Rete network slicing for model queries

Zoltán Ujhelyi[2], Gábor Bergmann[1], Dániel Varró[1]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
MTA-BME Lendület Research Group on Cyber-Physical Systems
1117 Budapest, Magyar tudósok krt. 2.
{bergmann,varro}@mit.bme.hu
[2] IncQuery Labs Ltd.
1113 Budapest, Bocskai út 77-79.
zoltan.ujhelyi@incquerylabs.com

Abstract. Declarative model queries captured by graph patterns are frequently used in model driven engineering tools for the validation of well-formedness constraint or the calculation of various model metrics. However, their high level nature might make it hard to understand all corner cases of complex queries. When debugging erroneous patterns, a common task is to identify which conditions or constraints of a query caused some model elements to appear in the results. Slicing techniques in traditional programming environments are used to calculate similar dependencies between program statements. Here, we introduce a slicing approach for model queries based on Rete networks, a cache structure applied for the incremental evaluation of model queries. The proposed method reuses the structural information encoded in the Rete networks to calculate and present a trace of operations resulting in some model elements to appear in the result set. The approach is illustrated on a running example of validating well-formedness over UML state machine models using graph patterns as a model query formalism.

Keywords: program slicing, model queries, graph patterns

1 Introduction

Modern industrial and open source modeling tools frequently rely upon various services built on top of incremental query evaluation techniques [1,2] for efficient revalidation of well-formedness constraints, recalculation of view models, re-execution of code generators or maintenance of traceability links [3,4]. EMF-IncQuery [3] is an open source Eclipse project which offers a declarative graph query language [5] for capturing queries and a scalable query engine for incremental query evaluation using the Rete algorithm [6].

Industrial domain-specific languages and tools (e.g. in the automotive, avionics or telecommunications domain) necessitate the development of large number of complex, interrelated queries, which turns out to be an error prone task in

industrial practice. Some constraints may accidentally be omitted, other constraints may be added to a query unintentionally, while patterns may be composed using a wrong order of parameters. While the EMF-IncQuery framework contains a type checker and various well-formedness constraints are also checked, such static checks still do not guarantee that query specifications are free of flaws, thus runtime debugging of queries need to be carried out in practice.

Unfortunately, the declarative nature of query languages makes debugging of query specifications a challenging task. The order of clauses in a query specification does not coincide with the actual evaluation order in case of local search based query evaluation [7,8]. Furthermore, incremental evaluation techniques further complicate the issue as all matches of all queries (and subqueries) are readily available immediately at any time.

Model transformation slicing was introduced in [9] as a technique to assist debugging of model transformations. As a conceptual difference with respect to traditional program slicing, a transformation slice includes not only the relevant instructions of the transformation program, but also those model elements that can influence the slicing criterion. A dynamic backward slicing approach was proposed for the transformation languages of VIATRA [9] and static backward slicing approach for ATL [10,11].

In the current paper, we propose a slicing technique for incremental graph patterns evaluated on top of Rete networks. Based upon an observed change in the match set of a graph pattern (e.g. an extra match or a missing match) we traverse the nodes of the Rete network in a bottom-up way to identify those tuples in other Rete nodes which may contribute to the observed aggregate change. We illustrate how this slicing information can be computed in the context of statechart models. Our slicing approach may assist the debugging of model queries by localizing suspicious spots in queries.

The rest of the paper is structured as follows. Sec. 2 gives a brief overview of graph patterns, and presents why slicing can help to debug incorrect pattern definitions. A formalization of incremental evaluation of model queries using Rete networks is provided in Sec. 3. The slicing approach itself is presented in Sec. 4 and is illustrated in the context of our running example. Related work is discussed in Sec. 5 while Sec. 6 concludes our paper.

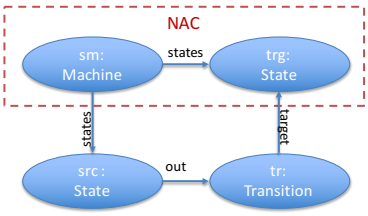
2 Motivating example and overview

*Graph pattern*s are a declarative, graph-like formalism representing a condition (or constraint) to be matched against instance model graphs. Graph patterns are used for various purposes in model-driven development, such as defining model transformation rules or defining general purpose model queries including model validation constraints in various advanced tools (such as eMOFLON, Henshin, EMF-IncQuery or VIATRA).

Informally, a graph pattern can be described as a set of *structural constraints* prescribing the interconnection between nodes and edges of given types. Furthermore, further constraint types, such as *pattern composition constraints* for

```
1  //'tr' represents a transition connecting two statemachines.
2  pattern DifferentStateMachine(tr : Transition) {
3      State.out(src, tr);
4      Transition.target(tr, trg);
5      find StateofMachine(sm, src);
6      neg find StateOfMachine(sm, trg);
7      //find StateOfMachine(sm, trg);
8  }
9
10 //Variable 'st' represents a State of the StateMachine 'sm'.
11 pattern StateOfMachine(sm : StateMachine, st : State) {
12     StateMachine.states(sm, st);
13 }
```

(a) Textual Notation



(b) Graph Notation

Fig. 1: Example graph patterns

the reuse of subpatterns, help the description of complex constraints. Finally, a *match* in a model M of a pattern is the binding of all variables to elements of M that satisfies all constraints expressed by the pattern. Efficient caching techniques based on Rete networks [6] enumerate all matches of a pattern and incrementally update the caches upon model changes.

Example 1 Fig. 1 describes a graph pattern using the textual syntax of EMF-IncQuery [3] that identifies transitions whose source and target states are in different states machines. It uses a subpattern called `StateOfMachine` (Line 11), connecting two variables of type `StateMachine` and `State` with the edge type of `StateMachine.states`. The main pattern `DifferentStateMachine` (Line 2) uses four variables to represent a `Transition`, a source and a target `State` and a `StateMachine`. The `Transition` and the two `State`s are connected with two edge constraints, while the states `fr` and `to` are connected to the statemachine by a positive (Line 5) meaning that variable `fr` has to be connected via the called pattern, and a negative pattern call (Line 6) which prevents `to` to be connected.

Fig. 1b depicts the same pattern using a graphical notation where nodes are entity constraints, edges are relational constraints, positive pattern calls are inlined (copied), and negative pattern call are marked by `NAC` areas.

During pattern development, engineers may accidentally make faults. For instance, imagine that the `neg` keyword is omitted from Line 6, and thus the definition of pattern `DifferentStateMachine` erroneously includes (the commented) Line 7 instead of Line 6. It results in a positive pattern call instead of a negative pattern call making the pattern to represent transitions where both source and target states are in the same state machine, thus completely replacing the correct match set of the pattern with incorrect matches.

During debugging of queries and transformations, when the developer identifies that the match set of a pattern is different from what was expected, he or she frequently wishes to learn what elementary model changes would result in the appearance of new match or the disappearance of an existing match of

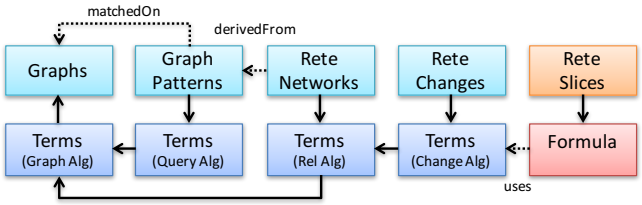


Fig. 2: Overview of formalization

a pattern. The current paper will present a formal slicing technique for Rete network based caches of graph patterns to answer such questions.

For that purpose, we present a chain of semantic mappings (see Fig. 2) by (1) starting from a Σ-term algebra to formalize graphs and then (2) (a subset of) the graph pattern language of EMF-IncQuery. (3) A relational algebraic treatment is provided for Rete networks to cache matches of patterns and (4) changes in the match sets are then handled by a change algebra. Finally, (5) Rete slicing is defined as specific formulae over this change algebra. While the main innovation of the paper is related to this final step, we briefly present the entire chain to provide solid foundations.

3 Graph patterns and Rete networks

We present an algebraic formalization of incremental graph pattern matching with Rete networks following the definitions of [12].

3.1 Graphs and graph patterns

Since Rete networks can be adapted to various graph formalisms, we omit the handling of types and use directed labeled and attributed graphs to represent models for the sake of generality and simplicity.

Definition 1 (Directed labeled attributed graph) A *directed labeled and attributed graph* $M = \langle V_M, E_M, L_M, D_M, \mathrm{src}_M, \mathrm{trg}_M, \mathrm{lbl}_M, \mathrm{attr}_M \rangle$ is a tuple, where V_M and E_M denote nodes and edges of the graph, respectively. L_M is a set of labels, while D_M represents a set of data nodes. The nodes, edges and data nodes represent the universe of the graph model $\mathbb{U}_M = V_M \cup E_M \cup D_M$.

Functions src_M and trg_M map edges to their source and target nodes, formally $\mathrm{src}_M : E_M \mapsto V_M$ and $\mathrm{trg}_M : E_M \mapsto V_M$. The labeling function lbl assigns labels to edges, formally $\mathrm{lbl}_M : (V_M \cup E_M) \mapsto L_M$, and the attribute function maps nodes to corresponding attribute values, formally $\mathrm{attr}_M : V_M \mapsto D_M$. We may omit subscript M when graph M is unambiguous. □

Graphs will serve as the core underlying semantic domain to evaluate graph patterns but we define an algebraic term representation (in the style of abstract

state machines [13]) for a unified treatment of formalization. For that purpose, we rely on some core definitions of terms, substitution, interpretation and formulas.

Definition 2 (Vocabulary and terms) A *vocabulary* Σ is a finite collection of function names. Each function name f has an *arity*, a non-negative integer, which is the number of arguments the function takes. Nullary function names are often called *constants*.

The **terms** of Σ are syntactic expressions generated inductively was follows: (1) Variables $v_0, v_1, v_2, \ldots$ are terms; (2) constants c of Σ are terms; (3) if function f is an n-ary function name and $t_1, \ldots, t_n$ are terms, $f\langle t_1, \ldots t_n\rangle$ is a term.

Since terms are syntactic objects, they do not have a meaning. A term can be evaluated, if elements of the model are assigned to the variables of the term.

Definition 3 (Substitution) Given a directed attribute graph model M, a **substitution** for M is a function s which assigns an element $s(v_i) \in \mathbb{U}_M$ to each variable v_i. A **partial substitution** assigns a value to only certain variables v_i.□

Definition 4 (Interpretation of terms) By induction on the length of term t, given a substitution s, a value $[\![t]\!]^M_s \in \mathbb{U}_M$ (the **interpretation of term** t in model M) is defined as follows:

1. $[\![v_i]\!]^M_s := s(v_i)$ (interpretation of variables);
2. $[\![c]\!]^M_s := c^M$ (interpretation of constants);
3. $[\![f\langle t_1, \ldots, t_n\rangle]\!]^M_s := f^M\langle [\![t_1]\!]^M_s, \ldots, [\![t_n]\!]^M_s\rangle$ (interpretation of functions).

A **ground term** is a term with a (full) substitution of variables. □

Definition 5 (Formulas) Formulas of Σ are generated inductively as follows:

1. Equality (and inequality) of two terms $t_1 = t_2$ is a formula f.
2. If f_1 and f_2 are formulas then $f_1 \wedge f_2$, $f_1 \vee f_2$ are formulas.

A simplified notation is used for predicates (i.e. boolean terms) which may omit $= \top$ and $= \bot$ from equality formulas. □

We first define algebraic terms to represent graph patterns which are evaluated over directed labeled attributed graphs as semantic models. A *match* of a pattern is a binding of all variables to model elements or attribute values that fulfill all the constraints of the graph pattern.

Definition 6 (Graph pattern and match set) A *graph pattern* P is a term over a special vocabulary Σ with function symbols for constraints including *structural constraints* (entity, relation), *equality checks*, *pattern definitions* with a disjunction of *pattern bodies* containing conjunction of constraints and *positive and negative pattern calls* and constants (representing model element identifiers and data values). The semantics of P is defined as an interpretation of the term over a graph M and along a substitution s as detailed in Table 1 for the key

Name	Interpretation
Entity	$[\![ent\langle l, v\rangle]\!]^M_s = \top$, if $\begin{cases} \text{lbl}([\![v]\!]^M_{s'}) = l, \text{where} \\ s' \supseteq s \wedge v \in \text{dom}(s') \end{cases}$
Relation	$[\![rel\langle l, v, v_s, v_t\rangle]\!]^M_s = \top$, if $\begin{cases} \text{lbl}([\![v]\!]^M_{s'}) = l \wedge \\ \text{src}([\![v]\!]^M_{s'}) = [\![v_s]\!]^M_{s'} \wedge \\ \text{trg}([\![v]\!]^M_{s'}) = [\![v_t]\!]^M_{s'}, \text{where} \\ s' \supseteq s \wedge \{v, v_s, v_t\} \subseteq \text{dom}(s') \end{cases}$
Equality check	$[\![eq\langle v_1, v_2\rangle]\!]^M_s = \top$, if $\begin{cases} [\![v_1]\!]^M_{s'} = [\![v_2]\!]^M_{s'}, \text{where} \\ s' \supseteq s \wedge \{v_1, v_2\} \subseteq \text{dom}(s') \end{cases}$
Inequality check	$[\![neq\langle v_1, v_2\rangle]\!]^M_s = \top$, if $\begin{cases} [\![v_1]\!]^M_{s'} \neq [\![v_2]\!]^M_{s'}, \text{where} \\ s' \supseteq s \wedge \{v_1, v_2\} \subseteq \text{dom}(s') \end{cases}$
Pattern Body	$[\![PB\langle v_1, \dots v_k\rangle \leftarrow c_1 \wedge \dots c_n]\!]^M_s = \top$, if $\begin{cases} \bigwedge_{i\in 1..n} [\![c_i]\!]^M_{s'} = \top, \text{where} \\ s' \supseteq s \wedge \{v_1, \dots v_k\} \subseteq \text{dom}(s') \end{cases}$
Graph Pattern	$[\![P\langle v_1, \dots v_k\rangle \leftarrow PB_1 \vee \dots PB_n]\!]^M_s = \top$, if $\begin{cases} \bigvee_{i\in 1..n} [\![PB_i]\!]^M_{s'} = \top, \text{where} \\ s' \supseteq s \wedge \{v_1, \dots v_k\} \subseteq \text{dom}(s') \end{cases}$
Positive Call	$[\![call(P^c\langle v_1, \dots v_n\rangle)]\!]^M_s = \top$, if $\begin{cases} [\![P^c\langle v^c_1, \dots v^c_n\rangle]\!]^M_{s'} = \top, \text{where} \\ \forall_{i\in 1..n} : s'(v^c_i) = s(v_i) \end{cases}$
Negative Call	$[\![neg(P^c\langle v_1, \dots v_n\rangle)]\!]^M_s = \top$, if $\begin{cases} [\![P^c\langle v^c_1, \dots v^c_n\rangle]\!]^M_{s'} = \bot, \text{where} \\ \forall_{i\in 1..n} : s'(v^c_i) = s(v_i) \end{cases}$

Table 1: Algebraic definition of graph patterns

	Different State Machines	State of Machine (SoM)
Variables	$src, trg, tr, sm, r_1, r_2$	$st, sm, r \in V^{rel}$
Constraints	$ent_1\langle State, src\rangle, ent_2\langle Transition, tr\rangle$ $ent_3\langle State, trg\rangle, ent_4\langle Machine, sm\rangle$ $rel_5\langle State.out, r_1, src, tr\rangle$ $rel_6\langle Transition.target, r_2, tr, trg\rangle$ $call_7(SoM\langle src, sm\rangle), neg_8(SoM\langle trg, sm\rangle)$	$ent_1\langle State, source\rangle$ $ent_2\langle Machine, sm\rangle$ $rel_3\langle Machine.States, r, sm, st\rangle$

Table 2: The Different State Machines pattern

elements of the EMF-IncQuery language [3]. For easier formulation, we use $\overline{V^k}$ as a shorthand to represent a vector of variables, formally $f\langle\overline{V^k}\rangle = f\langle t_1, \dots t_k\rangle$.

A *match* of P in M is a substitution s which satisfies all constraints. The *match set* is the set of all matches of a pattern in a graph model:

$$MS^P_M = \{s \mid [\![P\langle\overline{V^k}\rangle \leftarrow PB_1 \vee \dots PB_n]\!]^M_s = \top\}$$ □

Example 2 Table 2 provides the algebraic representation of graph pattern *Different State Machines* of Fig. 1 that identifies transitions which connect elements between different state machines.

Name	Interpretation
Entity/0	$[\![n^E\langle v\rangle]\!]^M = \{\langle v\rangle \mid [\![ent\langle l, v\rangle]\!]^M = \top\}$
Relation/0	$[\![n^R\langle v, v_s, v_t\rangle]\!]^M = \{\langle v, v_s, v_t\rangle \mid [\![rel\langle l, v, v_s, v_t\rangle]\!]^M = \top\}$
Projection/1	$[\![n^\pi\langle\overline{V^k}\rangle]\!]^M = \pi_{\overline{V^k}}[\![n^1\langle\overline{V^n}\rangle]\!]^M$, where $\overline{V^n} \supseteq \overline{V^k}$
Filter/1	$[\![n^\sigma\langle\overline{V^k}\rangle]\!]^M = \sigma[\![n^1\langle\overline{V^k}\rangle]\!]^M$
Join/2	$[\![n^\bowtie\langle\overline{V^k}\rangle]\!]^M = \begin{cases} [\![n^1\langle\overline{V^i}\rangle]\!]^M \bowtie [\![n^2\langle\overline{V^i}\rangle]\!]^M, \text{where} \\ \overline{V^k} = \overline{V^i} \cup \overline{V^j} \end{cases}$
Anti-join/2	$[\![n^\triangleright\langle\overline{V^k}\rangle]\!]^M = [\![n^1\langle\overline{V^k}\rangle]\!]^M \triangleright [\![n^2\langle\overline{V^j}\rangle]\!]^M$
Disjunction/2	$[\![n^\cup\langle\overline{V^k}\rangle]\!]^M = [\![n^1\langle\overline{V^k}\rangle]\!]^M \cup [\![n^2\langle\overline{V^k}\rangle]\!]^M$

Table 3: Relational algebraic operations of Rete networks

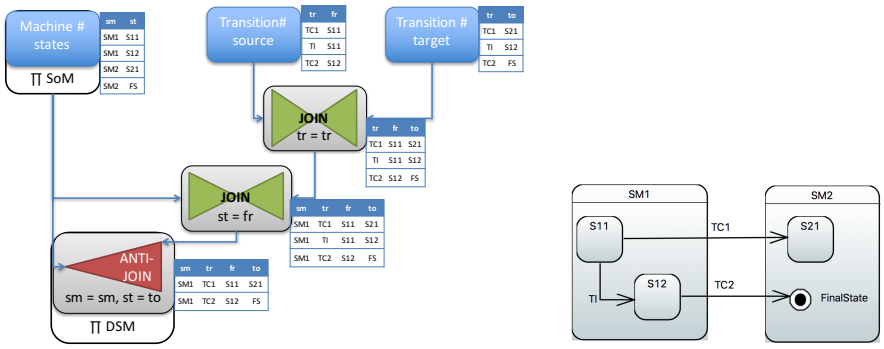
3.2 Graph pattern matching with Rete networks

The Rete algorithm [6] is a well-known and efficient technique of rule-based systems which has been adapted to several incremental pattern matchers [12,14,15]. The algorithm uses an incremental caching approach that indexes the basic model elements as well as *partial matches* of a graph pattern that enumerate all model element tuples which satisfy a subset of the graph pattern constraints. These caches are organized in a graph structure called Rete network supporting incremental updates upon model changes.

Definition 7 (Rete network) A *Rete network* is a directed acyclic graph $R \equiv \langle N, E, L, Term, \text{src}, \text{trg}, \text{lbl}, \text{attr}\rangle$, where N is a set of *Rete nodes* connected by edges E (along src and trg), $L = Kind \cup Index$ defines node kinds (entity E and relation input R, natural join $\bowtie$, filter σ, projection π, disjunction $\cup$ and anti-join $\triangleright$) as node labels and indices as edge labels (along lbl), while data associated to nodes are specific $Term$s of type $n^{op}\langle\overline{V^k}\rangle$.

Definition 8 (Memory of a Rete node) Each Rete node $n \in N$ of the Rete network R_P stores all matches of an instance model M which satisfy certain constraints which is denoted as $[\![n\langle\overline{V^k}\rangle]\!]^M$. Each Rete node n relies upon the memory of its parents $[\![n^i\langle\overline{V^k}\rangle]\!]^M$ to calculate its own content inductively by relational algebraic operators which are specified in details in Table 3.

The *memory of an input node* n^I lists entities and relations of the model with a specific label. Positive pattern calls are always mapped to join node, while negative pattern calls are expressed via anti-join nodes. A *production (output) node* in a Rete network contains all matches of a graph pattern P by expressing the complex constraints with a relational algebraic operations (e.g. projection, filter, join, anti-join, disjunction). The compilation of the graph pattern language of EMF-IncQuery into a corresponding Rete network is out of scope for the current paper and it is studied in [12] in details. We only rely on the correctness of a compilation $comp : P \mapsto N$ to guarantee that a match set of a graph pattern P (see Table 1) equals to the memory of the corresponding Rete node (as defined in Table 3), i.e. $MS_M^P = [\![n\langle\overline{V^k}\rangle]\!]^M$ where $n = comp(P)$.



(a) A Rete network (b) An instance model

Fig. 3: A Rete network for the Different State Machines pattern

Example 3 Fig. 3a depicts a Rete network for the `Different State Machines` pattern. Its input nodes cache three references: `states` of Machines; `out` references of `State`s and `target` references of `Transition`s. The first join node of the network connects the `source` and `target` states, while the second join node adds the container machines of the source patterns by joining the production node of the called pattern *State of Machine*. Finally, the anti-join node (depicted by the red triangle) ensures that the target state is not connected to the same state machine as the source node by filtering matches of its join parent node which also correspond to matches of the `states` node along the called pattern *State of Machine* (which is inlined during compilation).

We display the cached model elements of the instance model Fig. 3b in a table for each Rete node, describing two state machines with a few states and transitions (some of which cross the boundary of a statemachine). □

3.3 Incremental change-driven behavior of Rete networks

If the memory of a Rete nodes changes, the memory of all its children Rete nodes needs to be updated in accordance with the relational algebraic operation of the Rete node. For that purpose, we define a change algebra over terms n_+ and n_- (jointly denoted as n_Δ) which represent tuples added and removed from a Rete node n. We briefly revisit the semantics of such change terms in Table 4 while the reader is referred to [12] for a detailed discussion.

Definition 9 (Change algebra for Rete nodes) Let M be a graph model cached in a Rete network R and let Δ be a set of elementary model changes (represented by terms for creation and deletion of entities n^E_+, n^E_- or references n^E_+ and n^E_-) over this graph. We define a term n^{op}_Δ for each node n^{op} of the Rete network to represent matches that are changed by Δ with respect to M.

The semantics of such terms are inductively defined by using (i) match information n^{op} cached in R for M (i.e. the previous state of the model) and

Name	Interpretation
Entity	$[\![n^E_\Delta\langle v\rangle]\!]^{M,\Delta}=\begin{cases}([\![n^E\langle v\rangle]\!]^M=\top\wedge[\![n^E_-\langle v\rangle]\!]^\Delta=\top)\vee\\([\![n^E\langle v\rangle]\!]^M=\bot\wedge[\![n^E_+\langle v\rangle]\!]^\Delta=\top)\end{cases}$
Relation	$[\![n^R_\Delta\langle v,v_s,v_t\rangle]\!]^{M,\Delta}=\begin{cases}([\![n^R\langle v,v_s,v_t\rangle]\!]^M=\top\wedge[\![n^R_-\langle v,v_s,v_t\rangle]\!]^\Delta=\top)\vee\\([\![n^R\langle v,v_s,v_t\rangle]\!]^M=\bot\wedge[\![n^R_+\langle v,v_s,v_t\rangle]\!]^\Delta=\top)\end{cases}$
Projection	$[\![n^\pi_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}=\pi([\![n^1\langle\overline{V^n}\rangle]\!]^M\cup[\![n^1_\Delta\langle\overline{V^n}\rangle]\!]^{M,\Delta})\setminus\pi[\![n^1\langle\overline{V^n}\rangle]\!]^M$
Filter	$[\![n^\sigma_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}=\sigma[\![n^1_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}$
Join	$[\![n^{\bowtie}_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}=\begin{cases}([\![n_1\langle\overline{V^i}\rangle]\!]^M\bowtie[\![n^2_\Delta\langle\overline{V^j}\rangle]\!]^{M,\Delta}_n)\cup\\([\![n^1_\Delta\langle\overline{V^i}\rangle]\!]^{M,\Delta}\bowtie[\![n_2\langle\overline{V^j}\rangle]\!]^M)\cup\\([\![n^1_\Delta\langle\overline{V^i}\rangle]\!]^{M,\Delta}\bowtie[\![n^2_\Delta\langle\overline{V^j}\rangle]\!]^{M,\Delta})\end{cases}$
Anti-join	$[\![n^{\triangleright}_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}=\begin{cases}[\![n^1\langle\overline{V^k}\rangle]\!]^M\bowtie\\(\pi[\![n^2\langle\overline{V^j}\rangle]\!]^M\setminus\pi([\![n^2\langle\overline{V^j}\rangle]\!]^{M,\Delta}\cup[\![n^2_\Delta\langle\overline{V^j}\rangle]\!]^{M,\Delta}))\cup\\[\![n^1_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}\triangleright([\![n^2\langle\overline{V^j}\rangle]\!]^M\cup[\![n^2_\Delta\langle\overline{V^j}\rangle]\!]^{M,\Delta})\end{cases}$
Disjunction	$[\![n^{\cup}_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}=\begin{cases}\{[\![n^1_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}\mid([\![n^2\langle\overline{V^k}\rangle]\!]^M=\emptyset)\wedge([\![n^2_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}=\emptyset)\}\cup\\\{[\![n^2_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}\mid([\![n^1\langle\overline{V^k}\rangle]\!]^M=\emptyset)\wedge([\![n^1_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}=\emptyset)\}\cup\\\{[\![n^1_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}\mid[\![n^2_\Delta\langle\overline{V^k}\rangle]\!]^{M,\Delta}\}\end{cases}$

Table 4: Change algebra for Rete nodes

(ii) change already computed at parent nodes n^1_Δ and n^2_Δ of n^{op}_Δ split along operations *op* as detailed in Table 4. □

A brief informal explanation of these cases is as follows:

Entity and relation change A model entity appears in the change set n^E_Δ if (1) it exists in M and it is removed by Δ or (2) it does not exist in M and it is created by Δ (and same holds for model references).

Change in projection and filter nodes The change set of a projection node is defined as the difference of the new $n^1\langle\overline{V^n}\rangle\cup n^1_\Delta\langle\overline{V^n}\rangle$ and old $n^1\langle\overline{V^n}\rangle$ memory of the parent nodes. In case of a filter node the change set is the change set of its single parent $n^1_\Delta\langle\overline{V^k}\rangle$ filtered using the σ filter operator.

Change in join nodes The change set of a join node consists of the union of three change sets: (1) the join of the the memory of the first parent node $n^1\langle\overline{V^i}\rangle$ with the delta coming from the second parent $n^2_\Delta\langle\overline{V^j}\rangle$; (2) the join of the second parent $n^2\langle V_i\rangle$ with the first parent delta $n^1_\Delta\langle\overline{V^j}\rangle$; and (3) the join of the two parent deltas $n^1_\Delta\langle\overline{V^i}\rangle$ and $n^2_\Delta\langle\overline{V^j}\rangle$.

Change in anti-join nodes The change set of an anti-join node is the union of two sets: (1) the elements in the second parent delta $n^2_\Delta\langle\overline{V^j}\rangle$ that are filtering out pre-existing tuples from the first parent $n^1\langle\overline{V^k}\rangle$; and (2) the changed elements of the first parent $n^1_\Delta\langle\overline{V^k}\rangle$ that are not filtered out by the second parent or its changes.

Change in disjunction nodes The change set of a disjunction node is the union of three sets: (1) the delta of the first parent $n^1_\Delta\langle\overline{V^k}\rangle$ that was not present in the second parent $n^2\langle\overline{V^k}\rangle$; (2) the delta of the second parent $n^2_\Delta\langle\overline{V^k}\rangle$ that was not present in the first parent $n^1\langle\overline{V^k}\rangle$ and (3) elements that were added or removed by both parent changes.

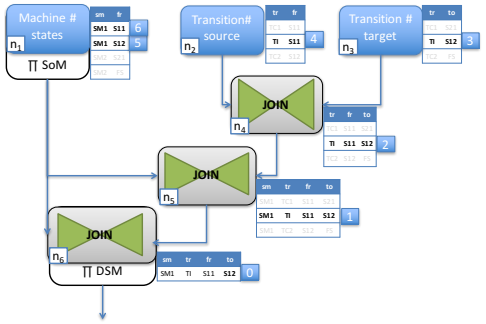
4 Slicing Rete networks of graph patterns

The change algebra of Table 4 precisely specifies how to propagate changes in Rete networks from input nodes to production nodes corresponding to graph patterns. However, an inverse direction of change propagation needs to be defined for debugging purposes.

Slicing of Rete networks will systematically collect dependencies from a(n aggregate) change at a production (pattern) node towards elementary changes at input nodes. More specifically, based on an observed change of a match of a pattern, we need to calculate how to change the caches of each parent node in the Rete network so that those changes consistently imply the specific changes of the match set of a production node. For instance, if a match is included in n_+^{op} (n_-^{op}, respectively) then it needs to be added to (removed from) the cache of the corresponding Rete node n^{op} to observe a specific change n_Δ^P of a production node. In a debugging context, if a specific match of pattern P is missed by the engineer then he or she can ask the slicer to calculate possible model changes that would add the corresponding match n_+^P.

As a slice, we present complete dependency information from aggregate changes to elementary changes by a logic formula over change terms which is calculated by appending new clauses in the form of (ground) change terms along specific matches s while traversing the Rete network from production nodes to input nodes. This slice is informally calculated as follows:

- The input of slicing is the appearance of a new match s in M or the disappearance of an existing match s in M at a production node P, which is a ground term $[\![n_+^P\langle\overline{V^k}\rangle]\!]_s^{M,\Delta}$ or $[\![n_-^P\langle\overline{V^k}\rangle]\!]_s^{M,\Delta}$ appended to the slice.
- For each ground term appended to the slice, we calculate what changes are necessitated at their parent Rete nodes, and append those potential changes to the slices one by one. Formulas are calculated in correspondence with Table 5 for the Rete nodes.
  - For instance, when *a match of a join node disappears* (see Join in Table 5b) then at least one of the corresponding partial matches of its parent nodes need to be removed, captured in the slice by the change terms $[\![n_-^1\langle\overline{V^i}\rangle]\!]_s^{M,\Delta}$ and $[\![n_-^2\langle\overline{V^j}\rangle]\!]_s^{M,\Delta}$ as disjunctive branches.
  - When a *new match of a join node appears* (see Join in Table 5a) then we add new matches to one or both parent nodes n^1, n^2 which is compliant with the match of the join node.
- Special care needs to be taken for *projection* and *anti-join* nodes which may need to fabricate new entities (identifiers) to create ground terms for unbound variables.
- As a base case of this recursive definition, we stop when
  - elementary changes of input nodes are reached (first two lines in Table 5a and Table 5b), or
  - a match already existing in the cache of a Rete node is to be added by a change (see Table 5c), or
  - when the deletion of a match is prescribed by a change which does not exist in M (see Table 5c).



(a) Slicing steps in a Rete network

$n_6{}^{\bowtie}_{-}\langle \_, \_, \_, S_{12}\rangle$
$n_6{}^{\bowtie}_{-}\langle SM_1, TI, S_{11}, S_{12}\rangle$ (0)
$n_5{}^{\bowtie}_{-}\langle SM_1, TI, S_{11}, S_{12}\rangle$ (1)
$n_4{}^{\bowtie}_{-}\langle TI, S_{11}, S_{12}\rangle$ (2)
$n_3{}^{R}_{-}\langle TI, S_{12}\rangle$ (3)
$n_2{}^{R}_{-}\langle TI, S_{11}\rangle$ (4)
$n_1{}^{R}_{-}\langle SM_1, S_{12}\rangle$ (5)
$n_1{}^{R}_{-}\langle SM_1, S_{11}\rangle$ (6)

(b) A Rete slice

Fig. 4: Sample Rete slice for faulty pattern

Definition 10 (Rete Slice) The slice of a change predicate $n_+(t)$ or $n_-(t)$ starting from a node n in a Rete network R over model M and along substitution s is a formula (derived in disjunctive normal form in our case) calculated in accordance with Table 5. □

Example 4 Fig. 4 depicts the sliced Rete network of the faulty version of the Different State Machines pattern. The only difference in its network (as opposed to the Rete network of the correct pattern in Fig. 3a) uses a join node instead of an anti-join node as a production node.

The slicing starts with noticing an undesired tuple where the variable *to* equals to the state S_{12}. At this point, we can ask the slicer how to remove this undesired tuple by calculating the slice of the change predicate $n_6{}^{\bowtie}_{-}\langle \_, \_, \_, S_{12}\rangle$.

0. The memory of node n_6 is checked for tuples matching the input predicate; a single tuple $n_6{}^{\bowtie}_{-}\langle SM_1, TI, S_{11}, S_{12}\rangle$ is found and added to the slice formula.
1. To remove the element from the output of the join node, following Table 5b the corresponding input tuples are to be removed from one of its parents. In this case, the node $n_5{}^{\bowtie}_{-}\langle SM_{11}, TI, S_{11}, S_{12}\rangle$ is added to the formula.
2. The first parent node $n_4{}^{\bowtie}_{-}\langle TI, S_{11}, S_{12}\rangle$ is selected and added to the formula.
3. $n_3{}^{R}_{-}\langle TI, S_{12}\rangle$ is selected as the dependency to remove, and added to the formula. At this point, an input node is reached so the recursion terminates.
4. However, we have to backtrack to node n_4, and evaluate the second case for the join node by adding $n_2{}^{R}_{-}\langle TI, S_{11}\rangle$ to a second branch of the formula.
5. Similarly, $n_1{}^{R}_{-}\langle SM_1, S_{12}\rangle$ and $n_1{}^{R}_{-}\langle SM_1, S_{11}\rangle$ are added to new branches.

Different cases for the same node are handled as disjunctions in the formula.

Node	Change	Append to formula
Entity/0	$[\![n^E_+\langle v\rangle]\!]^{M,\Delta}_s$	$: \top$
Relation/0	$[\![n^R_+\langle v, v_s, v_t\rangle]\!]^{M,\Delta}_s$	$: \top$
Projection/1	$[\![n^\pi_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1_+\langle \overline{V^j}\rangle]\!]^{M,\Delta}_s$
Filter/1	$[\![n^\sigma_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s \wedge \sigma\langle \overline{V^k}\rangle$
Join/2	$[\![n^\bowtie_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1_+\langle \overline{V^i}\rangle]\!]^{M,\Delta}_s \wedge [\![n^2_+\langle \overline{V^j}\rangle]\!]^{M,\Delta}_s$
		$[\![n^1\langle \overline{V^i}\rangle]\!]^{M}_s \wedge [\![n^2_+\langle \overline{V^j}\rangle]\!]^{M,\Delta}_s$
		$[\![n^2\langle \overline{V^j}\rangle]\!]^{M}_s \wedge [\![n^1_+\langle \overline{V^i}\rangle]\!]^{M,\Delta}_s$
Anti-join/2	$[\![n^\triangleright_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1\langle \overline{V^k}\rangle]\!]^{M}_s \wedge [\![n^c_-\langle \overline{V^j}\rangle]\!]^{M,\Delta}_s$
		$[\![n^1_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s \wedge [\![n^2\langle \overline{V^j}\rangle]\!]^{M}_s = \emptyset$
Disjunction/2	$[\![n^\cup_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$
		$[\![n^2_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$

(a) How to update inputs to add match m to output?

Node	Change	Append to formula
Entity/0	$[\![n^E_-\langle v\rangle]\!]^{M,\Delta}_s$	$: \top$
Relation/0	$[\![n^R_-\langle v, v_s, v_t\rangle]\!]^{M,\Delta}_s$	$: \top$
Projection/1	$[\![n^\pi_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1_-\langle \overline{V^k}, \overline{V^n}\rangle]\!]^{M,\Delta}_s$
Filter/1	$[\![n^\sigma_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$
Join/2	$[\![n^\bowtie_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1_-\langle \overline{V^i}\rangle]\!]^{M,\Delta}_s$
		$[\![n^2_-\langle \overline{V^j}\rangle]\!]^{M,\Delta}_s$
Anti-join/2	$[\![n^\triangleright_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$
		$[\![n^1\langle \overline{V^k}\rangle]\!]^{M}_s \wedge [\![n^2_-\langle \overline{V^j}\rangle]\!]^{M,\Delta}_s$
Disjunction/2	$[\![n^\cup_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$	$: [\![n^1\langle \overline{V^k}\rangle]\!]^{M}_s \neq \emptyset \wedge [\![n^1_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s \wedge$ $[\![n^2\langle \overline{V^k}\rangle]\!]^{M}_s = \emptyset$
		$[\![n^1\langle \overline{V^k}\rangle]\!]^{M}_s = \emptyset$ $[\![n^2\langle \overline{V^k}\rangle]\!]^{M}_s \neq \emptyset \wedge [\![n^2_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$
		$[\![n^1\langle \overline{V^k}\rangle]\!]^{M}_s \neq \emptyset \wedge [\![n^1_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s \wedge$ $[\![n^2\langle \overline{V^k}\rangle]\!]^{M}_s \neq \emptyset \wedge [\![n^2_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s$

(b) How to update inputs to remove match m from output?

Node	Change	Append to formula
Add existing tuple	$([\![n\langle \overline{V^k}\rangle]\!]^{M}_s = \bot) \wedge ([\![n_-\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s)$	$: \top$
Remove missing tuple	$([\![n\langle \overline{V^k}\rangle]\!]^{M}_s = \top) \wedge ([\![n_+\langle \overline{V^k}\rangle]\!]^{M,\Delta}_s)$	$: \top$

(c) Handling trivial cases

Table 5: Definition of slices for Rete networks of graph patterns

The final formula looks as follows:

$$
\begin{aligned}
&[\![n_6{}^{\bowtie}_{-}\langle v_1, v_2, v_3, v_4\rangle]\!]^{M}_{\{v_4 \mapsto S_{12}\}} = \\
&\Big(n_3{}^{R}_{-}\langle TI, S_{12}\rangle \wedge n_4{}^{\bowtie}_{-}\langle TI, S_{11}, S_{12}\rangle \wedge n_5{}^{\bowtie}_{-}\langle SM_{11}, TI, S_{11}, S_{12}\rangle \wedge n_6{}^{\bowtie}_{-}\langle SM_1, TI, S_{11}, S_{12}\rangle\Big) \vee \\
&\Big(n_2{}^{R}_{-}\langle TI, S_{11}\rangle \wedge n_4{}^{\bowtie}_{-}\langle TI, S_{11}, S_{12}\rangle \wedge n_5{}^{\bowtie}_{-}\langle SM_{11}, TI, S_{11}, S_{12}\rangle \wedge n_6{}^{\bowtie}_{-}\langle SM_1, TI, S_{11}, S_{12}\rangle\Big) \vee \\
&\Big(n_1{}^{R}_{-}\langle SM_1, S_{12}\rangle \wedge n_5{}^{\bowtie}_{-}\langle SM_{11}, TI, S_{11}, S_{12}\rangle \wedge n_6{}^{\bowtie}_{-}\langle SM_1, TI, S_{11}, S_{12}\rangle\Big) \vee \\
&\Big(n_1{}^{R}_{-}\langle SM1, S11\rangle \wedge n_6{}^{\bowtie}_{-}\langle SM_1, TI, S_{11}, S_{12}\rangle\Big) \qquad \square
\end{aligned}
$$

Although the formula refers to all nodes of the Rete network, the slice describes a reduced model: (1) the model element tuples unrelated to the criteria are not included, and (2) the tuples in a single disjunctive branch describe a possible series of operations that would result in a tuple matching the input predicate to appear or disappear.

5 Related work

Traditional program slicing techniques have been regularly and exhaustively surveyed in the past in papers like [16,17]. The current paper focuses on model transformation slicing [9,10,11], more specifically on incremental model queries. The main difference with respect to traditional approaches is that query slicing has to consider the specification and the model simultaneously.

Slicing of declarative programs The closest related work addresses the slicing of logic programs as declarative graph patterns [5,18] share certain similarities with logic programs. Forward slicing of Prolog programs are discussed in [19] based on partial evaluation, while [20] executes static and dynamic slicing of logic programs based on the procedural behaviour of the programs. [21] augments the data flow analysis with control-flow dependencies in order to identify the source of a bug included in a logic program and was extended in [22] to the slicing of constraint logic programs (with fixed domains). Program slicing for the Alloy language was proposed in [23] as a novel optimization strategy to improve the verification of Alloy specifications. Our conceptual extension to these existing slicing techniques is the incorporation of model elements into the slices.

Slicing queries over databases In the context of databases and data warehousing, related approaches called *data lineage tracing* [24] or *data provenance problem* [25] aim to explain why a selected record exists in a materialized view. These approaches focus on identifying the records of the original tables that contribute to a selected record, and expect the queries be correct. A further difference to our contribution is that storing partial results in a data warehousing context can be impractical due to high (memory) costs while in case of the Rete algorithm, these partial results are already cached to be available for slicing.

Model slicing Model slicing [26] techniques have already been successfully applied in the context of MDD. Slicing was proposed for model reduction purposes in [27,28] to make the following automated verification phase more efficient.

Lano et. al. [29] exploits both declarative elements (like pre- and postconditions of methods) and imperative elements (state machines) to construct UML model slices by using model transformations. The slicing of finite state machines in a UML context was studied by Tratt [30], especially, to identify control dependence. A similar study was also executed for extended finite state machines in [31]. A dynamic slicing technique for UML architectural models is introduced in [32] using model dependence graphs to compute dynamic slices based on the structural and behavioral (interactions only) UML models.

Metamodel pruning [33] can also be interpreted as a slicing problem where the effective metamodel is automatically derived as a view. Moreover, model slicing is used in [34] to modularize the UML metamodel into a set of small metamodels for each UML diagram type. Various model slicing techniques are merged by Blouin et al. [35] into a single, generative framework, using different approaches for different models. Still, none of the existing model slicing approaches address the slicing of model queries, the main focus of our work.

Model transformation debugging Slicing can be beneficial for debugging model transformations. The authors of [36] propose a dynamic tainting technique for debugging failures of model transformations, and propose automated techniques to repair input model faults [37]. Colored Petri nets are used for underlying formal support for debugging transformations in [38]. The debugging of triple graph grammar transformations is discussed in [39], which envisions the future use of slicing techniques in the context of model transformations.

6 Conclusion and future work

In this paper, we defined a dynamic slicing technique for Rete networks derived from graph patterns. As a slicing criterion, the appearance of a new match or the disappearance of an existing match is selected in a production node of the Rete network. Since a Rete network also caches partial matches, it is possible to follow match dependencies step by step back to the input nodes storing elementary graph nodes and edges. Such dependencies constitute the slice is captured as formulas over terms of a change algebra. As the main contribution, we provided a formal slicing technique for Rete networks of graph patterns constituted from the most frequently used language elements of the EMF-IncQuery framework. Our slicing technique was illustrated on a running example of UML state machines.

In the future, we plan to integrate this slicing approach into EMF-IncQuery [3] in order to use it for various tasks, such as presenting this slice together with the Rete networks graphically, easing the debugging of erroneous model queries. Furthermore, the approach seems promising for declarative bidirectional view model synchronization well, as it enables calculating possible source model changes for view model changes automatically.

Acknowledgements The authors would like to thank István Ráth for the valuable discussions during the preparation of this paper.

References

1. Reder, A., Egyed, A.: Incremental consistency checking for complex design rules and larger model changes. In: Model Driven Engineering Languages and Systems. LNCS. Springer (2012) 202–218
2. Bergmann, G., Horváth, A., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Model Driven Engineering Languages and Systems. LNCS. Springer (2010) 76–90
3. Ujhelyi, Z., Hegedüs, A., Bergmann, G., Horváth, A., Ráth, I., Varró, D.: EMF-IncQuery: An integrated development environment for live model queries. Science of Computer Programming **98** (2015) 80–99
4. Hegedüs, A., Horváth, A., Ráth, I., Starr, R., Varró, D.: Query-driven soft traceability links for models. Software and Systems Modeling (2014) 1–24
5. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A graph query language for EMF models. In: Theory and Practice of Model Transformations. LNCS. Springer (2011) 167–182
6. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**(1) (September 1982) 17–37
7. Varró, G., Friedl, K., Varró, D.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. Electronic Notes in Theoretical Computer Science **152**(0) (2006) 191 – 205 Proceedings of the Int. Workshop on Graph and Model Transformation (GraMoT 2005).
8. Búr, M., Ujhelyi, Z., Horváth, A., Varró, D.: Local search-based pattern matching features in EMF-IncQuery. In: Graph Transformation. LNCS. Springer International Publishing (2015) 275–282
9. Ujhelyi, Z., Horváth, A., Varró, D.: Dynamic backward slicing of model transformations. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. ICST '12, Washington, DC, USA, IEEE Computer Society (2012) 1–10
10. Clarisó, R., Cabot, J., Guerra, E., de Lara, J.: Backwards reasoning for model transformations: Method and applications. Journal of Systems and Software (2015)
11. Burgueno, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. Software Engineering, IEEE Transactions on **41**(5) (May 2015) 490–506
12. Bergmann, G.: Incremental Model Queries in Model-Driven Design. PhD dissertation, Budapest University of Technology and Economics, Budapest (2013)
13. Gurevich, Y.: Sequential Abstract-state Machines Capture Sequential Algorithms. ACM Trans. Comput. Logic **1**(1) (July 2000) 77–111
14. The JBoss Project: Drools - The Business Logic integration Platform (2014) `http://www.jboss.org/drools`.
15. Ghamarian, A., Jalali, A., Rensink, A.: Incremental pattern matching in graph-based state space exploration. Electronic Communications of the EASST **32** (2011)
16. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3) (1995) 121–189
17. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes **30**(2) (2005) 1–36
18. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming **68**(3) (October 2007) 214–234
19. Leuschel, M., Vidal, G.: Forward slicing by conjunctive partial deduction and argument filtering. In: Programming Languages and Systems. Volume 3444 of LNCS. Springer (2005) 140–140

20. Vasconcelos, W.: A flexible framework for dynamic and static slicing of logic programs. In: Practical Aspects of Declarative Languages. LNCS. Springer (1998) 259–274
21. Szilágyi, G., Harmath, L., Gyimóthy, T.: The debug slicing of logic programs. Acta Cybernetica **15**(2) (2001) 257–278
22. Szilágyi, G., Gyimóthy, T., Małuszyński, J.: Static and dynamic slicing of constraint logic programs. Automated Software Engineering **9** (2002) 41–65
23. Uzuncaova, E., Khurshid, S.: Kato: A program slicing tool for declarative specifications. In: 29th Int. Conf. on Software Engineering, IEEE (2007) 767–770
24. Cui, Y., Widom, J., Wiener, J.L.: Tracing the lineage of view data in a warehousing environment. ACM Transactions on Database Systems **25**(2) (2000) 179–227
25. Freire, J., Koop, D., Santos, E., Silva, C.T.: Provenance for Computational Tasks: A Survey. Computing in Science & Engineering **10**(3) (2008) 11–21
26. Kagdi, H., Maletic, J.I., Sutton, A.: Context-free slicing of UML class models. In: 21st Int. Conf. on Software Maintenance ICSM05, IEEE (2005) 635–638
27. Schaefer, I., Poetzsch-Heffter, A.: 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) 25–32
28. Shaikh, A., Clarisó, R., Wiil, U.K., Memon, N.: Verification-driven slicing of UML/OCL models. In: 25th IEEE/ACM Int. Conf. on Automated Software Engineering, ACM (2010) 185–194
29. Lano, K., Kolahdouz-Rahimi, S.: Slicing of UML models using model transformations. In: Model Driven Engineering Languages and Systems. LNCS. Springer (2010) 228–242
30. Androutsopoulos, K., Clark, D., Harman, M., Li, Z., Tratt, L.: Control dependence for extended finite state machines. In: Fundamental Approaches to Software Engineering, 12th Int. Conf., FASE 2009. LNCS, Springer (2009) 216–230
31. Korel, B., Singh, I., Tahat, L., Vaysburg, B.: Slicing of state-based models. Software Maintenance, IEEE Int. Conf. on (2003) 34 – 43
32. Lallchandani, J.T., Mall, R.: A dynamic slicing technique for UML architectural models. IEEE Transactions on Software Engineering **37**(6) (2011) 737 –771
33. Sen, S., Moha, N., Baudry, B., Jézéquel, J.: Meta-model pruning. In: Model Driven Engineering Languages and Systems. Springer (2009) 32–46
34. Bae, J.H., Lee, K., Chae, H.S.: Modularization of the UML metamodel using model slicing. In: 3rd Int. Conf. on Information Technology: New Generations, IEEE (2008) 1253–1254
35. Blouin, A., Combemale, B., Baudry, B., Beaudoux, O.: Modeling model slicers. In: Model Driven Engineering Languages and Systems. LNCS. Springer (2011) 62–76
36. Dhoolia, P., Mani, S., Sinha, V.S., Sinha, S.: Debugging model-transformation failures using dynamic tainting. In: Proceedings of the 24th European conference on Object-oriented programming. ECOOP'10, Springer-Verlag (2010) 26–51
37. Mani, S., Sinha, V.S., Dhoolia, P., Sinha, S.: Automated support for repairing input-model faults. In: 25th IEEE/ACM Int. Conf. on Automated Software Engineering, ACM (2010) 195–204
38. Schoenboeck, J., Kappel, G., Kusel, A., Retschitzegger, W., Schwinger, W., Wimmer, M.: Catch me if you can - debugging support for model transformations. In: Model Driven Engineering Languages and Systems. LNCS, Springer (2010) 5–20
39. Seifert, M., Katscher, S.: Debugging triple graph grammar-based model transformations. In: Fujaba Days. (2008) 19–25