place*.

3) Control Language: Complex MT programs can be assembled from elementary graph patterns and graph transformation rules using some kind of a control language. In our examples, we use abstract state machines 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, and single or simultaneous application at possible matches (**`forall`** and **`choose`**).

Example 4: The `fireTransition` rule (Line 27) in Figure 2 describes the firing of a transition in VIATRA2. At first the code determines whether the input parameter is a fireable `Transition` using the `isTransitionFireable` pattern. Then in a sequence the GT rule `removeToken` is called for each `sourcePlace`, followed by a call to GT rule `addToken` for every `targetPlace`.

C. A Sample Dynamic Backward Slice

To illustrate the MT slicing problem, we consider the execution of the rule `fireTransition` called with the transition `Query` as a parameter. Fig. 1c displays the model after the execution: the token from the place `Client` is removed, while a token is added to places `Store` and `Server`. As a *slicing criterion*, we selected the invocation of the GT rule `addToken` in Line 34 together with variable `Pl`.

We can calculate the backward slices for the criterion as follows. (1) At the last item of the trace the variable `Pl` is bound during the matching of the pattern `targetPlace`, so the pattern invocation is part of the slice (Line 33). (2) As the pattern matching of `targetPlace` uses model elements (`Server`, `Query` and the `IA` between them), they have to be

```
1  pattern sourcePlace(Tr, Pl) = {
2       Transition(Tr);
3       Place(Pl);
4       Place.OutArc(OA, Pl, Tr);
5  }
6  pattern targetPlace(Tr, Pl) = {
7       Transition(Tr);
8       Place(Pl);
9       Transition.InArc(IA, Tr, Pl);
10 }
11 pattern place(Pl) = {
12   Place(Pl);
13 }
14 pattern placeWithToken(Pl) = {
15   Place(Pl);
16   Place.Token(To);
17   Place.tokens(X, Pl, To);
18 }
19 gtrule addToken(in Pl) = {
20   precondition find place(Pl)
21   postcondition find placeWithToken(Pl)
22 }
23 gtrule removeToken(in Pl) = {
24   precondition find pattern placeWithToken(Pl)
25   postcondition find pattern place(Pl)
26 }
27 rule fireTransition(in T) =
28  if (find isTransitionFireable(T)) seq {
29   /* remove tokens from all input places */
30   forall Pl with find sourcePlace(T, Pl)
31     do apply removeToken(Pl); // GT rule invocation
32   /* add tokens to all output places */
33   forall Pl with find targetPlace(T, Pl)
34    do apply addToken(Pl));
35 }
```

Fig. 2: Slicing criterion: variable `Pl` in Line 34

added to the model slice. (3) The **`forall`** rule in Line 33 is included in the slice as it defines the variable `Pl`.

(4) On the other hand, the token removal operation (Line 31) does not affect the slicing criterion as `Pl` is a different variable (redefined locally), `T` is passed as input parameter, while no model elements touched by this GT rule are dependent from those required at the slicing criterion.

(5) Although the **`if`** condition in Line 28 does not define variables that are used later in the slice, it has to be added because the execution of the **`forall`** rule added to the slice depends on the evaluated condition.

(6) Finally, as the slice includes statements that use the variable `T`, its definition as an incoming parameter of the `fireTransition` rule is added to slice.

The calculated program slice is represented by underlined source statements in Figure 2: the pattern `targetPlace` and parts of the `fireTransition` rules are included.

As model elements are created and deleted during the execution of the transformation, the corresponding model slice can contain elements from multiple states. To display the slice, Fig. 1d shows the final model state of the Petri net by adding the deleted token from the place `Client` as a cross. In this figure the model slices are included inside the dashed rectangles: the transition `Query`, the places `Client` and `Server`, the arcs between the included transition and place, and the token in `Server`.

IV. Conclusion

In the paper, we proposed a dynamic backward slicing approach for model transformation programs. Compared to slicing traditional programs, MT slices need to simultaneously include the program statements as well as the model elements which affect the slicing criterion (which may also include a dedicated model element in addition to a program location and a variable). The calculation of MT slices relies upon an execution trace, which is typically persisted by development and transformation frameworks anyhow, thus our approach could be easily adapted to other MT frameworks and languages.

In the future, we plan to investigate slicing challenges for MT programs as presented in Sec. II, addressing dynamic forward slicing, and static (forward and backward) slicing. Additional future research may investigate how relevant breakpoints can be automatically inserted during MT debugging where slicing techniques can also be exploited.

Acknowledgment

This work was partially supported by the SecureChange (ICT-FET-231101) and the CERTIMOT (ERC_HU-09-1-2010-0003) projects and János Bolyai Scholarship.

The authors would like to thank to Gábor Bergmann and István Ráth for their valueable insights.

References

[1] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages 3(3)*, pp. 121–189, 1995.

[2] G. Szilágyi, L. Harmath, and T. Gyimóthy, "The debug slicing of logic programs," *Acta Cybernetica*, vol. 15, no. 2, pp. 257–278, 2001.

[3] G. Szilágyi, T. Gyimóthy, and J. Małuszyński, "Static and dynamic slicing of constraint logic programs," *Automated Software Engineering*, vol. 9, pp. 41–65, 2002.

[4] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon, "Verification-driven slicing of UML/OCL models," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2010, pp. 185–194.

[5] K. Lano and S. Kolahdouz-Rahimi, "Slicing of UML models using model transformations," in *Model Driven Engineering Languages and Systems*, ser. LNCS, D. Petriu, N. Rouquette, and Ø. Haugen, Eds. Springer Berlin / Heidelberg, 2010, vol. 6395, pp. 228–242.

[6] T. Reps and T. Bricker, "Illustrating interference in interfering versions of programs," in *ACM SIGSOFT Software Engineering Notes*, ser. SCM '89. New York, NY, USA: ACM, 1989, p. 46–55, ACM ID: 73347.

[7] J. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, p. 37–61, Jan. 1985, ACM ID: 2366.

[8] D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 214–234, 2007.

[9] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró, "A benchmark evaluation of incremental pattern matching in graph transformation," in *Proc. 4th International Conference on Graph Transformations, ICGT 2008*, 2008, pp. 396–410.