Benchmarking for Graph Transformation*

Gergely Varró[1,2], Andy Schürr[2], Dániel Varró[3]
gervarro@cs.bme.hu, andy.schuerr@es.tu-darmstadt.de, varro@mit.bme.hu

[1] *Department of Computer Science and Information Theory*
Budapest University of Technology and Economics

[2] *Real-Time Systems Lab*
Data Systems Technology Institute
Technical University of Darmstadt

[3] *Department of Measurement and Information Systems*
Budapest University of Technology and Economics

Abstract

Up to this point there did not exist any collection of benchmarks for comparing different tools in the graph transformation area. The aim of this paper is to bridge this gap and to provide a description of a basic set of benchmark examples together with scenarios for which the benchmarks can be used. Moreover, our initiative includes a quantitative comparison of the performance of graph transformation tools by defining certain parameter settings and optimization possibilities for different test cases that are requested to be implemented by tool providers.

1 Introduction

Benchmarking has a key role in decision making processes when a choice has to be made between several alternatives. In order to fill this role, system designers should get a proper view on the system, which means that characteristics of the system have to be measured under different circumstances (i.e., by using several parameter combinations for measurements).

Graph transformation [4, 11] provides a pattern and rule based manipulation of graph models. Since there is a couple of fields where graph based models can be used, graph transformation can be considered as a widely applicable approach. However, despite the large variety of graph transformation tools (AGG [5], Fujaba [6], Great

*This work was partially carried out during the visit of the first author to the Technical University of Darmstadt (Germany), and it was funded by the SegraVis Research Training Network.

[1], Groove [10], Progres [13], Viatra [17]), up to this point there did not exist any collection of benchmarks for comparing such tools.

Objectives. The aim of this paper is to bridge this gap and to provide a description of a basic set of benchmark examples together with scenarios for which the benchmarks can be used. Moreover, our initiative includes a quantitative comparison of the performance of graph transformation tools by defining certain parameter settings and optimization possibilities for different test cases that are requested to be implemented by tool providers.

In case of graph transformation benchmarks the sole measurable feature, which composes the base of comparison in turn, is the execution time of pattern matching and updating phases. (Note that the time needed for generating the initial models does not take part in measurements, and thus, this topic is not discussed in this paper.) Execution times are measured for several tools and on different test sets, while the underlying hardware remains the same for all benchmarks.

Related work. Benchmarking is a well-known approach from different fields of computer science.

- In [18], we proposed a relational database solution for performing graph transformation. Since other tools use different techniques, database benchmarks are irrelevant, nevertheless there exist several benchmarks [2, 7, 14] in this field.
- There are already some benchmarks created for rule-based expert systems (for details see [3]). They include the following problems.
  - Manners handles the problem of finding an acceptable seating arrangement for guests at a dinner party.
  - Waltz is a diagram labeling problem and it analyzes the lines of a 2-dimensional drawing, and labels them as if they were edges in a 3-dimensional object.
  - Aeronautical Route Planner (ARP) plots a course over a given terrain from point 1 to point 2, for a plane or missile.
  - The Weaver program is an expert system, which is composed of many other expert systems that communicate through a common black board. It is used to do VLSI routing for channels and boxes, and with over six hundred rules it is the largest benchmark among the above-mentioned benchmarks.

  As these benchmarks have been created for rule-based expert systems, no graphical rule and model description is possible. Moreover, according to the benchmark descriptions, the largest test set (i.e., the Weaver program) has about 600 rules, 1900 model elements and the transformation sequence consists of only 100 activations. However, the biggest problem with these benchmarks is that they do not cover the typical application areas of graph transformation. In their present form, they are unusable for graph transformation, and their adaptation would be cumbersome.

We have already launched our research in graph transformation benchmarking in [18], but that paper mentioned only one test set without any details, while our current aim is to provide complete and detailed descriptions for several benchmarks.

Overview of terminology. The *metamodel* describes the abstract syntax of a modeling language. Formally, it can be represented by a type graph. Nodes of the type graph are called *classes*. A class may have attributes that define some kind of properties of the specific class. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra *attributes*. Finally, *associations* define connections between classes.

The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as *slots* in the instance model. Inheritance in the instance model imposes that instances of the subclass can be used in every situation, where instances of the superclass are required.

Example. In order to present our concepts, the metamodel of the mutual exclusion problem (depicted in Fig. 1(a)) can be examined. It has only two classes, which are called Process and Resource. These classes are connected by edges of type next, request, held_by, release, token, and blocked, which correspond to associations in turn. This metamodel does not define any attributes. Similarly, no inheritance is specified in the figure.

A well-formed instance model of this domain is shown e.g., in Fig. 3(a). It has two processes (p1 and p2) and two links (n1 and n2) of type next.

A *graph transformation rule* $\mathsf{r} = (\mathsf{LHS}, \mathsf{RHS}, \mathsf{NAC})$ contains a left-hand side graph LHS, a right-hand side graph RHS, and negative application condition graphs NAC.

The *application* of a rule r to a *host (instance) model M* replaces a matching of the LHS in M by an image of the RHS. This is performed by (i) finding a matching of LHS in M (by graph pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain objects and links) (iii) removing a part of the model M that can be mapped to LHS but not to RHS yielding to the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the *derived model* M'. A *graph transformation* is a sequence of rule applications from an initial model M_I.

Example. The mutual exclusion algorithm can be described by 13 graph transformation rules as it is presented in Fig. 2. By choosing newRule (Fig. 2(a)) as our running example, we may state that in its LHS a link of type next connects two processes, while in its RHS the same processes (p1 and p2) also appear, but now there is also a third process p, which is placed between p1 and p2. The newRule does not have any negative application conditions.

This rule can be applied on the model that has been presented in Fig. 3(a). If p1 and p2 of the newRule are mapped to p1 and p2 of the model, respectively, in the pattern matching phase then the definition of rule application prescribes the deletion of n2 followed by the insertion of p3, n3 and n4. If the same rule is applied again but

now with an inverse mapping (p1 of newRule is mapped to p2 and vice versa), then the model of Fig. 3(b) is resulted.

The structure of the paper. In Sec. 2 an overview is given on the most common scenarios in graph transformation. Then we present in a tabular form which benchmark can be used for which scenarios. In this table benchmarks whose implementation is requested from tool providers are also marked. In the rest of the paper benchmarks are presented on a section by section basis. Section 3 introduces a benchmark example, which is typical for checking the specification of a system that is defined in a visual language with dynamic semantics. The benchmark of Sec. 4 is a model transformation example. In Sec. 5 a special test set is presented which is appropriate for testing only the pattern matching phase, but with different model and pattern sizes.

2 Benchmark features

Benchmarks can be characterized by different features including the size of the patterns, the maximum degree of nodes (fan-out) in the model, the number of successful matchings of a rule, and the length of the transformation sequence executed during the test. These *generic features* are fully determined by rule and model descriptions of the benchmark, so they are not influenced at all by optimization features of different tools.

On the other hand, based on the application order of rules in a specific test case, tools may perform different optimizations. Features, which are affected by optimizations, are called *tool-dependent*. Four tool-dependent features are identified initially.

- In case of *parallel rule execution*, all the matchings of a rule are calculated in the pattern matching phase, and then updates are performed in a transaction block on the collected matchings without re-evaluating valid matchings during the transaction.

- *'As long as possible' rule application* means an iterative execution of the selected rule for which the termination of the iteration is declared and guaranteed by the system designer.

  A standard graph rewriting step (with a pattern matching and an updating phase) is performed in each iteration. Thus, in order to avoid infinite loops, it must be ensured that the number of matching patterns always decreases, which, in addition, forms a sufficient guarantee for termination.

- The term *multiplicity based optimization* is used, when a tool employs a different (and usually simplified) strategy in order to find matching model elements for an edge with bounded multiplicity.

- Tools may provide a *parameter passing* possibility between consecutive rule applications. By passing model elements as parameters, pattern matching may be facilitated in the subsequent rewriting steps, since passed elements can be reused directly without performing any recalculation in these later steps.

The above-mentioned enumeration cannot be complete, since it does not contain any heuristics that is going to be discovered in the future and it further ignores features that are specific only for a single tool. In case of these heuristics tool providers are asked to prepare both an optimized and an unoptimized version of the system for the benchmark where the effects of the optimization are the most recognizable.

Generic features		**Mutex**			**OR mapping**	**Comb**	
		Short TS	Long TS	ALAP execution	Simple model	Several matchings	No matching
LHS size	large	-	-	-	+	PD	PD
	small	+	+	+	-	PD	PD
fan-out	large	PD	-	-	-	-	-
	small	PD	+	+	+	+	+
matchings	many	PD	PD	PD	-	+	-
	few	PD	PD	PD	+	-	+
transformation sequence length	long	-	+	-	-	-	-
	medium	-	-	+	-	-	-
	short	+	-	-	+	+	+

Tool-dependent optimization	**Mutex**			**OR mapping**	**Comb**	
	Short TS	Long TS	ALAP execution	Simple model	Several matchings	No matching
parameter passing	***REQ***	+	-	-	-	-
0..1 multiplicities	***REQ***	+	+	+	+	+
parallel execution	+	+	***REQ***	+	-	-
as long as possible	-	+	+	+	-	-

Table 1: Feature matrices of test cases and scenarios

Table 1 presents a feature matrix describing what purpose a certain test case can be useful for. Upper and lower parts of Table 1 show generic and tool-dependent features, respectively. If the given feature is characteristic for the test case then it is denoted by a plus sign (+). A minus sign (–) represents the case when the feature is not characteristic for the test set. If the characteristics of a feature depends on the concrete parameter settings, then it is called parameter dependent (marked by letters PD).

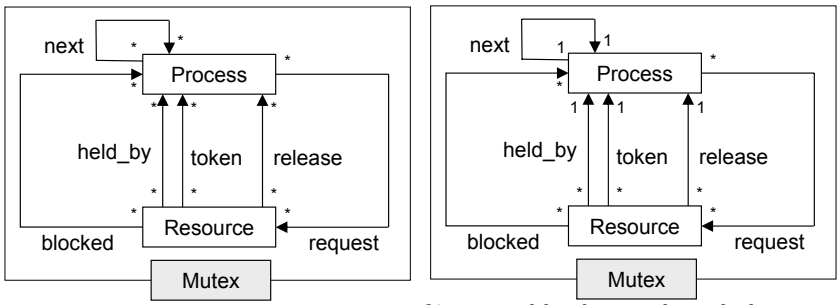
In case of tool-dependent features a plus sign has an additional meaning. It still denotes a feature of the test case for which optimization can be done, but in order to minimize coding efforts only one (in general the unoptimized) version of the solution is needed for the tool comparison. Notation REQ again denotes a characteristic feature; but in this case both optimized and unoptimized versions are required in order to be able to compare the effects of optimization. Detailed requirements for comparison are discussed at the end of test set descriptions (see **Optimization possibilities**). In the tool-dependent group minus signs (–) still represent non-characteristic features, so in these cases it is totally meaningless to use heuristics.

3 Distributed mutual exclusion algorithm

This benchmark is a distributed mutual exclusion algorithm whose full specification can be found in [8]. The algorithm is defined in a visual language with dynamic semantics. The scenario can be characterized by a nearly static graph structure, where

only tokens are passed around, and by short rewriting sequences that are respected for a long time. These rule application sequences (which are defined for test cases) describe possible behaviours of the system in different situations.

Fig. 1(a) presents the unoptimized metamodel of the domain, in which all edge multiplicities are of zero-to-many kind. Tools performing multiplicity based optimization heuristics may use the optimized metamodel (depicted in Fig. 1(b)) for this benchmark.



(a) Metamodel with zero-to-many multiplicities on all edges

(b) Metamodel with some edge multiplicities optimized

Figure 1: Metamodels for the mutual exclusion problem

Processes try to access shared resources. One requirement of the algorithm is to give access to each resource by at most one process at a time. This is achieved by using a token ring, which consists of processes connected by edges of type next. In the consecutive phases of the algorithm, (i) a process may issue a request on a resource, (ii) the resource may eventually be held by a process and finally (iii) a process may release the resource. The right to access a resource is modeled by a token. The algorithm also contains a deadlock detection procedure, which has to track the processes that are blocked.

The algorithm can be described by 13 graph transformation rules as presented in Fig. 2. The most complex rule (blockedRule in Fig.2(j)) has 4 nodes and 3 edges.

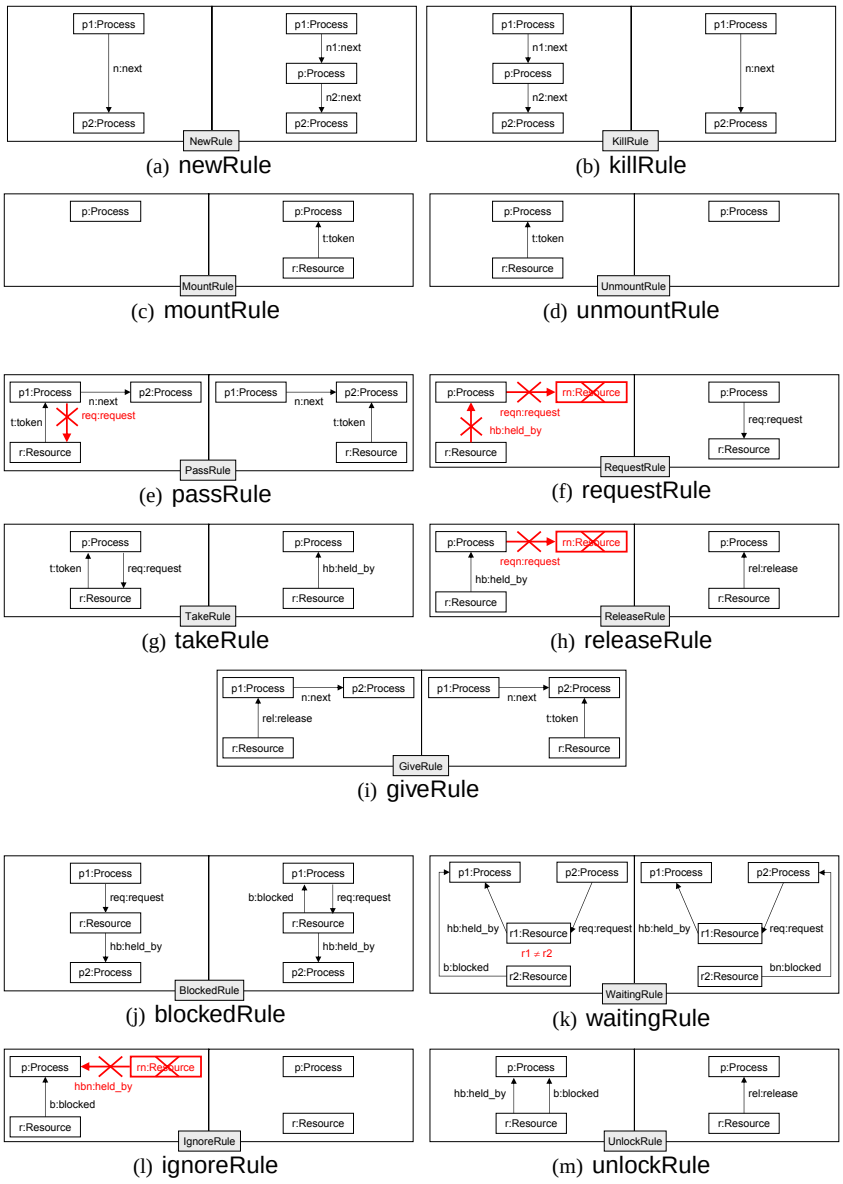
3.1 Short transformation sequences

This test case can be characterized by small LHS graphs and short transformation sequences. The number of fan-outs of model nodes and of matchings are parameter dependent, so they are not distinguishing features of this test case.

Initial instance graphs in this test set only contain two process nodes and two edges of type next linking the process nodes in both directions (as it is presented in Fig. 3(a)). The test set has one parameter N, which denotes the maximum number of processes appearing in the instance model during a specific test.

The transformation sequence can be described as follows.

1. The newRule (Fig. 2(a)) is applied first N–2 times in an arbitrary order. Since each application of newRule adds a process to the token ring, after this step the



(a) newRule (b) killRule

(c) mountRule (d) unmountRule

(e) passRule (f) requestRule

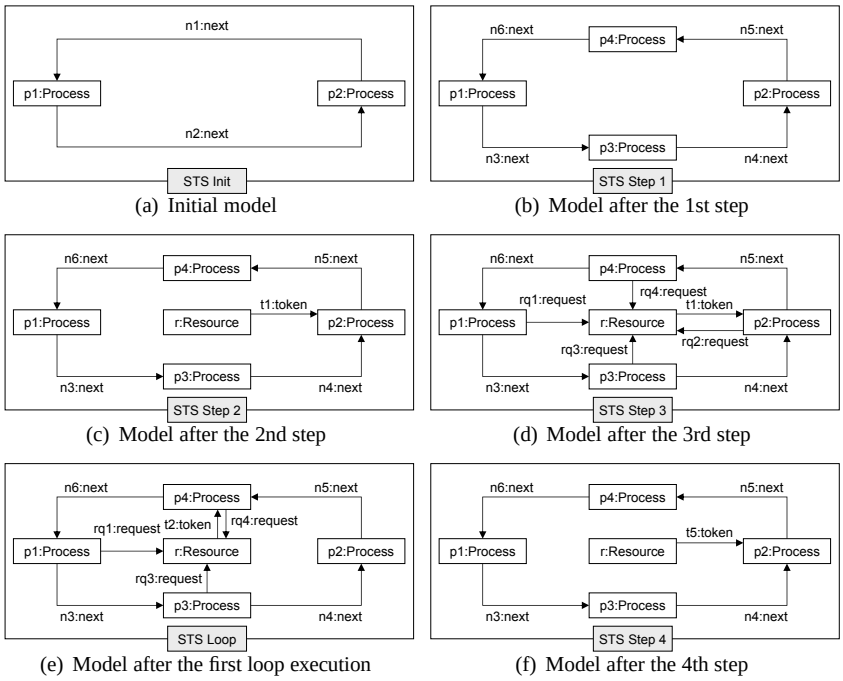
(g) takeRule (h) releaseRule

(i) giveRule

(j) blockedRule (k) waitingRule

(l) ignoreRule (m) unlockRule

Figure 2: Rules describing the mutual exclusion algorithm



(a) Initial model (b) Model after the 1st step

(c) Model after the 2nd step (d) Model after the 3rd step

(e) Model after the first loop execution (f) Model after the 4th step

Figure 3: Models in different phases of short transformation sequence

instance model will have a ring structure consisting of exactly N process nodes that are linked by N edges of type next as shown in Fig. 3(b).

2. The second step is to create a single resource by performing the mountRule (Fig. 2(c)) once. This rule also gives access rights to one of the processes, which is modeled by a newly created token edge. The instance model is shown in Fig. 3(c).

3. In the third step, each process issues a request for the single resource, which means the execution of requestRule (Fig. 2(f)) for N times. Regardless of the execution order, the final instance model will be the one that is presented in Fig. 3(d). (So it is possible to apply requestRule in parallel.)

4. The final step handles the requests that have been issued in the previous step. To handle a single request rules takeRule, releaseRule and giveRule have to be applied in this specific order. In order to speed up pattern matching, *parameter passing* is possible among the rules that belong to the same loop.

   TakeRule (Fig. 2(g)) assigns the process with the token to the resource by creating a held_by edge. Then releaseRule (Fig. 2(h)) lets the resource to be released by the process. Finally, the resource is released and the token is propagated to the next process in the token ring by the execution of giveRule (Fig. 2(i)). The instance model we have at this point is shown in Fig. 3(e).

   Since all the N processes have already requested the resource, the above-mentioned 3 rules have to be executed in a loop for N times, which results in a rule execution sequence of length $3N$. (Note that there exists only a single matching to which the subsequent rule can be applied at the time when the rule application is scheduled, so the rule execution order of the fourth step is fully deterministic.) In the end, the instance model will be the one that is depicted in Fig. 3(f).

The transformation sequence consists of $5N$–1 rule applications altogether. The largest instance graph that appears during the rule application phase has N+1 nodes and $2N$+1 edges (see Fig. 3(d)). N was chosen as 5, 100, and 1000 in our different experiments resulting in models of size 17, 302, and 3002, and transformation sequences of length 24, 499, and 4999, respectively.

Optimization possibilities and requirements.

- Instead of having zero-to-many multiplicities on all association ends, it is possible to restrict some of them to zero-to-one, as it is presented in Fig.1. Since the model contains only a single resource, knowing and using this fact may cause performance improvements for some tools, since pattern matching can be started at this well-defined node.

- As it was already mentioned in the test case description, the 3 rules in the loop of the fourth step may be applied in such way that the selected process and resource nodes can be passed to consecutive rules as parameters, which may speed up pattern matching.

In order to perform a wide range comparison tool providers are asked to prepare both an optimized and an unoptimized version of their solution for this test case. Since there are two independent optimization possibilities, this results in at most 4 different rule sets. (The possibility to set parameter N should also be provided as well.)

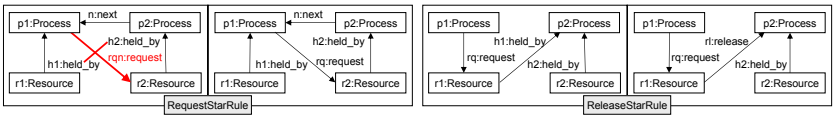
Despite the fact that parallel rule application is also possible in the third step, tool providers are asked to generate only an unoptimized version, where rewriting is executed sequentially.

The widest range comparison could certainly be achieved by performing isolated measurements for each combination of optimization possibilities, but it would unnecessarily increase the required efforts, and thus, it is avoided by measuring the effects of a given optimization heuristics only in a single test where its influence on performance is really representative. This effort minimization is the reason for using only one version. The unoptimized, sequential version is selected from the two alternatives, because it is surely supported by all tools.

3.2 Long transformation sequences

This test case can be characterized by small LHS graphs, small number of fan-outs of model nodes and long transformation sequences. The number of matchings is again parameter dependent, so it is not a distinguishing feature of this test case.

For this test set, we modified two rules (namely, requestRule and releaseRule of [8]) in order to restrict their applicability in certain situations and to get a deterministic transformation sequence. The modified rules are referred to as requestStarRule and releaseStarRule and are depicted in Fig. 4(a) and in Fig. 4(b), respectively,

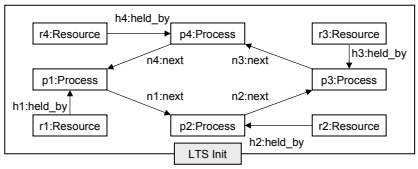


(a) requestStarRule (b) releaseStarRule

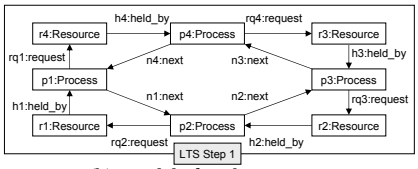
Figure 4: Extra rules for the long transformation sequence

In this case, we have two parameters (namely, N and R). N denotes the number of processes and resources in the initial instance model, and it influences both the model size and the length of the transformation sequence. We refer to a transformation sequence as *a basic execution unit*, if (i) instance graphs before and after execution are isomorphic, and (ii) the sequence can be executed several times in a loop. The role of R is to determine how many times a basic execution unit is executed during the test. As a consequence, R has influence only on the length of the transformation sequence.

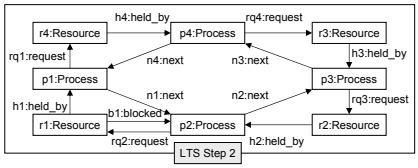
The initial instance model now consists of $2N$ nodes (N processes and N resources) and $2N$ edges. N edges are of type next and they are used to organize process nodes into a token ring. The other N edges mark processes holding resources in such a way that no held by edges have common ends (i.e., each resource is held by at most one process and each process reserves at most one resource). A sample initial instance model is presented in Fig. 5(a) for the $N = 4$ case.



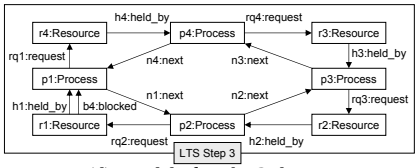
(a) Initial model with parameter $N = 4$



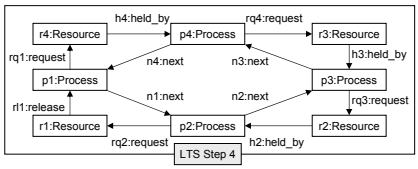
(b) Model after the 1st step



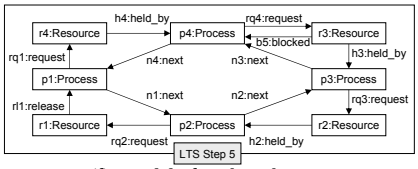
(c) Model after the 2nd step



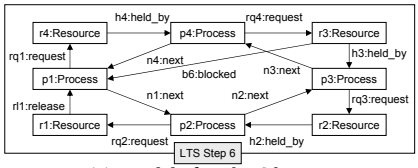
(d) Model after the 3rd step



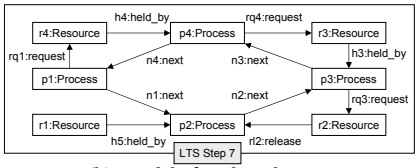
(e) Model after the 4th step



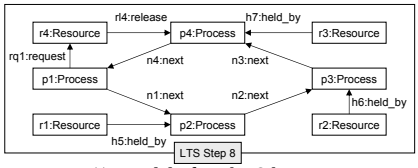
(f) Model after the 5th step



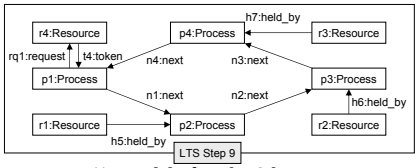
(g) Model after the 6th step



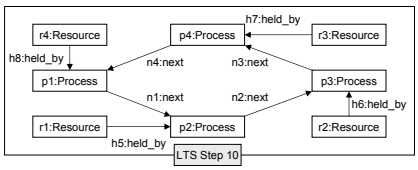
(h) Model after the 7th step



(i) Model after the 8th step



(j) Model after the 9th step



(k) Model after the 10th step

Figure 5: Models in different phases of long transformation sequence

The transformation sequence inside the basic execution unit is defined as follows.

1. As a first step, requestStarRule (Fig. 4(a)) is applied N times. RequestStarRule selects two neighboring processes holding each at least one resource, and the one that is ahead in the token ring, issues a request on the resource that is held by the other process, if it has not issued any requests yet on the same resource. The resulting instance model (see Fig. 5(b)) should be identical after any sequence of rule applications during the first step, so this set of rules can be applied in parallel.

2. This step is a single execution of a blockedRule (Fig. 2(j)), which initiates the deadlock detection algorithm by introducing a new blocked edge. There are N matchings for this rule before its application, so the graph transformation engine can choose freely on which matching the concrete rule is applied. The result of the rule application is something similar to Fig. 5(c).

3. The waitingRule (Fig. 2(k)) is executed now N–1 times. Since the model contains only a single blocked edge, this sequence is fully deterministic. Moreover, it describes how the blocked edge is propagated in the token ring of processes in the same direction that is marked by the set of next edges. After this step, the blocked edge makes a whole round in the token ring as it is depicted in Fig. 5(d).

4. Now a single execution of the unlockRule (Fig. 2(m)) follows, which can be done only on a single matching. This breaks the circular blocking situation that causes deadlock, by forcing a process to release its resource. The result will be a model that is shown in Fig. 5(e).

5. In the fifth step, the blockedRule (Fig. 2(j)) is executed once again, generating a new blocked edge. In this case, the rule can be applied on possible N–1 matchings. Since this is a nondeterministic choice, the result will be something similar to Fig. 5(f).

6. Now the waitingRule (Fig. 2(k)) is applied at most N–1 times. There exists only a single matching on which next rule application can be performed until the point, when the blocked edge points to the same process as the release edge (see Fig. 5(g)). From that point, no matchings can be found. The ratio of successful and unsuccessful rule application steps depends on the context on which the previous blockedRule was executed.

7. The ignoreRule (Fig. 2(l)) is executed once to restore the instance model that we had after the fourth step (Fig. 5(e)) by deleting the blocked edge.

8. The eighth step is an execution of a loop that contains giveRule, takeRule and releaseStarRule in this specific order. The first execution of the loop cycle yields the model of Fig. 5(h). In order to accelerate pattern matching parts of successful matches can be passed as parameters to the successive rule of the loop cycle.

   GiveRule (Fig. 2(i)) releases a resource that was held by a process, and gives the token to the next process in the ring. During the execution of a takeRule

(Fig. 2(g)), the process that has a token for a requested resource, reserves it by introducing a held_by edge between them. The releaseStarRule handles the release of a resource in a special context to ensure a deterministic execution order.

The loop is executed N–1 times altogether. Note that the cardinality of matchings of giveRule is decreased by one after each loop execution. The resulting model we get after the eighth step is presented in Fig. 5(i).

9. In the ninth step giveRule is performed once on the single matching that still exists, resulting in a model that is depicted in Fig. 5(j).

10. The final step is a single takeRule application again on the only possible matching, and the result (shown in Fig. 5(k)) will be isomorphic with Fig. 5(a). The single difference is that now each resource is held by the process that is one step ahead of the one that reserved the resource before the basic execution unit started.

A basic execution unit contains a transformation sequence of length $6N$+1. During the execution of such a basic unit the instance graph had exactly $2N$ nodes and at most $3N$+1 edges as can be seen in Fig. 5(c). This unit was executed R times in our experiments resulting in the same upper bound for the model size and a transformation sequence of length of $R(6N{+}1)$.

Concrete values of parameters were $N = 4$ and $R = 100$ in one case, resulting in a model with 8 nodes and 13 edges and a transformation sequence of length 2500. In the other case N had a value 1000, and R was equal to 1, which yielded a model of size 5001 and a transformation sequence of length 60001.

Optimization possibilities. There are test case specific optimization possibilities in the first and eighth step, but again, in order to minimize efforts, tool providers are asked to prepare only the unoptimized version (based on the metamodel presented in Fig. 1(a), using no parallelism and parameter passing) of their solution.

3.3 As long as possible rule application

This test case can be characterized by small LHS graphs, small number of fan-outs of model nodes and transformation sequences of medium length. The number of matchings is again parameter dependent, so it is not a distinguishing feature of this test case.

RequestRule has to be slightly modified again to ensure the appropriate behavior during the execution of this test set. The modified requestRule will be referred to as requestSimpleRule and is depicted in Fig. 6.

This test set uses N as its single parameter and it determines both the model size and the length of the transformation sequence. More precisely, N denotes both the number of processes and resources in the system.

The initial instance model consists of $2N$ nodes (N processes and N resources) and $2N$ edges again. N edges are of type next and they are used to organize process nodes into a token ring. The other N edges denote processes holding resources in such a way

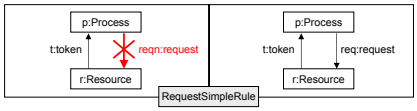
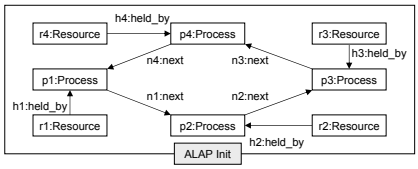
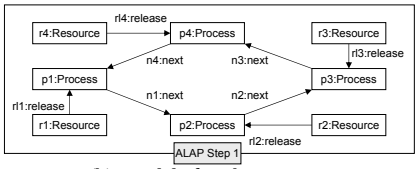


Figure 6: Simplified version of requestRule

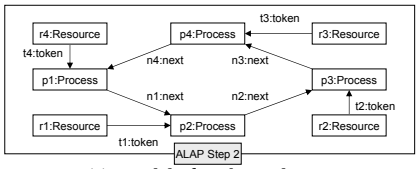
that no held by edges have common ends (i.e., each resource is reserved by at most one process and each process holds at most one resource). A sample initial instance model is presented in Fig. 7(a) for the $N = 4$ case.



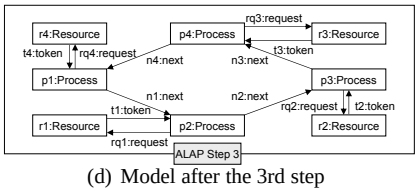
(a) Initial model with parameter $N = 4$



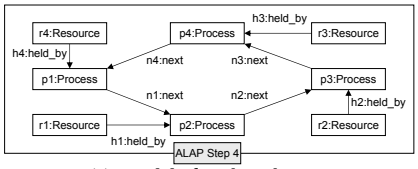
(b) Model after the 1st step



(c) Model after the 2nd step



(d) Model after the 3rd step



(e) Model after the 4th step

Figure 7: Models in different phases of 'as long as possible' rule execution

The test sequence consists of 4 major steps.

1. During the first step, releaseRule is executed N times, resulting in a model (see Fig. 7(b)) where all the resources are now linked to their corresponding processes via a release edge.

2. Then the execution of giveRule follows, which is performed N times. This rule application enables the next process in the ring to reserve the resource by giving the token to the process. The model looks like the one in Fig. 7(c) after this step.

3. The third step consists of N requestSimpleRule applications, which initiates a process to issue a request on the resource for which the process already has a token. As a result, we get the model of Fig. 7(d).

4. Finally, takeRule is executed N times. This rule makes the assignment of a process to a resource, if the process has already a token for the requested resource. The final instance model is again isomorphic to the initial model. The only difference is that in the final model, a certain resource is held by a process that is one step forward in the token ring (see Fig. 7(e)).

This test sequence has two special properties.

- Since the order of rule applications in a major step is irrelevant, the specific rule can be applied concurrently (in parallel) on different processes.
- Moreover, each rule application of a major step (i) disables the execution of the same rule on the same process, (ii) it leaves the enabledness of the same rule on other processes unchanged, and finally, (iii) it enables the execution of the following rule on the same process. These observations yield an 'as long as possible' style application of rules appearing in the same major step.

This test sequence produced models of size $5N$, which were 50, 150, 250, 500, and 1000 in the concrete runs.

Optimization possibilities and requirements. Tool providers are asked to create two versions of their solution. In the first (optimized) version parallel rule application is required in all situations where parallel execution is possible. In the other (unoptimized) version, no parallelism is allowed. This test set was selected to explore the effects of parallel execution, because it has the largest model size among the benchmarks, for which parallelism appears as a feature.

For all other features the preparation of only the unoptimized version (based on the metamodel presented in Fig. 1(a), without performing any possible optimization techniques for 'as long as possible' style rule application) is required.

4 Object to Schema Mapping

Now a typical model transformation example is presented. In this case, an algorithm is defined by means of graph transformation and it generates a relational database schema from a UML class diagram. The algorithm used in this benchmark performs a standard mapping that can be found in any database textbook (e.g., in [16]).

In order to be able to modify both the source and the target model with graph transformation, a metamodel should be defined that contains the metamodel of both the UML language and the relational database schema. Such an extended metamodel is presented in Fig. 8.

The part of the metamodel that describes the structure of class diagrams is a portion of the standard UML metamodel [15]. A package may consist of classes, associations, and nodes expressing generalization relations. This kind of containment is expressed by element ownership (EO) edges. Classes and associations may have attributes and association ends, respectively, as their classifier features (CF). At the same time, the same set of association ends are connected to classes by structural feature type (SFT)

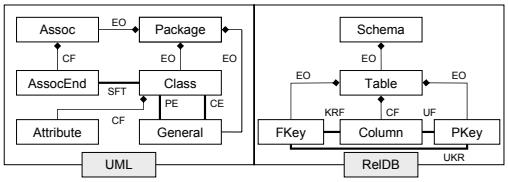


Figure 8: An extended metamodel for the object relational mapping

edges. Generalization connects superclasses to subclasses, which should also be expressed in the UML metamodel. This is achieved by child element (CE) and parent element (PE) edges. Line thickness does not have any additional meaning, the only role of thick lines is to make figures of instance models clearly arranged.

The target language describes the schema of relational databases. In this simplified metamodel, which conforms to CWM [9], schemas may contain tables, and tables may consist of primary key and foreign key definitions. Containing relation is again expressed by element ownership (EO) edges. Columns constitute classifier features (CF) of tables. Foreign keys express key relationships between columns with and without primary keys by using unique key relationship (UKR) and key relationship features (KFR), respectively. Finally, primary keys constitute unique features (UF) of columns.

Furthermore, in order to facilitate the execution of a correct transformation, source and target model nodes should be connected by reference edges, which are marked by dashed lines in figures. Note that in order to get perspicious figures references are not shown when presenting the metamodel.

The whole transformation can be described by 6 rules, which are shown in Fig. 9.

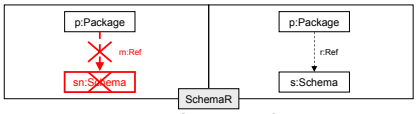
1. SchemaRule (Fig. 9(a)) simply generates a database schema for a UML package.

2. ClassRule searches for a class in the package, for which there does not exist a corresponding table in the database schema, and creates the corresponding table that has a single column tid, for which a primary key tpk is defined.

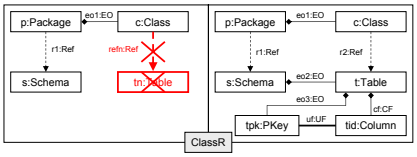
3. AssociationRule creates a new table in the database, if there has not been any table assigned yet. This new table has again a single column rid with a primary key rpk.

4. The AssocEndRule selects an unhandled association end, and generates an additional column relid in the table t_rel that has been created for the association itself. Moreover, a foreign key constraint is added to the t_rel table, which refers to the primary key cpk of the table t_c that is associated with the class.

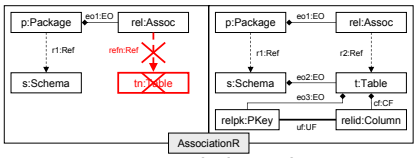
5. The inheritance relation in the UML model is handled by appropriate foreign key constraints in the database schema. This is expressed by the GeneralizationRule, which creates a foreign key constraint on the identifier column subid of the subclass table t_sub for any unhandled generalization node. The constraint will refer to the primary key suppk of the superclass table t_sup.



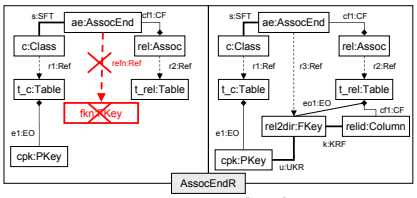
(a) SchemaRule



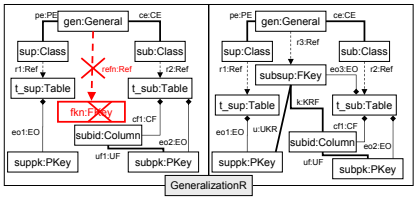
(b) ClassRule



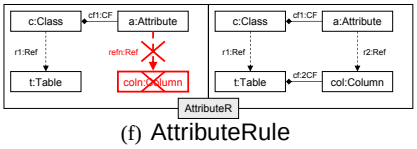
(c) AssociationRule



(d) AssocEndRule



(e) GeneralizationRule



(f) AttributeRule

Figure 9: Rules describing the object relational mapping

6. Finally, a new column is created in the table assigned to the class that includes the unhandled attribute. This is performed by the AttributeRule.

These rules have some special features.

- Note that all the above-mentioned rules have a structure that disables their re-execution on the same matching (context). This observation yields an 'as long as possible' style rule application.
- Moreover, application order of a rule on different matchings is irrelevant, so the specific rule can be applied in parallel.
- The transformation process together with the rules can also be expressed in a more declarative way by using triple graph grammars [12].

4.1 Transformation of a simple class diagram.

This is only an introductory test set that transforms the UML class diagram of the dining philosophers' problem to a corresponding database schema. Its most distinguishing characteristics is the relatively large size of LHS graphs. Since the initial model consists of only a few elements the transformation sequence is short. The number of matchings also depends on the structure of initial model, which yields only few matchings for each rule. The number of fan-outs are node dependent, thus nodes with either small or large number of incident edges can be found.

This test set is without parameter, so the initial instance model is exactly the one that is presented in Fig. 10.

The structures of rules allow 'as long as possible' style rule application, which yields the following transformation sequence.

1. We execute schemaRule as long as possible. In this specific case with having only one package in our initial model, it means a single rule application. The resulting model after this step is depicted in Fig. 11.
2. Since the model contains two classes, classRule is executed twice yielding the situation that is shown in Fig. 12.
3. The class diagram has 3 associations, so the application of associationRule yields an execution sequence of length 3, and to a model presented in Fig. 13.
4. There are 6 association ends to be transformed, so assocEndRule must be executed 6 times. When this step has been finished, the situation of Fig. 14 arises.
5. No inheritance relation appears in the instance model, so generalizationRule is not applied at all in this test set.
6. Finally, the single status attribute is transformed by using the attributeRule as long as possible, which means once in our specific case. As a result, we get an instance model (depicted in Fig. 15) that comprises both the initial UML source model and the corresponding target database schema.

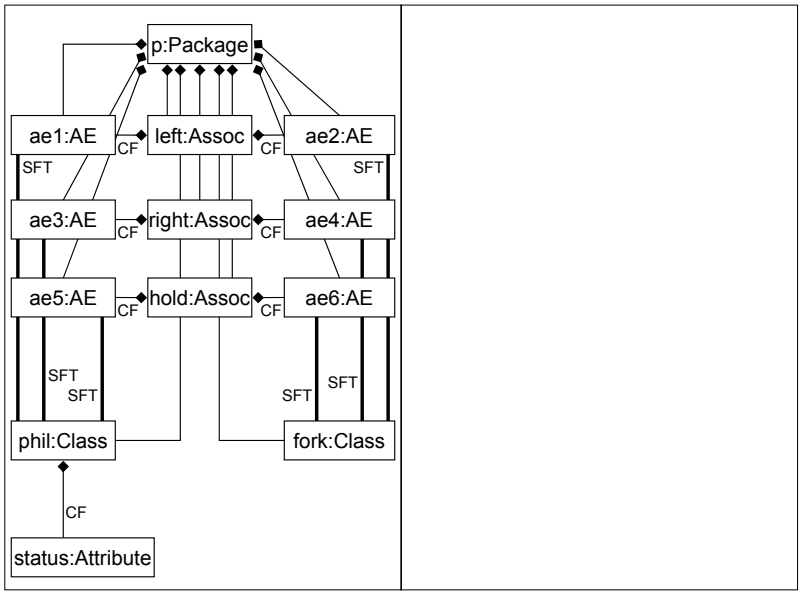


Figure 10: Initial model

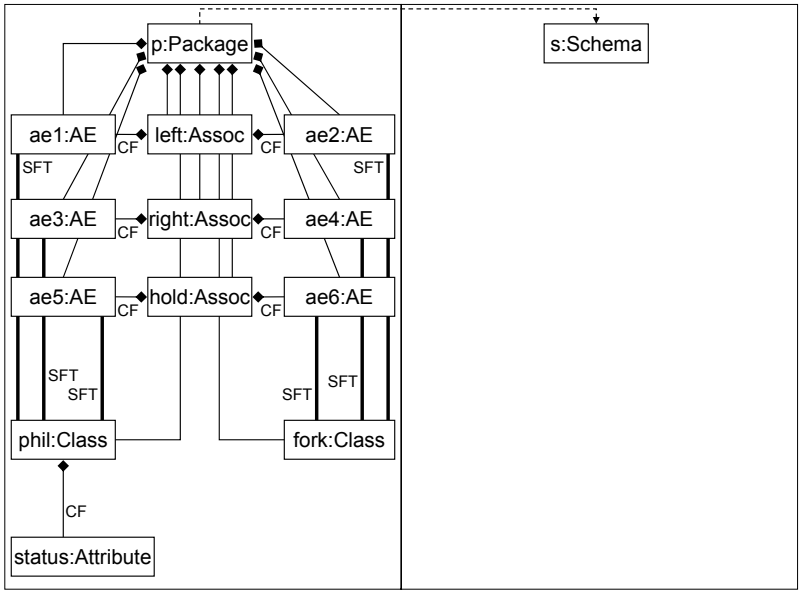


Figure 11: Model after the 1st step

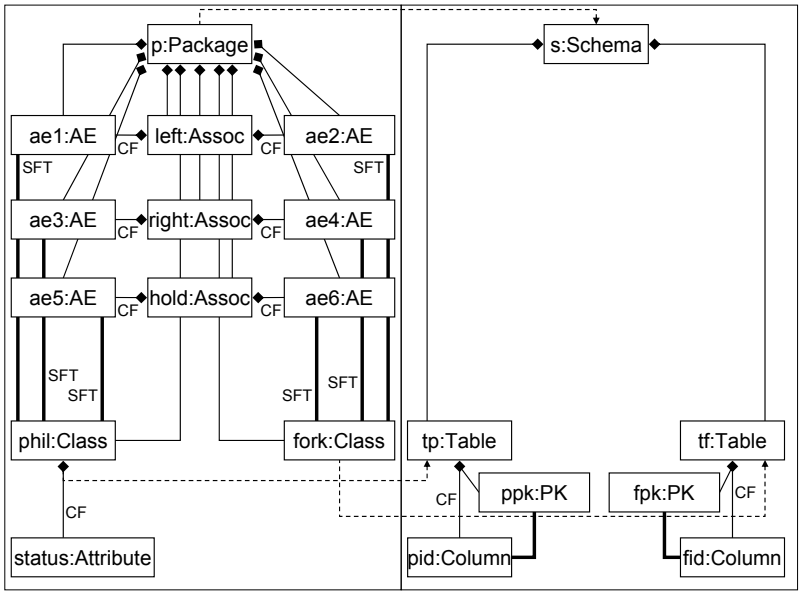


Figure 12: Model after the 2nd step

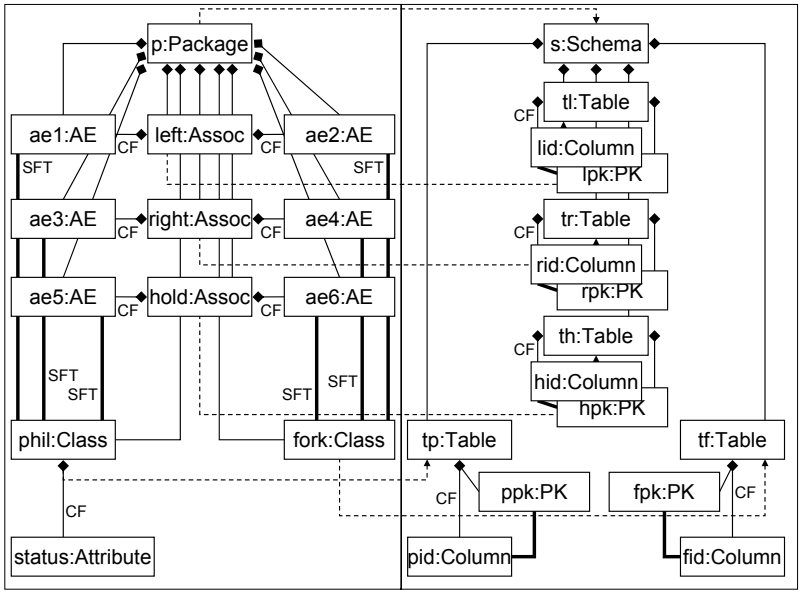


Figure 13: Model after the 3rd step

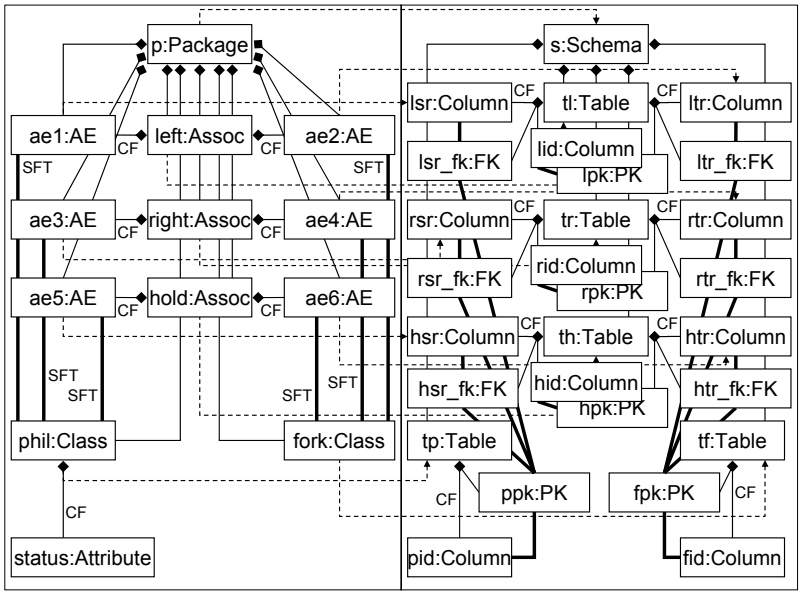


Figure 14: Model after the 4th step

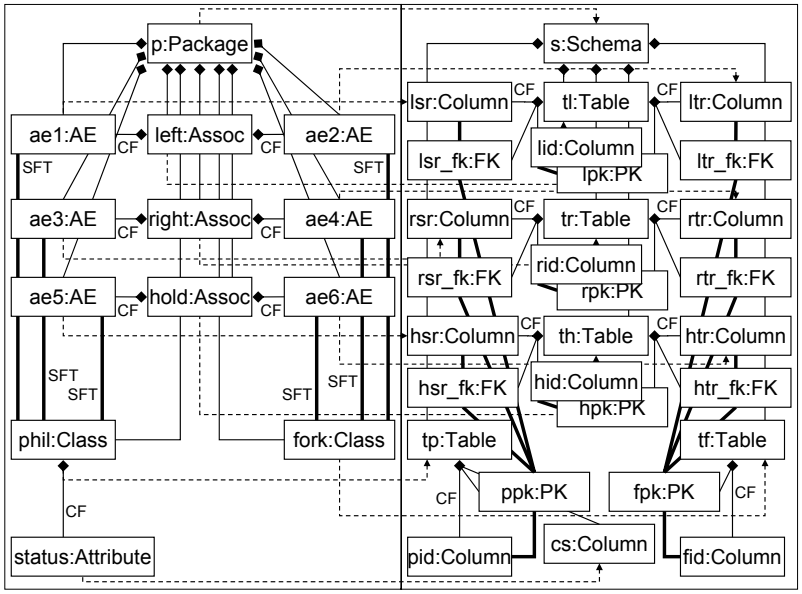


Figure 15: Model after the 5th step

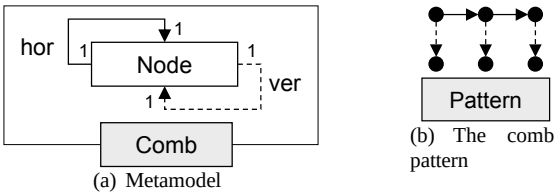
The length of the transformation sequence is fully determined by the number of nodes in the initial model. In our case this is a transformation sequence of length 13. The upper bound for the model size is 124, since the final model contains that many nodes and edges altogether.

Optimization possibilities. Though several optimization techniques can be performed for this test case, benchmark tests are intended to be executed on unoptimized solutions (supposing zero-to-many multiplicities on all edges, applying rules one after the other exactly as it was described without parallel and 'as long as possible' style rule execution).

Despite the fact that rules can be frequently applied in parallel in this benchmark, the third test set of the mutual exclusion algorithm (see Sec. 3.3) has been favoured for examining the effects of parallel execution, since this latter has larger model size. As a consequence, only the sequential, unoptimized version of this benchmark is required.

5 Comb structure

This is a special benchmark, since the left-hand side (LHS) and the right-hand side (RHS) are identical, and as a consequence, performing measurements for the updating phase is meaningless as nothing changes in the model graph. However, this benchmark is perfect for measuring the time needed by the tools to find the first matching or any further matchings of a pattern or to determine that no valid matchings exist. Note that despite all these time values originate from the pattern matching phase, they may significantly differ from each other as tools use diverse strategies in this phase. This benchmark is really appropriate for performing the above-mentioned measurements, since it can be characterized by having a wide range of settings both for model and pattern size and by allowing these size values to be set independently.



(a) Metamodel

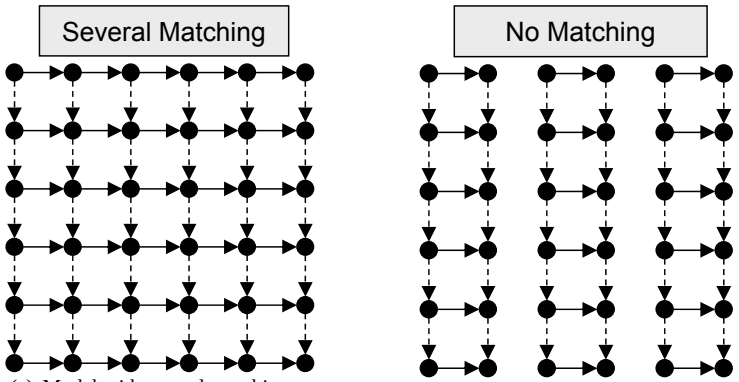
(b) The comb pattern

Figure 16: Metamodel and the comb pattern for $M = 3$

The metamodel that is depicted in Fig. 16(a) is quite simple now, since it contains only a single node type (node) and two edge types. These edge types are for horizontal (hor) and vertical (ver) edges.

The benchmark has M and N as its parameters and they influence the rule size and the model size, respectively. This benchmark has only a single rule with identical LHS and RHS, therefore the rule is free from any side effects (i.e., nothing is modified). The LHS (as shown in Fig. 16(b) for the $M = 3$ case) has a comb-like structure with

having $2M$ nodes arranged in 2 rows and M columns. Nodes belonging to the same column are linked by vertical edges in top-down direction. Furthermore, nodes in the upper row are linked by horizontal edges in left to right direction. The parameter M obviously influences the size of the pattern.



(a) Model with several matchings (b) Model without matchings

Figure 17: Instance models of size $N = 6$ of the comb benchmark

5.1 Several matchings

This test case can be characterized by a large number of matches of the pattern and by a small number of fan-outs of model nodes. Since there are no rules applied in this case, the transformation sequence is short. Model and rule size depend on parameters. The test set is applicable for measuring the time needed for calculating the first and the other consecutive matchings of the same rule. (A graph transformation engine may have significant difference in the calculation time needed for the first matching and for the other matchings.)

The initial instance model (depicted in Fig. 17(a) for the $N = 6$ case) is a grid of $N \times N$ nodes, where neighbouring nodes in horizontal and vertical direction are connected by hor and ver edges, respectively. (Horizontal edges always go from left to right, while vertical edges always point downwards.)

The transformation sequence consists of the above-mentioned single rule, for which all the matchings have to be found.

We may state that the pattern size is $4M$–1 ($2M$ nodes, M vertical edges and M–1 horizontal edges) and the model size is $3N^2$–$2N$, which comprises N^2 nodes, $N(N-1)$ vertical edges and $N(N-1)$ horizontal edges. The number of matchings is $(N-M+1)(N-1)$.

Model and pattern size values and the numbers of matchings for the 4 parameter combinations used in our experiments can be found in Table 2.

Grid size N	Comb width M	**Pattern size #**	**Model size #**	**No of matches #**
100	10	39	29800	9009
100	20	79	29800	8019
30	30	119	2640	29
50	50	199	7400	49

Table 2: Parameter summary

Optimization possibilities. Since setting parameters N and M influences model and rule sizes, the test set already provides a wide range of measurement possibilities, and thus, no optimization is necessitated from tool providers.

5.2 No matching

This test case has no matchings and it can be characterized by a small number of fan-outs of model nodes. No rules are applied again, so the transformation sequence is short. Model and rule sizes are again parameter dependent. The test set can be used for measuring the time needed for a graph transformation engine to determine that a rule is not applicable in a certain situation.

The initial instance model (depicted in Fig. 17(b) for the $N = 6$ case) is again an N by N grid of nodes, where neighbouring nodes in vertical direction are connected by vertical edges. But in this case, neighbouring nodes in horizontal direction are linked to each other if and only if the index of the target node has a non-zero remainder after a division by (M–1). In practical terms, it means that every (M–1)th connection is missing in the horizontal direction resulting in a situation that a comb of width M cannot be placed on this grid. (Horizontal edges always go from left to right again, and similarly vertical edges always point downwards again.)

The transformation sequence consists of the above-mentioned single rule.

We may state that the pattern size is $4M$–1 ($2M$ nodes, M vertical edges and M–1 horizontal edges) and the model size is $3N^2 - 2N - N\left\lfloor\frac{N-1}{M-1}\right\rfloor$, which comprises N^2 nodes, $N(N-1)$ vertical edges and $N\left(N-1-\left\lfloor\frac{N-1}{M-1}\right\rfloor\right)$ horizontal edges. The rule application should fail in this test set.

Model and pattern size values for the 4 parameter combinations used in our experiments can be found in Table 2.

Optimization possibilities. Since setting parameters N and M influences model and rule sizes, the test set already provides a wide range of measurement possibilities, and thus, no optimization is expected from tool providers.

6 Conclusion

In the paper, we gave an overview on typical application scenarios of graph transformation together with their characteristic features. Afterwards, a sample problem was selected for each scenario, and we described these benchmarks in details by presenting their metamodel and the graph transformation rules, which represent the behaviour of the system. Furthermore, we worked out several specifications of test sets, each consisting of an initial model and a graph rewriting sequence, in order to cover as many tool-specific optimization possibilities as possible, which allows a thorough analysis on the effects of different optimization strategies.

Our main goal was to provide precise and general benchmarks, which support the execution of repeatable measurements and which can be used on all the currently available graph transformation tools without modification.

Since these benchmarks never constitute a completed work, we plan to extend the set of descriptions in several directions in the future.

1. A model analysis example is the most principal benchmark that is missing from this paper.

2. The object relational mapping should also be extended by at least a test case, in which models have a large number of outgoing edges.

3. Finally, an interesting example would be to have a loop as a graph rewriting sequence, which consists of such rules of which only one can be fired at a time.

However, our upcoming tasks in the near future include (i) the implementation of benchmarks on several tools, (ii) the execution of runtime measurements, and (iii) a tool comparison, which is based on the results.

References

[1] Agrawal, A. and G. Karsai, *A UML-based graph transformation approach of implementing domain-specific model transformations*, International Journal on Software and System Modeling (2003), submitted.

[2] Bitton, D., D. J. DeWitt and C. Turbyfill, *Benchmarking database systems: A systematic approach*, in: M. Schkolnick and C. Thanos, editors, *Proc. of the 9th International Conference on Very Large Data Bases (VLDB)* (1983), pp. 8–19.

[3] Brant, D. A., T. Grose, B. Lofaso and D. P. Miranker, *Effects of database size on rule system performance: Five case studies*, in: *Proc. of the 17th International Conference on Very Large Data Bases (VLDB)*, 1991, pp. 287–296.

[4] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, "Handbook on Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools," World Scientific, 1999.

[5] Ermel, C., M. Rudolf and G. Taentzer, "In [4], chapter The AGG-Approach: Language and Tool Environment," World Scientific, 1999 pp. 551–603.

[6] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph rewrite language based on the Unified Modeling Language*, in: G. R. G. Engels, editor, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, LNCS **1764** (1998).

[7] Gray, J., editor, "The Benchmark Handbook for Database and Transaction Systems (2nd Edition)," Morgan Kaufmann, 1993.

[8] Heckel, R., *Compositional verification of reactive systems specified by graph transformation*, in: E. Astesiano, editor, *Fundamental Approaches to Software Engineering: First International Conference, FASE*, LNCS **1382** (1998), pp. 138–153.

[9] Object Management Group, "CWM: Common Warehouse Metamodel," `http://www.omg.org`.

[10] Rensink, A., *The GROOVE simulator: A tool for state space generation*, in: J. Pfalz, M. Nagl and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Lecture Notes in Computer Science **3062** (2004), pp. 479–485.

[11] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformation, volume 1: Foundations," World Scientific, 1997.

[12] Schürr, A., *Specification of graph translators with triple graph grammars*, in: *Proc. of the 20th Intl. Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994)*, LNCS **903** (1995), pp. 151–163.

[13] Schürr, A., A. Winter and A. Zündorf, "In [4], chapter PROGRES: Language and Environment," World Scientific, 1999 .

[14] Transaction Processing Performance Council, "TPC Benchmark C (Standard Specification, Revision 5.3)," (2004), `http://www.tpc.org/tpcc/`.

[15] U2-Partners, "UML: Infrastructure v. 2.0 (Third revised proposal)," (2003), `http://www.u2-partners.org/artifacts.htm`.

[16] Ullman, J. D., J. Widom and H. Garcia-Molina, "Database Systems: The Complete Book," Prentice Hall, 2001.

[17] Varró, D., G. Varró and A. Pataricza, *Designing the automatic transformation of visual languages*, Science of Computer Programming **44** (2002), pp. 205–227.

[18] Varró, G., K. Friedl and D. Varró, *Graph transformation in relational databases*, in: T. Mens, A. Schürr and G. Taentzer, editors, *Int. Workshop on Graph-Based Tools (GraBaTs)*, 2004.