Incremental Pattern Matching for the Efficient Computation of Transitive Closure[⋆]

Gábor Bergmann, István Ráth, Tamás Szabó, Paolo Torrini, Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2
{bergmann,rath,varro}@mit.bme.hu, {szabta89,ptorrx}@gmail.com

Abstract. Pattern matching plays a central role in graph transformations as a key technology for computing local contexts in which transformation rules are to be applied. Incremental matching techniques offer a performance advantage over the search-based approach, in a number of scenarios including on-the-fly model synchronization, model simulation, view maintenance, well-formedness checking and state space traversal [1,2]. However, the incremental computation of *transitive closure* in graph pattern matching has started to be investigated only recently [3]. In this paper, we propose multiple algorithms for the efficient computation of *generalized transitive closures*. As such, our solutions are capable of computing reachability regions defined by *simple graph edges as well as complex binary relationships defined by graph patterns*, that may be used in a wide spectrum of modeling problems. We also report on experimental evaluation of our prototypical implementation, carried out within the context of a stochastic system simulation case study.

1 Introduction

In model-driven software engineering, queries and transformations are nowadays core techniques to process models used to design complex embedded or business systems. Unfortunately, many modeling tools used in practice today have scalability issues when deployed in large-scale modeling scenarios, motivating research and development efforts to continue improving performance for essential use-cases such as model management, transformations, design-time analysis and code generation.

Transitive closure is generally needed to express model properties which are recursively defined, often used in reasoning about partial orders, and thus widely found in modeling applications, e.g. to compute model partitions or reachability regions in traceability model management [4] and business process model analysis [5]. In graph transformations, recursive graph patterns are most frequently

[⋆] This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003) project and the János Bolyai Scholarship and the grant TÁMOP-4.2.1/B-09/1/KMR-2010-0002.

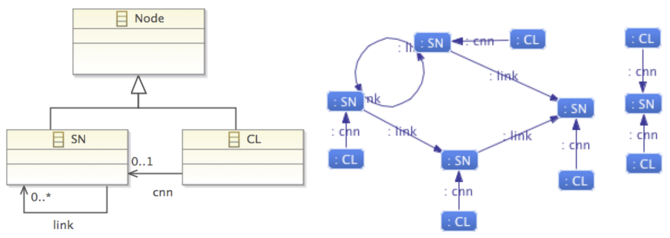
used to specify transitive closure for processing recursive model structures [6]. At the meta-level, they may provide the underpinnings for n-level metamodeling hierarchies where transitive type-subtype-instance relationships need to be maintained [7], or for maintaining order structures such as those proposed in [8] for spatially aware stochastic simulation.

Incremental graph pattern matching has already demonstrated scalability in a number of such scenarios [1,9,10], especially when pattern matching operations are dominating at runtime (e.g. view maintenance, model synchronization, well-formedness checking and state space traversal [2]). However, recursive incremental pattern matching was, up to now, supported only for acyclic subgraphs. Therefore, as it has been recently recognized in [3], the efficent integration of transitive closure computation algorithms for graphs would provide a crucial extension to the current capabilities of incremental pattern matchers.

Challenges. By analyzing related work, we observed that in order to efficiently adapt transitive closure computation for the specific needs of graph pattern matching in graph transformations, three key challenges need to be addressed. First, the Rete algorithm (used e.g. in VIATRA2 [6], EMF-INCQUERY [10], GROOVE [2], JBoss Drools [11] and other tools) does not *handle cyclic closure correctly*, i.e. in the presence of graph cycles, incremental updates of recursive patterns may yield false matching results. Second, for functionally complete pattern matching it is important to support *generic transitive closure*, i.e. the ability to compute the closure of not only simple graph edges (edge types), but also derived edges defined by binary graph patterns that establish a complex logical link between a source and a target vertex. Finally, the adaptation should *align with the general performance characteristics of incremental pattern matching* to impose a low computational overhead on model manipulation operations and minimize runtime memory overhead.

Contributions of the paper. To address the above challenges, we adapted different general purpose graph transitive closure algorithms [12,13] to the specific needs of incremental graph pattern matching. After analyzing common characteristics of several modeling scenarios, we developed a novel version of *IncSCC* [12], the incremental transitive computation algorithm based on the maintenance of strongly connected components. We demonstrate the feasibility of our approach by extending the high-level pattern language of the VIATRA2 framework to support the correct computation of transitive closure. In order to evaluate experimentally the performance of the extended pattern matcher, we relied on the GRaTS stochastic simulator [14] built on VIATRA2 that was used to run a simple structured network model scenario, specifically tailoring the simulation to compare the characteristics of these algorithms.

Structure. The rest of the paper is structured as follows. Sec. 2 introduces (meta)modeling and graph transformation as preliminaries necessary to understand the rest of the discussion, and describes a case study – the stochastic simulation of a structured network – that illustrates the technical details of our approach. Sec. 3 elaborates the transitive closure problem, and describes the novel adaptation of the *IncSCC* algorithm in detail. Sec. 4 reports on our per-



(a) Metamodel (b) Instance model example

Fig. 1. Meta- and instance models of the case study

formance benchmark findings. The paper is concluded by a discussion of related work in Sec. 5 and the conclusion in Sec. 6.

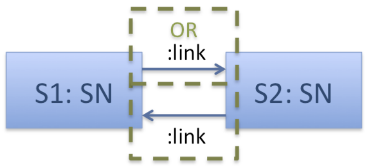
2 Preliminaries

2.1 Metamodeling and Graph Patterns

We rely on the VIATRA2 model transformation tool [6] as the technological framework for our approach. However, all metamodels will be presented in a traditional EMF syntax to stress that all the main concepts presented could be transferred to other modeling environments as well, e.g. by using the EMF-INCQUERY framework [10] with EMF. VIATRA2 uses a canonical metamodeling approach for its model repository, with three core concepts (entities, properties and relations) that correspond to vertices, attributes (labels) and edges of a typed graph. All metalevels (types and instances) are uniformly represented in a containment hierarchy that makes up the VIATRA2 *model space*.

Case study example (meta- and instance models). Theoretically, here we rely on a typed single pushout graph transformation approach with attributes and negative application conditions [15]. We consider a structured network evolving according to some rules. The wider network is formed by an overlay on supernodes that represent external ports of local area networks, and we may query the existence of connections between any pair of nodes. A simple VIATRA2 metamodel is shown in Fig. 1a. We model networks as graphs that may consist of two kinds of `Node`s: they may either be *LAN* clients (instances of the type `CL`) or *LAN* supernodes (`SN`) to which clients may connect (through connections of type `cnn`). Supernodes can connect to each other through connections of type `link` (see [16] for technical details). A sample instance model (as visualized in VIATRA2) is shown in Fig. 1b.

Case study example (graph patterns). A sample graph pattern is shown in Fig. 2, using a graphical concrete syntax for illustration (on the left) and also the textual representation from the actual VIATRA2 transformation (on the right). This pattern represents the `linked` relation between any two supernodes $S1$ and $S2$ (both required to be of type `SN`, as expressed in lines 2–3 and 6–7) that are



```
pattern linked(S1,S2) = {
  SN(S1);
  SN(S2);
  SN.link(S1,S2);
 } or {
  SN(S1);
  SN(S2);
  SN.link(S2,S1);
 }
```

Fig. 2. Graph pattern to capture linked supernodes

connected by a relation of type `link` in either direction (as expressed by means of the *or* construct).

2.2 Incremental Pattern Matching in Graph Transformations

Graph transformation systems use pattern matching algorithms to determine the parts of the model that correspond to the *match set* of a graph pattern. *Incremental pattern matching engines* (INC) [17,18,19,2] rely on a *cache* in which the matches of a pattern are stored explicitly. The match set is readily available from the cache at any time without searching, and the cache is incrementally updated whenever changes are made to the model. The result can be retrieved in constant time – excluding the linear cost induced by the size of the match set itself –, making pattern matching extremely fast. The trade-off is space consumption of the match set caches, model manipulation performance overhead related to cache maintenance, and possibly the initialization cost of the cache.

In terms of transformation performance, INC has been observed by several experiments [17,1,2] to be highly scalable in a number of scenarios, particularly when complex patterns with moderately sized match sets are matched frequently, without excessive changes to the model in-between. This is typical of the *as-long-as-possible* style of transformation execution, which is frequently used for model simulation purposes.

Overview of Rete-based incremental pattern matching. Rete [20] is a well-known incremental pattern matching technique from the field of rule-based expert systems. A Rete net consists of *Rete nodes* (not to be confused with the vertices of the model graph), each storing a relation corresponding to the *match set* of a partial pattern, i.e. the set of model element tuples that satisfy a given subset of pattern constraints. Rete nodes are connected by *Rete edges* so that the content of a Rete node can be derived from its parent nodes. The Rete edges propagate incremental updates of the match sets, i.e. whenever the contents of a Rete node is changed, child nodes are also updated using the difference (inserted or deleted tuples). There are three types of nodes in the Rete net: (i) *input nodes* serve as the knowledge base of the underlying model, e.g. there is a separate node for each entity or relation type, enumerating the set of instances as tuples; (ii) *intermediate nodes* perform operations to derive a set of partial matches; finally, (iii) *production nodes* store the complete match set of a given pattern.

2.3 Stochastic Model Simulation by Graph Transformations

A simulation framework called *GRaTS* [21] for *generalized stochastic graph transformation* [22], along the lines of [23], has been introduced in [14] and further extended [21], built on top of the Rete-based pattern matching infrastructure of VIATRA2, to support the design-time analysis of discrete event systems and to validate stochastic properties of the system-under-design. A model in GraTS consists of a graph transformation system in which each transformation *action rule* (see Fig. 3), is augmented with a probability distribution governing the delay of its application (in our simple case study, we use exponential distributions that are characterised by a *weight* parameter – a higher weight will result in the rule being executed more frequently). Additionally, each valid action rule match represents a possible event. A stochastic experiment consists of a model together with a set of transformation rules, each used as a *probe rule*, allowing to aggregate user-defined statistics on simulation runs.

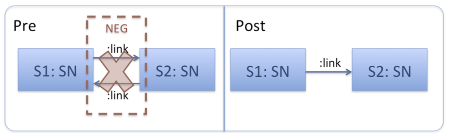
Simulation in GraTS procedes by discrete steps, each determined by the execution of an action rule, leading from one state to another, where each state is characterised by the set of enabled events — i.e. all the valid rule matches, maintained as a priority queue. Statistics are also collected step-wise, by computing valid matches of probe rules and aggregating data. The simulation engine relies heavily on the incremental pattern matcher to keep track efficiently of valid rule matches and especially, in the case of events, of their enabling time [14,8,21].

Discrete event stochastic simulation can be characterised as a semi-Markov process [24,25], as – aside from exponential distributions found in most stochastic tools – *generalised probability distributions are supported*. Even though this capability is not used in the simple model of this paper, it allows for modelling of complex network scenarios [22] involving hybrid features, such as jitter and bandwidth in realistic modelling of VoIP [22,16], in which transitive closures can typically arise quite often [14].

Case study example (action rule). In our case study, a sample action rule is shown in Fig. 3. Here, the `AddLink` operation is defined, whereby redundant overlay `links` can be added to a pair of LAN supernodes $S1, S2$ that are not directly connected, as expressed by the negative application condition (line 5) referring to the `linked` pattern of Fig. 2. By an execution of this rule for a given $S1, S2$ pair, a new `link` will be added to the model (line 7).

3 Transitive Closure in a Rete-based Matcher

A brief overview about transitive closure in graph transformations is given in Sec. 3.1. The applied solution - a special Rete node capable of efficient incremental transitive closure calculations - is discussed in Sec. 3.2. Then Sec. 3.3 presents the general purpose incremental graph transitive closure algorithms that we we adapted and evaluated.



```
gtrule AddLink() = {
 precondition pattern lhr(S1,S2) = {
  SN(S1);
  SN(S2);
  neg find linked(S1,S2);
 } action {
  new(SN.link(S1,S2));
 }
}
```

Fig. 3. Graph transformation rule to add redundant overlay links between disconnected supernodes

3.1 Transitive Closure

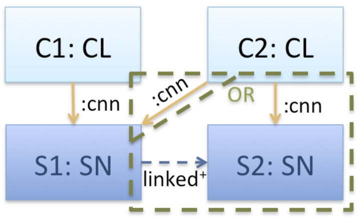
Generic and Irreflexive Transitive Closure For a binary relation E over a domain D, the *irreflexive transitive closure* E^+ consists of $\langle u, v\rangle$ pairs of elements for which there is a *non-empty* finite linked sequence $\langle u = w_0, w_1\rangle, \langle w_1, w_2\rangle, \ldots, \langle w_{k-1}, w_k = v\rangle$ of pairs in E.

In case of *generic transitive closure*, the base relation E is a "derived edge", not restricted to simple graph edges, but defined by any two-parameter graph pattern (e.g. with path expressions, attribute checks). We focus on the most general approach: generic, irreflexive transitive closure.

Transitive Closure Operations Any program computing the transitive closure E^+ of a binary relation E is required to expose a subroutine `Construct(E)` that builds a data structure for storing the result and possibly auxiliary information as well. Afterwards, the following reachability queries can be issued: `Query(Src,Trg)` returns whether Trg is reachable from Src; `Query(Src,?)` returns all targets reachable from Src, while `Query(?,Trg)` returns all sources from where Trg can be reached; finally `Query(?,?)` enumerates the whole E^+.

In case of incremental computation, the following additional subroutines have to be exposed: `Insert(Src,Trg)` updates the data structures after the insertion of the $\langle Srg, Trg\rangle$ edge to reflect the change, while `Delete(Src,Trg)` analogously maintains the data structures upon an edge deletion. To support further incremental processing, both of these methods return the delta of E^+, i.e. the set of source-target pairs that became (un)reachable due to the change.

Strongly Connected Components (SCC), Condensed Graph A graph is strongly connected iff all pairs of its vertices are mutually transitively reachable. An SCC of a graph is a maximal subset of vertices within a graph that is strongly connected. As the SCC of a vertex v is the intersection of the set of ancestors and descendants of the vertex, each graph has a unique decomposition S into disjoint SCCs. For a graph $G(V, E)$, the SCCs form the *condensed graph* $G_c(S, E_c)$, where two SCCs are connected iff any of their vertices are connected: $E_c = \{\langle s_i, s_j\rangle \mid s_i, s_j \in S \land \exists u \in s_i, v \in s_j : \langle u, v\rangle \in E\}$. It follows from the definitions that a condensed graph is always acyclic.



```
 1 pattern pconnected(C1,C2) = {
 2   SN(S1);
 3   CL.cnn(C1,S1);
 4   CL.cnn(C2,S1);
 5 } or {
 6   CL.cnn(C1,S1);
 7   CL.cnn(C2,S2);
 8   // transitive closure
 9   find linked+(S1,S2);
10 }
```

Fig. 4. Transitive closure within graph pattern to capture overlay-connected clients

Case study example (transitive closure in graph patterns) The example in Fig. 4 demonstrates transitive closure features in graph pattern matching. A transitive closure over the overlay network of supernodes is specified by the pattern `pconnected` that defines the relationship between any two client nodes $C1, C2$ which are (i) either sharing a common supernode to which they are both directly connected along `cnn` edges (lines 2–4), or (ii) their "pconnection" is indirect in the sense that their supernodes $S1, S2$ are reachable from each other through a transitive `linked+` relationship (lines 6–9). The latter is the generic transitive closure of the derived edge defined by binary pattern `linked` (see Fig. 2).

3.2 Integration of Graph Transitive Closure into Rete

A transitive closure result will be represented by a Rete node, like any other pattern. We integrate dynamic transitive closure algorithms into Rete nodes by exploiting the operations specified in Sec. 3.1. Generic transitive closure (see Sec. 3.1) is achieved by attaching such a Rete node to a parent node that matches a graph edge or an arbitrary binary graph pattern (derived edge).

Fig. 5 (a) shows the transitive closure node in the Rete network. It is an intermediate node which receives updates from a binary graph pattern (here denoted as binary relation E) and forms a two-way interface between Rete and a transitive closure maintenance algorithm. Whenever the Rete node for E^+ receives an insertion / deletion update from its parent node E, the `Insert()`/`Delete()` subroutine is invoked. The subroutine computes the necessary updates to E^+, and returns these delta pairs, which will then be propagated along the outgoing edge(s) of the Rete node. Queries are invoked when initializing the child nodes, and later as a quick lookup to speed up join operations over the node contents.

Alternatively, transitive closure can be expressed as a recursive graph pattern. This solution was rejected, as Rete, having first-order semantics without fixpoint operators, might incorrectly yield a (still transitive) superset of the transitive closure: in graph models containing cycles, obsolete reachabilities could cyclically justify each other after their original justification was deleted.

Case study example (transitive closure Rete node) Here we demonstrate the behaviour of a Rete node that computes the transitive closure E^+ of the binary graph pattern E, e.g. `linked+` for the overlay network `linked` between super

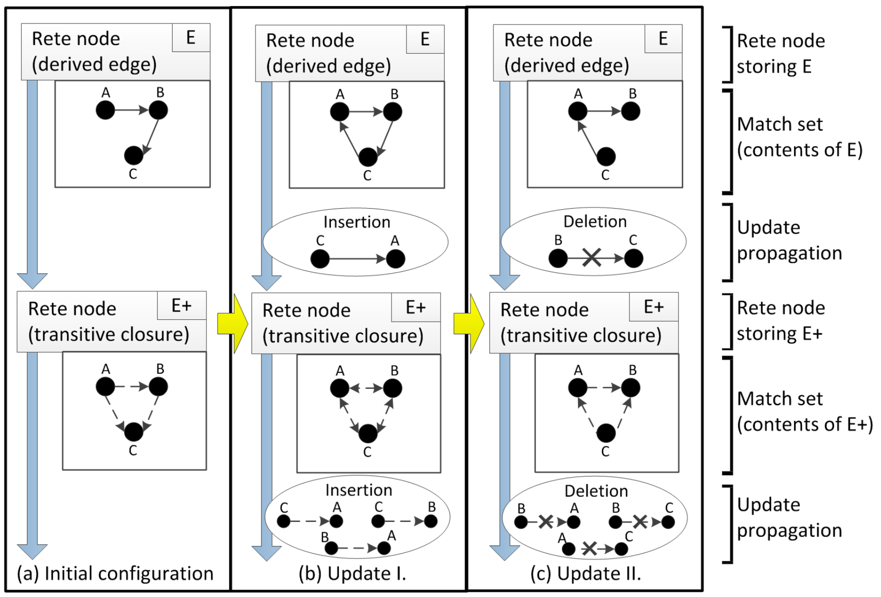


Fig. 5. Transitive closure Rete node during insertion of $\langle C, A\rangle$ and deletion of $\langle B, C\rangle$

nodes. Initially, as seen in Fig. 5 (a), the parent node E stores `linked`, i.e. the binary relation $\{\langle A, B\rangle, \langle B, C\rangle\}$. Its child node E^+ contains the set of reachable pairs: $\{\langle A, B\rangle, \langle A, C\rangle, \langle B, C\rangle\}$.

Fig. 5 (b) shows the insertion of edge $\langle C, A\rangle$ into E. Rete propagates this update from the E to E^+, where the operation `Insert(C,A)` is invoked to adjust the transitive closure relation to $\{\langle A, B\rangle, \langle A, C\rangle, \langle B, A\rangle, \langle B, C\rangle, \langle C, A\rangle, \langle C, B\rangle\}$, i.e. the whole graph becomes strongly connected. The computed difference (delta) is the insertion of $\{\langle B, A\rangle, \langle C, A\rangle, \langle C, B\rangle\}$ into E^+, which is propagated in the Rete network to child nodes of E^+.

Finally, Fig. 5 (c) shows an edge deletion. E^+ is notified of the deletion of $\langle B, C\rangle$ from E, and invokes `Delete(B,C)`. Thus E^+ becomes $\{\langle A, B\rangle, \langle C, A\rangle, \langle C, B\rangle\}$, and the propagated delta is the deletion of $\{\langle A, C\rangle, \langle B, A\rangle, \langle B, C\rangle\}$.

3.3 Incremental Graph Transitive Closure Maintenance Algorithms

An incremental transitive closure algorithm is required to operate the Rete node proposed in Sec. 3.2. From the rich literature (see Sec. 5), we selected and adapted two such algorithms. Here we provide an overview of their core ideas.

DRed - Delete and REDerive This simple algorithm explicitly stores E^+. `Construct()` initializes the closure relation using a standard non-incremental algorithm, and `Query()` is directly answered based on E^+. The update operations are derived from the DRed [26] algorithm for recursive Datalog queries.

`Insert(Src,Trg)` computes the newly reachable pairs as $E^* \circ \{\langle Src, Trg\rangle\} \circ E^*$, and adds them to E^+ (unless already reachable), where $A \circ B := \{\langle u, v\rangle \mid \exists w : \langle u, w\rangle \in A \wedge \langle w, v\rangle \in B\}$.

`Delete(Src,Trg)` computes an overestimation of the delta as $E^+_D = (E^* \circ \{\langle Src, Trg\rangle\} \circ E^*) \setminus E$, and marks these pairs for deletion. Then it attempts to derive again these marked reachability pairs using unaffected ones as $E^+_D \bigcap (E \circ (E^+ \setminus E^+_D))$; successfully rederived pairs are removed from E^+_D, allowing further ones to be rederived until a fixpoint is reached. The final contents of E^+_D are the deleted reachability pairs removed from E^+.

IncSCC - Incremental Maintenance of Strongly Connected Components We have also implemented the transitive closure algorithm *IncSCC*, where the name *IncSCC* stands for Incremental SCC maintenance.

The main idea of the algorithm, from [12], is to reduce update time and memory usage by eliminating unnecessary reachability information, namely, that each vertex is reachable from every other vertex within the same SCC. Thus, the two concerns of the algorithm are maintaining (i) a decomposition S of the graph into SCCs, and (ii) transitive reachability within the condensed graph. The latter is a simpler problem with several efficient solutions, as the condensed graph is acyclic; our implementation relies on the "basic algorithm" from the original paper [12], that will be called the *Counting Algorithm*, as it simply keeps track of the number of derivations of each transitive reachability pair.

In the following, we give a brief overview of (our implementation of) IncSCC. For details and analysis, refer to [12].

Implementing `Construct(E)` The SCC partitioning of the initial graph are computed using *Tarjan's algorithm* [27] based on depth-first search. Afterwards, the condensed graph is constructed, and the Counting Algorithm is initialized to provide reachability information between SCCs.

Implementing `Query()` operations As the most significant improvement over [12], the transitive closure relation E^+ is not stored explicitly in our IncSCC solution to reduce the memory footprint. However, reachability in graph $G(V, E)$ can be reconstructed from the partitioning S of SCCs and the reachability relation E^+_c of condensed graph $G_c(S, E_c)$, since for $s_1, s_2 \in S, u \in s_1, v \in s_2 : \langle s1, s2\rangle \in E^*_c$ iff $\langle u, v\rangle \in E^*$. Therefore when receiving a reachability query, the parameter vertices are mapped to SCCs, where reachability information in the condensed graph is provided by the Counting Algorithm. Vertices enumerated in the answer are obtained by tracing back the SCCs to vertices.

Implementing `Insert(Source,Target)` First, a lookup in S maps the vertices to SCCs. Afterwards, there are three possible cases to distinguish. If (i) $\langle Source, Target\rangle$ are in different SCCs, the new edge of the condensed graph is handled by the Counting Algorithm, which can confirm that no cycle is created in the condensed graph. If, however, (ii) the inserted edge caused a cycle in the condensed graph, then the cycle is collapsed into a single SCC. Finally, if (iii) $\langle Source, Target\rangle$ are in the same SCC, there is no required action. Computation details of the delta relation is omitted here for space considerations.

Implementing `Delete(Source,Target)` The algorithm first performs a lookup in S to map the vertices to SCCs; afterwards, we once again distinguish three possible cases. (1) If $\langle Source, Target\rangle$ are in the same SCC but $Target$ remains reachable from $Source$ after the edge deletion (as confirmed by a depth-first-search), no further actions are required. (2) If $\langle Source, Target\rangle$ are in the same SCC but $Target$ is no longer reachable from $Source$ after the edge deletion, then the SCC is broken up (using Tarjan's algorithm) into smaller SCCs, because it is no longer strongly connected. Finally, (3) if $\langle Source, Target\rangle$ are in different SCCs, then the edge is deleted from the condensed graph, which is in turn is handled by the Counting Algorithm.

4 Benchmarking

4.1 Measurement Scenario

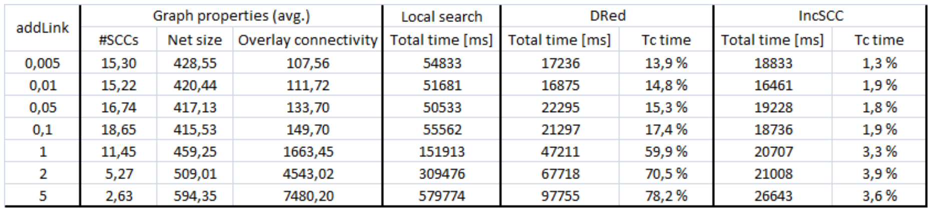
To find out the performance differences between various pattern matching algorithms for transitive closure, we ran a series of measurements[1] on simplified stochastic model simulation process, used to analyse the probability of the network being (fully) connected (so that each client can communicate with every other one, through their direct supernode connections and the transitive overlay links between supernodes). The connectivity measure was registered through a *probe* of the match set of the `pconnected` pattern (Fig. 4), reporting the size of the match set after each simulation step.

A simulation run consisted of 2000 steps (rule applications), and along with the total execution time of the run, we also registered the wall times for various sub-phases – such as the time it took to propagate updates through the transitive closure Rete node – using code instrumentation. The experiments were carried out with three different strategies of evaluating graph patterns and transitive closure: (a) local search pattern matching as implemented in VIATRA2, (b) Rete-based incremental matching with the DRed algorithm for transitive closure, (c) Rete with IncSCC for transitive closure. We have investigated the performance of these solutions in two series of experiments.

The first series considered various model structures induced by different probability weight parameterizations of the `addLink` rule (i.e. increasingly frequent applications of the rule). It was run on an initial model of 2000 vertices in 20 isolated components, each containing 10 supernodes and 9 clients per supernode. The second series settled on a fixed value 2 of `addLink` weight (thus keeping the frequency of the rule application roughly constant), and considered increasingly

[1] Performed on Intel Core i5 2,5 GHz, 8 GB RAM, Java Hotspot Server vm build 1.7.0_02-b13 on 64-bit Windows, with 4 Rete threads. The entire corpus is available at `http://viatra.inf.mit.bme.hu/publications/inctc`

Table 1. Graph properties and simulation performance, depending on `addLink` weight



addLink	Graph properties (avg.)			Local search	DRed		IncSCC	
	#SCCs	Net size	Overlay connectivity	Total time [ms]	Total time [ms]	Tc time	Total time [ms]	Tc time
0,005	15,30	428,55	107,56	54833	17236	13,9 %	18833	1,3 %
0,01	15,22	420,44	111,72	51681	16875	14,8 %	16461	1,9 %
0,05	16,74	417,13	133,70	50533	22295	15,3 %	19228	1,8 %
0,1	18,65	415,53	149,70	55562	21297	17,4 %	18736	1,9 %
1	11,45	459,25	1663,45	151913	47211	59,9 %	20707	3,3 %
2	5,27	509,01	4543,02	309476	67718	70,5 %	21008	3,9 %
5	2,63	594,35	7480,20	579774	97755	78,2 %	26643	3,6 %

larger models sizes (from 1000 to 10000 vertices), initially divided into 10 to 100 components similarly to the first series.

4.2 Results and Analysis

Table 1 shows the results of the first experiment series. For each value of `addLink` *weight*, we have displayed (i) the values of the probes (as well as the number of strongly connected components) averaged over an entire simulation run; (ii) for each of the three solutions the total execution time and, in case of the incremental algorithms, (iii) the time spent initializing and updating the transitive closure node (expressed as a percentage of total time).

The first series of experiments reveals that as the application frequency of `addLink` increases, the frequent rule executions make the graph more and more connected. DRed performance significantly degrades for more connected graphs (e.g. as larger and larger number of pairs have to be rederived after deletion), to the point that transitive closure maintenance dominates the total execution time of the simulation. IncSCC however takes advantage of SCCs and runs efficiently in all cases, having a negligible impact on the overall runtime of the simulation and Rete maintenance. Local search in VIATRA2 is orders of magnitudes slower than either of the incremental approaches.

Fig. 6 shows the results of the second experiment series. For each model size on the horizontal axis, we have displayed the average number of SCCs in the model, and on the logarithmic vertical axis the total simulation execution times in case of the three solutions. The second measurement series demonstrates that IncSCC has a better complexity characteristic on large models than DRed, while both scale significantly better than LS.

5 Related Work

Dynamic computation of transitive closure While there are several classical algorithms (depth-first search, etc.) for computing transitive reachability in graphs, efficient incremental maintenance of transitive closure is a more challenging task. As transitive closure can be defined as a recursive Datalog query, incremental Datalog view maintenance algorithms such as DRed [26] can be applied as a

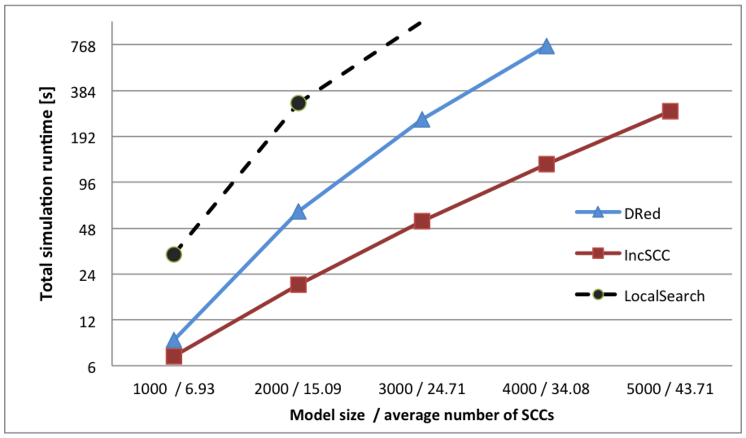


Fig. 6. Performance on increasing model sizes (`addLink` weight is 2)

generic solution. There is also a wide variety [28] of algorithms that are specifically tailored for the fully dynamic[2] transitive reachability problem. Some of these algorithms provide additional information (shortest path, transitive reduction), others may be randomized algorithms (typically with one-sided error); the majority focuses on worst-case charactersitics in case of dense graphs. The spectrum of solutions offers various trade-offs between the cost of operations specified in Sec. 3.1.

Even if the original graph has a moderate amount of edges (*sparse graph*), the size of the transitive closure relation can easily be a quadratic function of the number of vertices, raising the relative cost of maintenance. A key observation, however, is that in many typical cases vertices will form large SCCs. This is exploited in a family of algorithms [12,13] including IncSCC that maintain (a) the set of SCC using a dynamic algorithm, and also (b) the transitive reachability relationship between SCCs. Choosing such an algorithm is justified by simplicity of implementation, the sparse property of typical graph models and the practical observation that large SCCs tend to form.

Incremental pattern matching and transitive closure in graph and model transformation Apart from VIATRA2, GROOVE [2] also features a Rete-based incremental pattern matcher, and is therefore the most closely related work. In fact, the Rete implementation in GROOVE has recently been extended [3] by the capability of incrementally maintaining transitive closure relations. They also introduced a new type of Rete node that accepts a binary relationship as input and emits its transitive closure as output. The transitive closure node in GROOVE implements a simple algorithm that maintains the set of all paths

[2] Note that the graph algorithms community uses the term "fully dynamic" instead of "incremental", as the latter has a secondary, more restrictive meaning in context of the transitive closure maintenance problem.

(walks) of any length that can be composed from the original binary relation, even if many of them are redundant due to having the same sources and targets. This results in factorial time and space complexity, as opposed to the various polynomial solutions found in literature and also in our solution. Their solution is only capable of computing the transitive closures of so called *regular (path) expressions*; we believe our notion of "derived edge" is more general, as it includes arbitrary graph structures (e.g. circular patterns as context, attribute restrictions, etc.). Finally, the experimental assessment in [3] is conducted under substantially different conditions, such as the graph being linear; in contrast, our solution proves to be scalable for non-linear graphs as well.

In the future, we would like to carry out experimental comparison of the transitive closure features of GROOVE and VIATRA2. This will need significant additional effort, as the running example of our current paper relies on a complex peer-to-peer model and a stochastic simulator engine that would be difficult to replicate on GROOVE, while the case study example in [3] relies on model checking capabilities that are not supported in VIATRA2.

Some other graph transformation tools [29,30] feature path expressions, including transitive closure, without maintaining the result incrementally. In a graph with a low branching factor, they can still be feasible in practice. There are other model transformation tools that offer incremental evaluation. The incremental tranformation solution in ATL [31] relies on impact analysis of OCL expressions, meaning that the entire OCL expression will be re-evaluated whenever a relevant element in the model changes; however standard OCL cannot express transitive closure in arbitrary graphs. There is an incremental evaluation technique for Tefkat [19] that maintains an SLD resolution tree of the pattern expression; but without special handling of transitive closure, the SLD tree expands all possible paths from source to target, leading to factorial complexity similarly to GROOVE.

6 Conclusion

We have presented the extension of the incremental pattern matcher of VIATRA2 with a dedicated capability for maintaining generic transitive closure built on a fully dynamic transitive closure maintenance strategy. The results were evaluated in terms of performance on a P2P stochastic system simulation case study.

Our measurements have shown the performance impact of incrementally evaluating generalized transitive closure to be affordable. This implies that the inclusion of transitive closure based probes and rule guard conditions is feasible and scalable in stochastic model simulation, even in case of dynamically changing graph structures. As for the performance of transitive closure algorithms, our investigation demonstrated the overall superiority of *IncSCC* in a wide range of model structures.

As future work, we plan to (i) conduct more detailed benchmarking in other scenarios, (ii) integrate transitive closure maintenance into the EMF-based EMF-INCQUERY [10], and (iii) investigate additional transitive closure algorithms.

References

1. Bergmann, G., Ákos Horváth, Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds.: Graph Transformations. Volume 5214 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 396–410
2. Ghamarian, A.H., Jalali, A., Rensink, A.: Incremental pattern matching in graph-based state space exploration. Electronic Communications of the EASST (2010) GraBaTs 2010, Enschede.
3. Jalali, A., Ghamarian, A.H., Rensink, A.: Incremental pattern matching for regular expressions. In: 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2012)
4. Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.C., Rummler, A., Sousa, A.: A model-driven traceability framework for software product lines. Software and Systems Modeling **9** (2010) 427–451 10.1007/s10270-009-0120-9.
5. Kovács, M., Gönczy, L., Varró, D.: Formal analysis of bpel workflows with compensation by model checking. International Journal of Computer Systems and Engineering **23**(5) (2008)
6. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3) (2007) 214–234
7. Madari, I., Lengyel, L., Mezei, G.: Incremental model synchronization by bi-directional model transformations. In: Computational Cybernetics, 2008. ICCC 2008. IEEE International Conference on, IEEE (2008) 215–218
8. Torrini, P., Heckel, R., Ráth, I., Bergmann, G.: Stochastic graph transformation with regions. ECEASST **29** (2010)
9. Horváth, A., Bergmann, G., Ráth, I., Varró, D.: Experimental assessment of combining pattern matching strategies with VIATRA2. International Journal on Software Tools for Technology Transfer (STTT) **12** (2010) 211–230 10.1007/s10009-010-0149-7.
10. Bergmann, G., Horváth, A., Ráth, I., Varró, D.: Incremental evaluation of model queries over EMF models. In Petriu, D., Rouquette, N., Haugen, Ø., eds.: Model Driven Engineering Languages and Systems. Volume 6394 of LNCS. Springer Berlin / Heidelberg (2010) 76–90
11. Proctor, M., et al.: Drools Documentation. JBoss. `http://labs.jboss.com/drools/documentation.html`.
12. Poutré, J.A.L., van Leeuwen, J.: Maintenance of transitive closures and transitive reductions of graphs. In: Graph-Theoretic Concepts in Computer Science, International Workshop, WG '87. Volume 314 of Lecture Notes in Computer Science., Springer (1988) 106–120
13. Frigioni, D., Miller, T., Nanni, U., Zaroliagis, C.: An experimental study of dynamic algorithms for transitive closure. ACM JOURNAL OF EXPERIMENTAL ALGORITHMICS **6** (2000) 2001
14. Torrini, P., Heckel, R., Ráth, I.: Stochastic simulation of graph transformation systems. In Rosenblum, D., Taentzer, G., eds.: Fundamental Approaches to Software Engineering. Volume 6013 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 154–157; DOI: 10.1007/978-3-642-12029-9_11.
15. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer (2006)
16. Khan, A., Heckel, R., Torrini, P., Ráth, I.: Model-based stochastic simulation of P2P VoIP using graph transformation. In: Proceedings of the 17th International

Conference on Analytical and Stochastic Modeling Techniques and Applications. (2010)

17. Varró, G., Varró, D.: Graph transformation with incremental updates. In: Proc. GT-VMT 2004, International Workshop on Graph Transformation and Visual Modelling Techniques. Volume 109 of ENTCS., Elsevier, Elsevier (2004) 71–83
18. Varró, G., Varró, D., Schürr, A.: Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In: Graph and Model Transformation (GraMoT 2006). Volume 4 of Electronic Communications of the EASST., EASST (2006)
19. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 321–335
20. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**(1) (September 1982) 17–37
21. Torrini, P., Ráth, I.: GraTS: graph transformation-based stochastic simulation — Documentation. (2012) `http://viatra.inf.mit.bme.hu/grats`.
22. Khan, A., Torrini, P., Heckel, R.: Model-based simulation of VoIP network reconfigurations using graph transformation systems. In Corradini, A., Tuosto, E., eds.: Intl. Conf. on Graph Transformation (ICGT) 2008 - Doctoral Symposium. Volume 16 of Electronic Communications of the EASST. (2009) http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/view/26.
23. Kosiuczenko, P., Lajios, G.: Simulation of generalised semi-Markov processes based on graph transformation systems. Electronic Notes in Theoretical Computer Science **175** (2007) 73–86
24. Cassandras, C.G., Lafortune, S.: Introduction to discrete event systems. Kluwer (2008)
25. D'Argenio, P.R., Katoen, J.P.: A theory of stochastic systems part I: Stochastic automata. Inf. Comput. **203**(1) (2005) 1–38
26. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally (extended abstract). In: Proc. of the Int. Conf. on Management of Data, ACM. (1993) 157–166
27. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing **1**(2) (1972) 146–160
28. Demetrescu, C., Italiano, G.F.: Dynamic shortest paths and transitive closure: algorithmic techniques and data structures. J. Discr. Algor **4** (2006) 353–383
29. Schürr, A.: Introduction to PROGRES, an attributed graph grammar based specification language. In Nagl, M., ed.: Graph–Theoretic Concepts in Computer Science. Volume 411 of LNCS., Berlin, Springer (1990) 151–165
30. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: The 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, ACM Press (2000)
31. Jouault, F., Tisi, M.: Towards incremental execution of ATL transformations. In Tratt, L., Gogolla, M., eds.: Theory and Practice of Model Transformations. Volume 6142 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 123–137 10.1007/978-3-642-13688-7_9.