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
{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 which 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. We evaluate our approach using different transformation case studies.**

***Keywords*-Program slicing; Model transformations**

I. Introduction

Model-driven design (MDD) aims to simultaneously improve quality and productivity by providing early model validation and automating various phases of software development including source code, test case or configuration generation. Model transformations (MT) play a central role in automating such tasks. Model-to-model transformations take one (or more) source model(s) as input and derive one (or more) target model(s) as output typically together with detailed traceability links. In-place model transformations operate on the same model instance to provide model simulation or refactoring.

Model transformations are captured in the form of a MT program, which can be taken as a regular piece of software. However, elementary steps in MT programs are captured by data-driven declarative rules, while complex transformations are assembled from elementary steps using imperative control structures. Due to this hybrid nature of MT languages, the direct adaptation of existing software engineering results is problematic, especially, when designing complex MTs where debugging and validation plays a crucial role.

Many integrated development environments (IDEs) include sophisticated program slicing techniques to calculate control and data dependencies between the statements of a program. When debugging MTs, transformation experts would require similar support to identify parts of the MT program which have causal dependence on a selected program statement (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. For instance, declarative model queries issued by transformation rules might introduce data dependencies that can only be detected by creating relevant slices of the transformed models as well.

In a previous short paper [1], we introduced the concepts of MT slicing for the first time (up to our best knowledge), and identified the main scientific challenges. We argued that the adaptation of existing program slicing techniques turns out to be non-trivial as MT programs take models as an additional input. Therefore, MT slices 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 IDEs, which frequently save additional information when executing a MT in order to provide undo/redo support. Consequently, execution traces of a MT run are readily available for MT slicing.

In [1], we informally sketched the idea of dynamic backward slicing of model transformations. In the current paper it is extended by formal definitions of MT slices and the slicing algorithm itself. An extensive experimental evaluation is also featured using various case studies taken from previous model transformation benchmarks.

The rest of the paper is structured as follows. At first dynamic backward MT slicing is demonstrated informally on an example in Sec. II, followed by the presentation of our slicing algorithm in Sec. III. An experimental evaluation of our slicing algorithm is given in Sec. IV. Related work is discussed in Sec. V, while Sec. VI concludes our paper.

II. Model Transformation Slicing: An Example

A traditional program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest [2]. In [1] we defined the slicing problem of MTs as illustrated in Fig. 1. The slicing algorithm receives three inputs: the *model transformation program*, the *models* on which the MT program operates and the *slicing criterion* denoting the point of interest, that is specified by a location (statement) in the MT program in

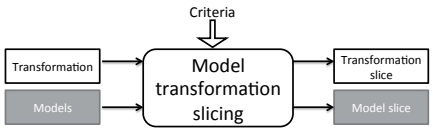


Figure 1: Slicing Problem of Model Transformations

combination with a subset of the MT program variables and elements of the model. As 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).

Of the various program slicing techniques we focus on dynamic and backward slicing in this paper. *Dynamic slicing* relies on a specific execution of the program. In case of MT slicing, the affected statements of the slicing criterion are calculated with respect to this specific execution, while model slices can be identified on its (input) models. A *backward slice* of a MT consists of (1) all statements and control predicates of the program and (2) all elements of the underlying model the slicing criterion depends on.

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 MT language of the VIATRA2 framework. The example is frequently used to demonstrate the usage of MTs in model simulation scenarios, and it already served as a performance benchmark for MTs [3]. Furthermore, 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.

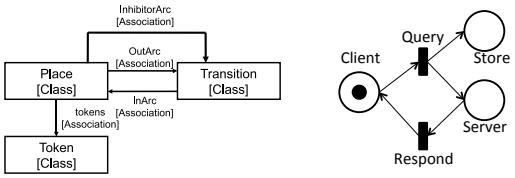
A. Running example: Simulation of Petri nets

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. 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* cannot fire. The structure of the modeling language of Petri nets is formalized by a corresponding metamodel in Fig. 2a.

Example 1: Fig. 2b depicts a simple Petri net. The net consists of three *places*, representing a Client, a Server and a Store and two *transitions*. The Client issues a query (the Query transition fires), the query is saved in the store (a token is created), the Server is activated (a token is created), and the Client waits for a response (its token is removed).

B. The VIATRA2 transformation language

1) Graph patterns: Graph patterns are often considered as atomic units of MTs [4]. They represent complex structural conditions (or constraints) that are needed to be fulfilled by a part of the (input) models. Graph patterns are also used to declaratively define model manipulation steps.



(a) Metamodel for Petri nets (b) A Simple Petri net

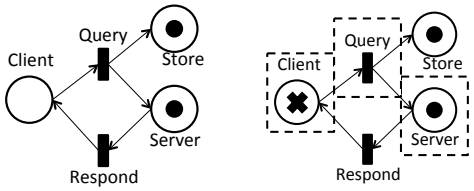
Figure 2: A Petri net Example

```
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 }
```

Lst. 1: Petri net Firing in VIATRA2 with a Calculated Slice

Example 2: We present two graph patterns from Lst. 1: `sourcePlace` (Line 1) identifies the source places of a transition, while `targetPlace` (Line 6) identifies the target places of a transition. Both patterns contain a `Transition` node `Tr` and a `Place` node `Pl` that are connected by an edge of type of `OutArc` and `InArc`, respectively. It is important to note that the variables `OutArc` and `InArc` are internal pattern variables which are not available outside the pattern.

2) Graph transformation: Graph transformation (GT) [5] 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



(a) The Model After the Firing of the Transition

(b) The Calculated Model Slices

Figure 3: The Dynamic Slice of the Firing of the Query Transition

(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 3: Lst. 1 uses 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` rule (Line 19) consists of a single *place*, while its RHS extends it with a *token*. This means, applying the rule creates a token and connects it to the *place*. The `removeToken` rule (Line 23) is expressed by swapping the same patterns: the token from LHS is removed in accordance with the RHS.

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 machine (ASM) [6] 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`**), rule invocation (**`call`**), variable declarations and updates (**`let`** and **`update`** constructs), **`if`**–**`then`**–**`else`** structures, and GT rule application on a single match (**`choose`**) or all possible matches (**`forall`**).

Example 4: The `fireTransition` rule (Line 27) describes the firing of a Petri net transition in VIATRA2. At first it is determined whether the input parameter is fireable 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 slicing problem for model transformations, we consider the execution of the rule `fireTransition` called with the transition `Query` as a parameter. The program slice is depicted in Lst. 1, while the model slice in Fig. 3.

Fig. 3a displays the model after the firing: the token from the place `Client` is removed, while tokens are added to the places `Store` and `Server`. As *slicing criteria*, we selected the GT rule application in Line 34 and the variable `Pl`.

During the execution of this firing, it is possible to obtain an execution trace, that records for each program statement the variable and model element accesses (both read and write). We can calculate the backward slices for the criteria by traversing the execution trace backwards. For each record of the trace, we determine whether the criteria depends on it either directly or indirectly 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 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 used later in the slice, it has to be added as one of its contained **`forall`** rule is added to the slice.

(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 model slices of the MT program might reference elements created or deleted during the execution of the transformation. E.g., tokens in the places `Store` and `Server` were created, while the one in the place `Client` was removed by the transformation run. To illustrate the model slice, we added both the created and deleted elements to Fig. 3b, and marked the elements in the slice. The crossed token (in the place `Client`) is deleted. Elements of the slice are contained within dashed rectangles - namely, transition `Query`, place `Server` and the token in the place `Server`.

III. A Dynamic Backward Slicing Algorithm for Model Transformations

In this section we formalize our approach by first defining execution traces for transformation programs (Sec. III-A), and show how to derive them during execution. Then we present an algorithm (Sec. III-B) that uses these traces to slice MT programs and models simultaneously.

A. Execution traces

1) Definitions: In advanced development and transformation environments, execution traces are often saved during transformation execution to support undo/redo or to provide traceability information between source and target models. This is achieved by storing the set of created, modified or removed model elements for each executed statement. To support slicing, this information has to be extended with the set of used transformation program statements and variables.

We define an *execution trace* as a sequence of trace records that represents the execution of the MT program.

Table I: Illustration of the Slicing Algorithm

(a) Execution Trace

(b) The Execution of the Algorithm

	Produces		Uses		Removes	*lookup*		*require*		*found*		slices	
	P_v	P_m	U_v	U_m	R_m	Var	ME	Var	ME	Var	ME	*translice*	*modelslice*
(0) **rule** fireTransition(**in** T)	$\{T\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{T\}$	$\{Ser, Qu, Cli, ToC\}$	$\emptyset$	$\emptyset$	$\{T\}$	$\emptyset$	$\{0,1,5,8\}$	$\{Ser, Qu, Cli, ToC\}$
(1) **if** (**find** isFireable(T))	$\emptyset$	$\emptyset$	$\{T\}$	$\{Cli, ToC, Qu\}$	$\emptyset$	$\{T\}$	$\{Ser, Qu\}$	$\{T\}$	$\{Cli, ToC, Qu\}$	$\emptyset$	$\emptyset$	$\{1,5,8\}$	$\{Ser, Qu, Cli, ToC\}$
(2) **forall** Pl	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{T\}$	$\{Ser, Qu\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{5,8\}$	$\{Ser, Qu\}$
(3) **find** sourcePlace(T,Pl)	$\{Pl\}$	$\emptyset$	$\{T\}$	$\{Cli, Qu\}$	$\emptyset$	$\{T\}$	$\{Ser, Qu\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{5,8\}$	$\{Ser, Qu\}$
(4) **apply** removeToken(Pl)	$\emptyset$	$\emptyset$	$\{Pl\}$	$\{Cli\}$	$\{ToC\}$	$\{T\}$	$\{Ser, Qu\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{5,8\}$	$\{Ser, Qu\}$
(5) **forall** Pl	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{T\}$	$\{Ser, Qu\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{5,8\}$	$\{Ser, Qu\}$
(6) **find** targetPlace(T,Pl)	$\{Pl\}$	$\emptyset$	$\{T\}$	$\{Sto, Qu\}$	$\emptyset$	$\{T\}$	$\{Ser, Qu\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{8\}$	$\{Ser, Qu\}$
(7) **apply** addToken(Pl)	$\emptyset$	$\{ToSt\}$	$\{Pl\}$	$\{Sto\}$	$\emptyset$	$\{T\}$	$\{Ser, Qu\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{8\}$	$\{Ser, Qu\}$
(8) **find** targetPlace(T,Pl)	$\{Pl\}$	$\emptyset$	$\{T\}$	$\{Ser, Qu\}$	$\emptyset$	$\{Pl\}$	$\emptyset$	$\{T\}$	$\{Ser, Qu\}$	$\{Pl\}$	$\emptyset$	$\{7\}$	$\{Ser, Qu\}$
(9) **apply** addToken(Pl)	$\emptyset$	$\{ToSe\}$	$\{Pl\}$	$\{Ser\}$	$\emptyset$	$\{Pl\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Loops and conditions are handled by referencing the same program statement either several times or none at all.

Definition 1: Let M be the set of model elements, S the set of program statements and V the set of program variables of a model transformation program. A *trace record* is a tuple $TR = (s, sub, P_m, P_v, U_m, U_v, R_m)$, where

- $s \in S$ is a program statement
- $sub \in S$ references the substatements of s, i.e. the set of program statements executed while s is being executed,
- $P_v \subseteq V$ denote the variables produced (created or updated) by the statement s
- $U_v \subseteq V$ are the variables used by s
- $P_m \subseteq M$ are the model elements produced by s
- $U_m \subseteq M$ are the model elements used (read) by s
- $R_m \subseteq M$ are the model elements removed by s

A program statement *uses* a variable or model element, if its execution depends on (the value of) the selected variable or model element. Similarly, a statement *produces* a variable or model element, if its value is created or changed during the execution of the statement. However, if a model element is *removed* during the execution, it is added to the removed set. The reason for this distinction is twofold: (1) the removal of the model element implies that the model element is already created, and (2) if at a later point a removed model element is tried to be accessed, it indicates a transformation error to be reported to the developer.

Such trace records can be built from the executed statements of the transformation program. The records of imperative language elements (e.g. ASM rules) can be derived using similar logic to traditional program slicing methods, while the trace records of declarative rules (e.g. GT rules) are derived based on its execution semantics.

In the following we describe the derivation from VIATRA2-specific declarative graph patterns and transformation rules. We believe the execution of specific language constructs of other transformation languages, such as OCL constraints can also be mapped to trace records similarly.

Example 5: To illustrate the use of execution traces, we present in Table Ia the partial execution trace of the Petri net simulator program considering the execution of the rule `fireTransition` with the parameter of the transition `Query`. The first column displays the corresponding program statement, an identifier of the record. Substatements are presented with indentation - e.g., the **`find`** `sourcePlace(T,Pl)` statement is a substatement of the **`forall`** `Pl` statement in row (2). The remaining columns display the produced, used and removed variables and model elements respectively. For readability, the associations between the model elements (e.g. of type *InArc*, *OutArc* and *tokens*) are omitted.

2) Trace record of graph pattern calls: A graph pattern is called with a set of program variables as parameters. These variables either reference a model element (input or *bound parameter*) or are undefined (output or *unbound parameter*). In case of successful matching, all variables will reference a model element: bound parameters remain unchanged, while values are assigned to unbound parameters.

Additionally, the matched model elements have to be added to the trace record as used model elements. Moreover, as the model remain unchanged, both the set of produced and removed model elements are empty. It is important to note that all model elements have to be added, not only the ones available as pattern parameter.

Example 6: Line (2) of Table Ia displays the trace record of the call of the `sourcePlace` pattern. The variable `T` is bound, while `Pl` is unbound, so $P_v = \{Pl\}$ and $U_v = \{T\}$. The pattern is matched to the following model elements: $Tr \leftarrow Query$, $Pl \leftarrow Client$ and $OutArc \leftarrow OutArc(Client, Query)$, so $U_m = \{Query, Client, OutArc(Client, Query)\}$.

3) Execution trace of GT rules: Graph transformation rules can be considered as pattern matching followed by a set of model manipulation rules (typically model element creation or removal). So the trace record is constructed by first creating the trace record of the precondition pattern of the rule (as in case of graph patterns), then adding the

```
Require: ∀i ∈ 1..n : TR_i                          ▷ Ordered list of trace records
  let lookup ← Criteria
  let translice ← ∅
  let modelslice ← ∅
  for i ← n, 1 do                                  ▷ Iterating over trace records backward
    TR_i = (s, sub, P_m, P_v, U_m, U_v, R_m)
    if (P_v ∪ P_m) ∩ lookup ≠ ∅ then               ▷ Data dependency
      let require ← U_v ∪ U_m ∪ R_m
      let found ← P_v ∪ P_m
      let lookup ← (lookup − found) ∪ require
      translice ← translice ∪ {s}
      modelslice ← modelslice ∪ U_m ∪ P_m ∪ R_m
    else if R_m ∩ lookup ≠ ∅ then                 ▷ Use of removed model element
      ERROR
    else if sub ∩ translice ≠ ∅ then              ▷ Control dependency
      require ← U_v ∪ U_m ∪ R_m
      lookup ← lookup ∪ require                   ▷ Found is empty here
      translice ← translice ∪ {s}
      modelslice ← modelslice ∪ U_m ∪ P_m ∪ R_m
    end if
  end for
```

Lst. 2: The Slicing algorithm

modified elements to corresponding list (T_m or R_m).

If a model element is added both to the used and removed set, it is possible to remove it from the uses set, as the removed set already represents this information.

Example 7: Line (3) of Table Ia presents the trace record of the invocation of the GT rule `removeToken`. As precondition it calls the `placeWithToken` pattern resulting in the following output: $U_v = \{T, Pl\}$, $P_v = \emptyset$ and $U_m = \{Client, TokenC, tokens(Client, TokenC)\}$.

As the postcondition pattern prescribes to remove the token (and the related `tokens` edge), two remove operations will be recorded in the trace. The corresponding sets of the trace record are as follows: $P_m = \emptyset$ and $R_m = \{TokenC, tokens(Client, TokenC)\}$. Finally, the model elements $TokenC$ and $tokens(Client, TokenC)$ can be removed from U_m, as they appear in the removed set R_m.

B. Generating backward slices from execution traces

We generate program slices by calculating dependencies between trace records. Our algorithm, presented in Lst. 2, is similar to the dynamic slicing algorithm in [7], with the addition of producing model slices as well.

The input of the algorithm is the ordered list of trace records (TR_i) and the variables and model elements from the slicing criteria. The algorithm assumes that the statement referenced in the last record is the statement of the criteria. This is not a real limitation, it is safe to omit the trace records after the slicing criteria when deriving a backward slice.

The algorithm maintains three sets during the execution. (1) Set `translice` stores the program statement, and (2) set `modelslice` represents the model elements already identified as parts of the slice. Finally, (3) set `lookup` stores the variables and model elements that are considered for direct data dependencies. The algorithm checks for every record:

1) If the current trace produces a variable or model element that is present in the set `lookup`, then the records has to be added because of *data dependency* (either between the record and the criteria, or between the record and an element already in the slice).
2) If the recorded statement *removes an element* that is present in the lookup, it suggests an error, because a model element is used that was previously removed.
3) If a substatement of the record is already added to the trace, the record has to be added because of *control dependency* (e.g. a pattern call inside a **`forall`** rule cannot be evaluated without the **`forall`** rule).

The `lookup` set is initialized to the variables and model elements in the slicing criteria, and it is updated each time a statement is added to the slice. More specifically, variables and model elements produced by the statement are removed, while elements used or removed by the statement are added. After every trace record is processed, the `lookup` set will store the variables and model elements the slicing criteria depends on, but which are not initialized or modified during the execution of the MT program. As a consequence, the variables in the final `lookup` set are the subset of the input variables of the transformation and the model elements are the subset of the input model the criteria depends on.

Example 8: We illustrate the slicing algorithm using the execution trace from Table Ia and variable `Pl` as slicing criteria. Table Ib displays (in a bottom-up way) the changes of the maintained sets. In line (9) the `lookup` starts with the criteria that is only referenced in the call of the GT rule `applyToken`, so neither set changes. The call of the pattern `targetPlace` (line (8)) produces the variable `Pl`, so it is removed from the set `lookup`. However, the referenced variable `T` and the model elements `Query`, `Server` and `Inarc(Server,Query)` are added to the set `lookup`.

The other iteration in the **`forall`** rule (line (6) and (7)) does not reference any element that is introduced in the `lookup` set, thus, it is not included in the slice. The **`forall`** rule (line (5)) produces no elements, but as it contains the already included pattern call, it is added to the slice. One can calculate similarly that the remaining three statements do not provide new dependencies for the slicing criteria.

The conditional rule in line (1) is added since its substatement **`forall`** rule from line (5) is already in the slice.

And finally, at line (0) we can detect that the model elements `Query`, `Client`, `Server` and `Inarc(Server,Query)` and `Outarc(Client, Query)` had to exist before executing the `fireTransition` rule. Similarly, the input variable `T` is also present in the slice that is defined by the caller of the rule. As a result, we obtain exactly those model elements and program statements highlighted in the example of Sec. II-C.

C. Implementation

The proposed dynamic backward slicing algorithm has been implemented within the VIATRA2 model transformation framework. For this purpose, we had to slightly extend the VIATRA2 interpreter to generate execution trace records.

Execution trace records are modelled (and optionally persisted) as EMF models in our implementation. This offers reusability by allowing other MT tools to provide such records from their execution by similar modifications of their interpreters. However, we believe in EMF-based approaches these modifications could be substituted by relying on advanced EMF modeling technologies.

Finally, the created execution trace model together with the slicing criterion selected by the transformation engineer is passed as an input to the slicing algorithm, and the calculated slices are displayed in the graphical user interface.

IV. Evaluation

The aim of the evaluation is to demonstrate that our MT slicing approach provides small, relevant slices describing both control and data dependencies from the selected slicing criteria with respect to the corresponding execution trace. To illustrate the simultaneous slicing of the models and MT programs, we selected four fundamentally different MT programs available in the VIATRA2 transformation framework, which were already used in the past for performance benchmark investigations (e.g. at various model transformation tool contests). These MT programs are used "as is", without any manual changes to them:

- Our running example, the *Petri net simulator* [3] transformation highlights typical domain specific language simulation. However, due to its low complexity we used it only as a stress test to our approach with various model sizes.
- The *AntWorld case study* [8], [9] of the GraBaTs Tool Contest 2008 is a larger model simulator program modeling the life of an ant colony with a continously increasing population (and world). The transformation features a more complex control structure, so we expect that both model and control dependencies will contribute to the calculated slices.
- The BPEL2SAL [10] case study developed as part of the SENSORIA FP6 EU project is a good example for typical *model-to-model* transformations with complex control structure and transformation rules. In this case, due to its large size and more imperative implementation style, we expect that the slicing of the transformation program will have a stronger correspondence with the control flow rather than the data flow.
- We also evaluated the *Reengineering case* of the Transformation Tool Contest 2011 [11], that extracts state machine models from a Java abstract syntax graph. The transformation is similar in nature to the BPEL2SAL case study, however the output model is created only from a small subset of the input metamodel.

We evaluated our approach with the case studies using multiple test cases, focusing on various aspects of MTs.

In Table II for each test case we measured the number of program statements (*statement coverage*), program variables (*variable coverage*) and model elements (*model coverage*). For statement coverage we display the number of statements present in the slice (column *'#'*), and their ratio to the number of statements available in the MT program (column *'Total'*) and in the trace (column *'Trace'*). The variable and model coverage values are to be interpreted similarly.

We expect that these coverage metrics will be good indicators of how the different slices correspond to the structure of the MT program and the input model.

During the evaluation of the test cases we noticed that, similar to most debugging techniques, trace generation slows down normal MT execution by $2-3$ times. However, even the longest traces were processed by the slicing algorithm in less then a minute, which we consider reasonable. A more detailed performance assessment is planned as future work.

A. The Petri net Simulator Case study

1) Test Cases: We generated Petri nets ranging from small (consisting 10 elements) to large (consisting 10000 elements), and executed firing sequences of $10-10000$ iterations each. As criteria we selected a token created in the final iteration. To manage the non-deterministic nature of the program, for every net and firing sequence size the simulator was executed ten times, and the result was averaged.

2) Lessons Learned: Table IIa shows the slice sizes of the simulator transformation. Because of the non-determinism of the transformation, firing only a few *transitions* in a large net the iterations will be independent from each other, making the case non-representative. We omitted such cases from the final results. Additionally, we removed the number of *tokens* from the model slice size to get more comparable results, as the number of tokens clearly dominated the model slice, especially in case of smaller net sizes.

These results were in line with our expectations for slicing simulation MTs:

- Program slices are largely model independent: the slices cover exactly the same program statements and variables. However, if the firings were independent from the criteria, then some statements and variables were not included in the result slice, although such cases are not presented in Table IIa.
- Model slices largely depend on both the number of firings and the overall size of the net: if the number of iterations exceeds the size of the net, most *places* and *transitions* are present in the trace and slice.

B. The AntWorld Case Study

1) Overview of the Transformation: The AntWorld case study describes a transformation to simulate the life of a simple ant colony searching for food to spawn more ants on a dynamically growing rectangular 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 easily found again by other ants.

Table II: Slice Sizes of Transformation Programs

(a) Slicing Results of the Petri net Simulator

	Firings	Statement Coverage			Variable Coverage			Model Coverage		
		#	Total	Trace	#	Total	Trace	#	Total	Trace
10	10	14±0	35,9%	50,0%	5±0	45,5%	62,5%	22,1±6,2	50,2%	74,7±18,6%
	100	14±0	35,9%	50,0%	5±0	45,5%	62,5%	43,9±0,3	99,8%	99,8±0,7%
	1000	14±0	35,9%	50,0%	5±0	45,5%	62,5%	44±0,0	100,0%	100,0±0,0%
	10000	14±0	35,9%	50,0%	5±0	45,5%	62,5%	44±0,0	100,0%	100,0±0,0%
100	100	14±0	35,9%	50,0%	5±0	45,5%	62,5%	195,7±32,8	52,0%	86,3±11,1%
	1000	14±0	35,9%	50,0%	5±0	45,5%	62,5%	371±5,8	98,7%	99,9±0,3%
	10000	14±0	35,9%	50,0%	5±0	45,5%	62,5%	376±0,0	100,0%	100,0±0,0%
1000	100	14±0	35,9%	50,0%	5±0	45,5%	62,5%	70,6±49,1	1,8%	19,5±11,9%
	1000	14±0	35,9%	50,0%	5±0	45,5%	62,5%	684,2±308,2	17,8%	38,1±17,4%
	10000	14±0	35,9%	50,0%	5±0	45,5%	62,5%	3239,8±305,4	84,1%	93,3±6,6%
10000	100	14±0	35,9%	50,0%	5±0	45,5%	62,5%	58,2±35,9	0,1%	16,3±8,0%
	1000	14±0	35,9%	50,0%	5±0	45,5%	62,5%	477,9±140,2	1,2%	19,4±5,0%
	10000	14±0	35,9%	50,0%	5±0	45,5%	62,5%	1211,9±1435,0	3,0%	16,8±19,4%

(b) Slicing Results of the BPEL2SAL Transformation

	Break	Statement Coverage			Variable Coverage			Model Coverage		
		#	Total	Trace	#	Total	Trace	#	Total	Trace
Hello, world	Break 1	93	1,6%	24,8%	8	0,3%	8,3%	16	0,3%	1,0%
	Break 2	183	3,2%	28,9%	24	0,8%	11,6%	56	1,2%	2,8%
	Break 3	199	3,4%	26,6%	22	0,7%	8,1%	48	1,0%	2,2%
	Break 4	415	7,1%	30,7%	127	4,2%	20,9%	237	5,0%	7,4%
	End	832	14,3%	44,2%	374	12,3%	41,5%	717	15,0%	15,9%
Credit process	Break 1	93	1,6%	24,8%	8	0,3%	8,3%	16	0,1%	1,0%
	Break 2	203	3,5%	30,2%	24	0,8%	10,6%	63	0,2%	1,9%
	Break 3	217	3,7%	27,3%	22	0,7%	7,5%	50	0,2%	1,1%
	Break 4	664	11,4%	42,3%	259	8,5%	35,8%	2509	8,6%	17,1%
	End	1181	20,3%	46,3%	577	18,9%	45,2%	5174	17,8%	20,0%

(c) Slicing Results of the Antworld Case Study

	Break	Statement Coverage			Variable Coverage			Model Coverage		
		#	Total	Trace	#	Total	Trace	#	Total	Trace
10 rounds	Grab	75,15±10,6	50,3±8,6%	50,4±6,6%	40,4±6,4	27,3±0,1%	53,8±7,9%	201,4±84,6	36,0±13,9%	36,0±13,9%
	Deposit	77,6±12,9	53,0±0,5%	53,0±7,1%	41,75±8,4	28,2±1,1%	56,1±9,6%	206,1±75,7	38,5±12,5%	38,5±12,5%
	Move	75,1±7,1	51,3±3,3%	51,3±3,8%	40,35±4,9	27,2±6,4%	54,4±4,9%	204,0±114,2	32,7±15,2%	32,7±15,2%
	Search	17,85±25,7	12,3±7,1%	12,4±16,9%	6,55±15,6	4,4±3,1%	8,9±22,5%	49,1±127,8	9,2±17,6%	9,2±17,6%
	Evaporate	82,1±5,3	54,6±7,9%	54,7±3,1%	43,4±3,6	29,3±2,4%	57,4±3,9%	161,3±100,0	29,9±12,1%	29,9±12,1%
	Consume	81,5±8,9	55,6±1,2%	55,6±3,7%	43,1±5,5	29,1±2,2%	58,2±5,2%	157,8±70,0	27,1±13,1%	27,1±13,1%
	Grow	45,55±0,7	29,1±1,5%	29,1±1,6%	19±0,0	12,8±3,8%	24,1±1,3%	376,8±80,1	47,8±8,9%	47,8±8,9%
50 rounds	Grab	90,8±0,4	52,0±1,9%	52,0±2,3%	49±0,0	33,1±0,8%	56,3±2,2%	857,3±226,2	52,2±6,2%	52,2±6,2%
	Deposit	90,75±0,6	52,3±0,5%	52,3±1,4%	49±0,0	33,1±0,8%	56,3±2,2%	896,3±349,7	58,5±1,7%	58,5±1,7%
	Move	90,85±0,5	51,9±8,9%	51,9±9,3%	49±0,0	33,1±0,8%	56,3±2,2%	812,8±269,4	52,9±7,2%	52,9±7,2%
	Search	44±39,0	25,2±1,1%	25,2±22,3%	21,4±23,1	14,4±6,2%	24,6±26,6%	284,6±374,1	19,5±25,8%	19,5±25,8%
	Evaporate	96,95±7,0	55,5±7,5%	55,6±4,1%	51,25±4,3	34,6±2,8%	58,9±4,9%	736,5±228,3	48,4±11,5%	48,4±11,5%
	Consume	95,9±7,8	54,9±4,1%	54,9±4,5%	50,6±4,9	34,1±8,9%	58,2±5,7%	729,1±203,0	49,2±6,4%	49,2±6,4%
	Grow	46±0,0	26,4±3,7%	26,4±3,7%	19±0,0	12,8±3,8%	21,8±3,9%	886,1±295,8	38,3±4,1%	38,3±4,1%

(d) Slicing Results of the Reengineering Case Study

	Break	Statement Coverage			Variable Coverage			Model Coverage		
		#	Total	Trace	#	Total	Trace	#	Total	Trace
	Break 1	17	10,6%	17,9%	6	4,6%	16,7%	21	0,1%	12,1%
	Break 2	33	20,5%	33,0%	18	13,8%	48,6%	65	0,4%	28,8%
	Break 3	35	21,7%	34,7%	18	13,8%	47,4%	102	0,7%	12,1%
	Break 4	39	24,2%	32,0%	19	14,6%	46,3%	139	0,9%	15,6%
	End	39	24,2%	31,7%	19	14,6%	46,3%	150	1,0%	16,8%

The simulation is executed in rounds, executing 7 different tasks each round. 4 tasks describe the behaviour of the ants: (1) searcher ants *grab food* from their current field, (2) ants at the hill *deposit the food* they were carrying, (3) carrier ants *move* closer to the hill and produce pheromones and (4) searcher ants *search* for food. After the ant tasks 3 world management tasks are executed: (5) pheromone *evaporates* from the fields, (6) the hill *consumes food* to produce more ants and finally (7) the *grid grows* when a searcher has crossed the boundary, and new food is produced.

The simulation starts with a very small model that is extended rapidly during the execution of the transformation.

2) Test Cases: We defined a breakpoint at the end of each task, that was triggered after 10 or 50 rounds. To handle the random nature of the simulation each case was executed 10 times, and the results were averaged. Although the tasks are implemented separately, we expect large slices because of various data and model dependencies between the tasks.

3) Lessons learned: Table IIc describes the size of the result slices. Based on the results we concluded that

- When comparing the Statement coverage results of the 10 round and 50 round simulations, we found that both slice and trace sizes are similar when considering the same breakpoints. This means, in 10 rounds the commonly used program statements are covered, and longer executions increase coverage slightly.
- However, the number of variables used in the trace increases with a longer execution - the same percentage of variables from the trace is present in the slice, the trace itself uses a larger set of available variables.
- The different executions of the simulation resulted in slices of similar size (small standard deviation), except the searcher test cases. In that case sometimes the slices are very small (a few statements and model elements), while in other cases they are an order of magnitude larger (large standard deviation).
- In this case there is no difference between the total and effective model size, as the transformation traverses the small, predefined model and extends it during runtime.

C. The BPEL2SAL Transformation

1) Overview of the Transformation: The BPEL2SAL transformation program [10] is a complex MT that transforms business process models captured in BPEL into the SAL model checker to analyze error propagation scenarios.

The transformation is one of the most complex ones created in VIATRA2: it consists of 102 ASM rules, 177 graph patterns and is over 8000 lines of code. It was implemented more like a traditional imperative program and thus does not use GT rules. This results in some really complicated ASM rules (over 100 lines of code) with complex control dependencies. However, as the control flow is less data-oriented, data dependencies are easier to calculate.

2) Test Cases: For the BPEL2SAL test cases, we focused on the analysis of the MT program itself using two different

processes, where the *Hello World* model represents the most simple BPEL model with a single process together with one start and one end nodes, while the *Credit process* [12] is an industrial example describing a complex credit evaluation process consisting of 220 BPEL model elements.

The BPEL2SAL transformation can be clearly separated into 4 phases, which deal with creating the parts of the generated SAL transition system description. The first phase (1) produces *type definitions*, followed by the (2) generation of *variable declarations*. Then (3) the variables are initialized, and (4) the SAL *transitions* are created. The first three phases are of roughly the same size, while the fourth is significantly more complicated and longer.

To create slices of this MT, we set up a breakpoint after each phase (and an additional breakpoint in the middle of phase (4) to split its complexity). In each case, we selected the last created model element and corresponding creation statement as the slicing criteria. Our aim with this setup is to get an overview on the distribution of complexity over the different phases of the transformation.

3) Lessons Learned: We expected that slices contain only a few variables and model elements, as the various phases of the transformation are well separated: although a large number of model elements are created in each phase, only a limited number of them are reused from earlier phases (e.g. identifiers referencing type names, etc.).

Table IIb displays the results of the measurements. Based on these results we conclude that

- As the execution progresses between the phases, the slices get typically larger (with the notable exception of the third breakpoint). Compared to the size of the trace, we found that around 25–45% of the statements are in the slice, a similar, but smaller 8–45% of variables, and only 15–20% model elements. This was in line with our expectations that only a limited number of variables are passed between phases.
- However, total statement coverage is small (under 20%), as various parts of the MT program are only executed if certain BPEL constructs are used in the input (e.g. compensation handling).
- The variable coverage behaves similarly as the statement coverage due to the fact that variables of non-executed statements are also not part of the trace.
- Finally, the model coverage results show that the transformation progresses the effective coverage converges to the total coverage metrics, meaning that the transformation covers the complete input model as expected.

D. The Reengineering Case Study

1) Overview of the Transformation: The objective of the Reengineering case study [11] is the extraction of a simple state machine with actions and triggers from an abstract syntax graph of a Java program. The transformation itself is much simpler and smaller then the BPEL2SAL case, and uses a similar imperative style with graph patterns. The main difference is between the two transformations is the handling of their input: the BPEL2SAL transformation reads its entire input, and performs complex transformation steps, while the Reengineering case operates only on a small subset.

The transformation can be broken down into two phases: the first phase creates every state, transition, trigger and action, then the second phase links the states and transitions.

2) Test Cases: To evaluate the slicing of the Reengineering transformation, we used the input model defined with the transformation in [11]. We evaluated 4 breakpoints: (1) during transition creation, (2) after trigger creation, (3) after action creation and (4) during linking.

3) Lessons Learned: As the various tasks of the transformation are interdependent, we expect a large number of variables present in the slices, as any previously variable or model element may be reused.

Table IId presents the resulting slice sizes calculated from the breakpoints. Based on these results we noticed that

- Slices use about one third of the elements from the trace, and almost half of the variables. This suggests many data dependencies between phases.
- However, Total model coverage is very low: as by the specification the goal is to extract information from the model, a large part of the input model is irrelevant and is present neither in the slice nor the trace.

E. Summary

Our measurements show that our MT slicing approach is able to provide small and understandable slices that encapsulates both control and data (model) dependency information simultaneously. In addition to that, by the analysis of the measurements we concluded that

- In case of complex, imperative control structures, MT slices are dominantly created from the dependency between the statements of the transformation programs, thus traditional imperative program slicing techniques are applicable and provide appropriate results.
- In case of declarative transformation rules slices are primarily created based on model dependencies.

In complex MTs declarative and imperative rules are often combined, where our combined approach is the most useful.

V. Related Work

Traditional program slicing techniques have been regularly and exhaustively surveyed in the past in papers like [2], [13]. The current paper does not enhance the theory of program slicing with new core algorithms, rather it innovatively applies a combination of existing techniques to solve a slicing problem in a new domain, in the field of MT programs. Anyhow, the main difference with respect to traditional program slicing is that a MT slice simultaneously identifies program variables *and* model elements affecting a specific location of interest of our MT program.

A. Slicing of Declarative Programs

In the context of program slicing, the closest related work addresses the slicing of logic programs (as the declarative pattern and rule language of VIATRA2 shares certain similarities with logic programs). Forward slicing of Prolog programs are discussed in [14] based on partial evaluation, while the static and dynamic slicing of logic programs in [15] uses a similar slicing algorithm as ours. Our work was also influenced by [16] which augments the data flow analysis with control-flow dependences in order to identify the source of a bug included in a logic program. This approach was later extended in [17] to the slicing of constraint logic programs (with fixed domains). Our conceptual extension to existing slicing techniques for logic programs is the incorporation of model slices into the slices.

Program slicing for the declarative language of the Alloy Analyzer was proposed in [18] as a novel optimization strategy to improve the verification of Alloy specifications.

B. Model slicing

Model slicing [19] techniques have already been successfully applied in the context of MDD. Slicing was proposed for model reduction purposes in [20], [21] to make the following automated verification phase more efficient.

In the context of UML models, Lano et. al. [22] exploits both declarative elements (like pre- and postconditions of methods) and imperative elements (state machines) when constructing UML model slices. Here the authors use model transformations for carrying out slicing of UML models. The slicing of finite state machines in a UML context was studied by Tratt [23], especially, to identify control dependence. A similar study was also executed for extended finite state machines in [24]. A dynamic slicing technique for UML architectural models is introduced in [25] using model dependence graphs to compute dynamic slices based on the structural and behavioral (interactions only) UML models.

Metamodel pruning [26] can also be interpreted as a specific slicing problem in the context of MDD where the effective metamodel is aimed to be automatically derived, which represents a specific view. Moreover, model slicing is used in [27] to modularize the UML metamodel into a set of small metamodels for each UML diagram type.

Various model slicing techniques are successfully merged by Blouin et al. [28] into a single, generative framework, using different slice types for different models.

Still, none of the existing model slicing approaches in the context of MDD address the slicing of MT programs, which is the main contribution of our current work.

C. Model transformation debugging

Basic debugging support is provided in many model transformation tools including ATL, GReAT, VIATRA2, FUJABA, Tefkat, and many more. The authors of [29] propose a dynamic tainting technique for debugging failures of model transformations, and propose automated techniques to repair input model faults [30]. On the contrary, our assumption is that the transformation itself is faulty, and not the input model. Colored Petri nets are used for underlying formal support for debugging transformations in [31]. The debugging of triple graph grammar transformations is discussed in [32], which envisions the future use of slicing techniques in the context of model transformations.

A forensic debugging approach of model transformations introduced in [33] by using the trace information of model transformation executions in order to determine the interconnections of source and target elements with transformation logic. It is probably closest to our current work as it proposes that MT slicing can be used for the selective re-execution of (parts of) the MT in a controlled runtime environment to gather knowledge about specific bugs, but no detailed discussion is provided how MT slicing could be carried out.

VI. 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 model element in addition to a statement and a variable). The calculation of MT slices relies upon an execution trace, which is typically persisted by development and transformation frameworks, thus our approach could be easily adapted to other MT frameworks and languages. Our measurements have also indicated that dynamic slices can be efficiently computed for complex MT programs and large source models using the proposed rich trace information.

A future research area is the building of a debugger for MT programs that records traces during execution, and uses the presented approach to step backward to the last relevant executed statement for the selected criteria. We also plan to adapt our approach to other MT environments by creating trace records from the various language elements.

Additionally, in line with the overview of MT slicing problems in [1] we plan to address dynamic forward slicing, and static (forward and backward) slicing for the future.

ACKNOWLEDGMENT

This work was partially supported by the SecureChange European project, (ICT-FET-231101) the CERTIMOT project (ERC_HU-09-1-2010-0003), by the grant TÁMOP - 4.2.2.B-10/1–2010-0009 and János Bolyai Scholarship.

REFERENCES

[1] Z. Ujhelyi, A. Horváth, and D. Varró, "Towards dynamic backwards slicing of model transformations," in *Automated Software Engineering, 26th IEEE/ACM Int. Conf. on*, 2011.

[2] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages 3(3)*, 1995.

[3] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró, "A benchmark evaluation of incremental pattern matching in graph transformation," in *Proc. 4th Int. Conf. on Graph Transformations*, 2008.

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

[5] G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.

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

[7] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, 1988.

[8] A. Zündorf, "Antworld benchmark specification, GraBaTs 2008," 2008, available from http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2008performancecase.pdf.

[9] A. Horváth, G. Bergmann, I. Ráth, and D. Varró, "Experimental assessment of combining pattern matching strategies with VIATRA2," *Int. Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, 2010.

[10] L. Gönczy, Á. Hegedüs, and D. Varró, "Methodologies for Model-Driven Development and Deployment: an Overview," in *Rigourous Software Engineering for Service-Oriented Systems*. Springer, 2011.

[11] T. Horn, "Program understanding: A reengineering case for the transformation tool contest," in *TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland*. EPTCS, 2011.

[12] M. Alessandrini and D. Dost, "SENSORIA Deliverable D8.3.a: Finance case study: Requirements modelling and analysis of selected scenarios," S&N AG, Tech. Rep., 2007.

[13] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, 2005.

[14] M. Leuschel and G. Vidal, "Forward slicing by conjunctive partial deduction and argument filtering," in *Programming Languages and Systems*. Springer, 2005, LNCS 3444.

[15] W. Vasconcelos, "A flexible framework for dynamic and static slicing of logic programs," in *Practical Aspects of Declarative Languages*. Springer, 1998, LNCS.

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

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

[18] E. Uzuncaova and S. Khurshid, "Kato: A program slicing tool for declarative specifications," in *29th Int. Conf. on Software Engineering*. IEEE, 2007.

[19] H. Kagdi, J. I. Maletic, and A. Sutton, "Context-free slicing of UML class models," in *21st Int. Conf. on Software Maintenance ICSM05*. IEEE, 2005.

[20] I. Schaefer and A. Poetzsch-Heffter, "Slicing for model reduction in adaptive embedded systems development," in *Int. Workshop on Software engineering for adaptive and self-managing systems*. New York, USA: ACM, 2008.

[21] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon, "Verification-driven slicing of UML/OCL models," in *25th IEEE/ACM Int. Conf. on Automated Software Engineering*. ACM, 2010.

[22] K. Lano and S. Kolahdouz-Rahimi, "Slicing of UML models using model transformations," in *Model Driven Engineering Languages and Systems*. Springer, 2010, LNCS 6395.

[23] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, "Control dependence for extended finite state machines," in *Fundamental Approaches to Software Engineering, 12th Int. Conf., FASE 2009*. Springer, 2009, LNCS 5503.

[24] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, "Slicing of state-based models," *Software Maintenance, IEEE Int. Conf. on*, 2003.

[25] J. T. Lallchandani and R. Mall, "A dynamic slicing technique for UML architectural models," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, 2011.

[26] S. Sen, N. Moha, B. Baudry, and J. Jézéquel, "Meta-model pruning," in *Model Driven Engineering Languages and Systems*. Springer, 2009.

[27] J. H. Bae, K. Lee, and H. S. Chae, "Modularization of the UML metamodel using model slicing," in *3rd Int. Conf. on Information Technology: New Generations*. IEEE, 2008.

[28] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux, "Modeling model slicers," in *Model Driven Engineering Languages and Systems*. Springer, 2011, LNCS 6981.

[29] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha, "Debugging model-transformation failures using dynamic tainting," in *24th European conference on Object-oriented programming*. Springer, 2010.

[30] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha, "Automated support for repairing input-model faults," in *25th IEEE/ACM Int. Conf. on Automated Software Engineering*. ACM, 2010.

[31] J. Schoenboeck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Catch me if you can - debugging support for model transformations," in *Model Driven Engineering Languages and Systems, 13th Int. Conf.* Springer, 2010, LNCS 6002.

[32] M. Seifert and S. Katscher, "Debugging triple graph grammar-based model transformations," in *Fujaba Days*, 2008.

[33] M. Hibberd, M. Lawley, and K. Raymond, "Forensic debugging of model transformations," in *Model Driven Engineering Languages and Systems, 10th Int. Conf.* Springer, 2007, LNCS 4735.