Benchmarking for Graph Transformation*

Gergely Varró
Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Magyar tudósok körútja 2.
H-1521 Budapest, Hungary
gervarro@cs.bme.hu

Andy Schürr
Real-Time Systems Lab
Data Systems Technology Institute
Technical University of Darmstadt
Merckstraße 25
D-64283 Darmstadt, Germany
andy.schuerr@es.tu-darmstadt.de

Dániel Varró
Department of Measurement and Information Systems
Budapest University of Technology and Economics
Magyar tudósok körútja 2.
H-1521 Budapest, Hungary
varro@mit.bme.hu

Abstract

Model transformation (MT) is a key technology in the model-driven development approach of software engineering that provides automated means to capture the evolution of models and mappings between modeling languages. The pattern and rule-based paradigm of graph transformation is considered a very popular approach for specifying such model transformations. While the expressiveness *of different MT specification techniques is frequently compared on well-known transformation problems (e.g. UML-to-XMI, or UML-to-EJB mappings), no such benchmarks exist currently for comparing the* performance *of different model transformation tools. In the paper, we propose a systematic method for quantitative benchmarking in order to assess the performance of graph transformation tools. Typical features of the graph transformation paradigm and various optimization strategies exploited in different tools are identified and categorized. Moreover, the performance of several popular graph transformation tools is measured and compared on a well-known distributed mutual exclusion problem.*

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

1. Introduction

1.1. Model transformations in Model Driven Development

Model Driven Development. Model Driven Development (MDD) — also known as the Model Driven Architecture (MDA) and Model Integrated Computing (MIC) — has recently become a leading trend in software engineering. The aim of MDD is to carry out a thorough system modeling before implementation. Key ideas of MDD are (i) to create models of the software on various abstraction levels and from various viewpoints and (ii) to support automatic code generation from these models. The main advantages of the MDD concept are the reuse of high abstraction level models and an increase in productivity by high degree of automation. In fact, the general idea of MDD is not restricted to software engineering domains, but also applicable e.g. to business modeling [14] and civil engineering [6].

A key requirement of model driven development is to support the decision of domain experts by presenting them these models in an easy to understand visual way using a notation they are familiar with. Therefore the role of domain specific visual languages (DSVLs) is to assist system designers in formulating precise models on a higher level of abstraction using domain specific notations. Intensive research has been carried out in both industrial and academic fields to develop powerful methods [3, 7] and tools (DiaGen [20], Key [2], MetaEdit [19], Pounamu [31], VLCC [11]) to

Source: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing © IEEE 2005
Permission to reprint/republish this material for advertising or promotional purposes or
for creating new collective works for resale or redistribution must be obtained
from the IEEE by writing to pubs-permission@ieee.org

support DSVLs.

Model transformation. The aim of model transformations is to carry out automated translations within and between (visual) modeling languages. The MDD approach requires (i) a high-level specification language to capture such transformations, (ii) efficient algorithms and techniques to automate the execution of transformations, and (iii) extensive tool support for the industrialization of such transformations.

As pointed out in [12], model-to-model transformation tools/approaches can be divided into several categories depending on the characteristics of their rule language. They include 1) direct manipulation [5], 2) relational [23], 3) graph transformation based [28] 4) structure driven [21], 5) hybrid [4], and 6) XSLT based [22] approaches. Despite this large variety of possible rule languages, we focus only on graph transformation based solutions in this paper.

Graph transformation as a model transformation approach. Graph transformation [13, 25] provides a visual, pattern and rule based manipulation of graph models. Its history is dated back to the 1970s well before the MDD paradigm has been evolved. Since then, graph transformation has proved its maturity in the specification of visual languages and the prototyping of visual language tools. More precisely, the visual formalism of graph transformation is used to manipulate models, which are usually instances of visual modeling languages. Moreover, graph transformation is a popular technique also for capturing model transformations. Its popularity in the field is indicated by the large variety of tools (such as AGG [15], Fujaba [16], Great [1], Groove [24], PROGRES [26], Viatra [28]).

1.2. Benchmarking in model transformation

The aim of benchmarking is to systematically measure the performance of a system under different and precisely defined circumstances (i.e. by using several parameter combinations and data sets for these measurements). Such systematic measurements help system engineers in decision making i.e. when a choice has to be made between different alternatives by providing a proper assessment on the system characteristics.

While there exists a large variety of benchmarks and facilities supporting experiment design in different fields of computer engineering such as CLASP [10] for artificial intelligence, TPC Benchmark C [27] for relational databases and Manners, Waltz, ARP or Weaver [8] for rule-based expert systems, respectively, no real benchmarks exist in the field of model transformation.

Several specification examples (mappings such as UML-to-XMI in QVT [23], object-relational [23], UML-to-EJB [18], UML-to-XSD [9]) exist for model transformation approaches, but their main goal is to *demonstrate the expressiveness of the given approach*. Although these examples have arisen from important software engineering problems, they lack a systematic method for *measuring the performance of model transformation tools*. In fact, our initial research in benchmarking (reported in [29]) may also fall into this category, since the selection and arrangement of measurements were organized in a rather ad-hoc way without (i) defining and categorizing the properties of model transformation problems (from a benchmarking viewpoint) and (ii) fine-tuning our measurements to exploit the sophisticated optimization techniques of tools.

As a summary, the model transformation community still lacks *systematic benchmarks for measuring the performance of different tools*.

1.3. Objectives

Our aim in the current paper is to partially bridge this gap by proposing benchmarks for graph transformation tools. The area of graph transformation is a natural choice, since it is supported by a large variety of tools and a rich theory, which enables to determine some basic categories of benchmarks. We hope that our benchmark examples and benchmarking approach can later be adapted to the benchmarking of general model transformation tools as well.

After a brief introduction to the basic concepts of graph transformation (Sec. 2) we first (i) determine the most common features of graph transformation problems and tools (Sec. 3). Based on tool-specific properties (ii) we identify various optimization strategies that are used in several graph transformation tools. Moreover, (iii) we design benchmarks for different problems, each consisting of several test sets that fall into different categories. Due to space limitations only a single benchmark is selected for the current paper while the rest of the benchmark examples can be found in [30]. (iv) We execute measurements on this problem in a systematic way (by using carefully selected parameter settings and optimization strategy combinations). Finally, (v) we compare the results and shortly analyze the effects of optimization methods. Our conclusions are in Sec 6.

The main novelty can be characterized as the identification and categorization of benchmark properties for graph transformation. Furthermore, we emphasize that, up to our knowledge, this is the first systematic and quantitative performance comparison among graph transformation based tools.

2. Graph transformation

This section overviews the foundations of modeling language specification and simulation. In order to specify the abstract syntax of the modeling language, the concept of metamodeling is used and presented. On the other hand, for

simulating the behaviour of models, the paradigm of graph transformation (for details see [13, 25]) is applied.

In order to illustrate the basic terms and concepts, a distributed mutual exclusion algorithm (with full specification in [17]) has been selected as a running example. The same algorithm is used as a benchmark example in later sections.

In our running example, processes try to access shared resources. One requirement of the algorithm is that each resource may be accessed by at most one process at a time. This is achieved by using a token ring of processes. 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.

2.1. Metamodels and instance models

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*. *Associations* define connections between classes. Both ends of an association may have a *multiplicity* constraint attached to them, which declares the number of objects that, at run-time, may participate in an association. The most typical multiplicity constraints are i) the at most one (0..1), and (ii) the arbitrary (denoted by *).

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) 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. 2. It has four processes (p1 to p4) and four links (n1 to n4) of type next, which organize processes into a ring.

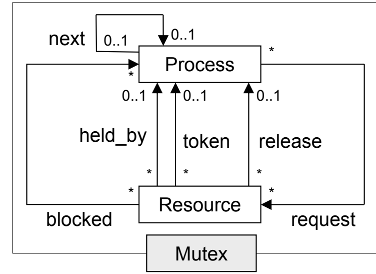


Figure 1. Metamodel for the mutual exclusion problem

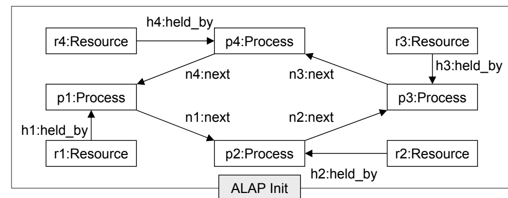


Figure 2. A sample instance model

Four resources (r1 to r4) also appear in the model. Each resource is held by a separate process, which can be expressed by the four edges of type held_by (h1 to h4) connecting the resources to the corresponding processes. Furthermore, the instance model of Fig. 2 obviously conforms to all multiplicity constraints of both metamodels.

2.2. Graph transformation rules

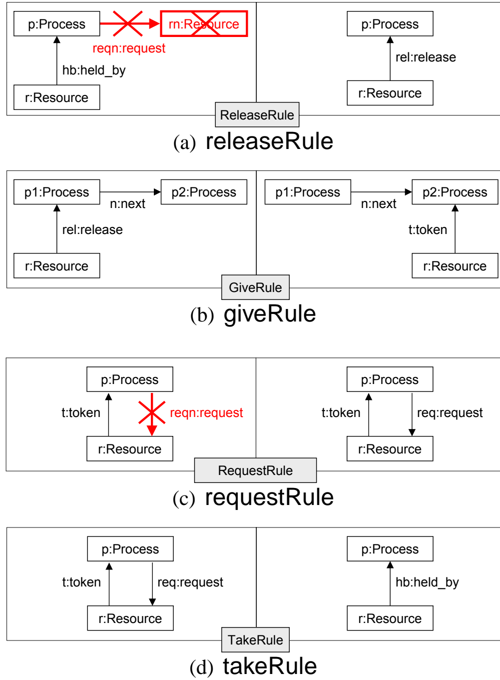
A *graph transformation rule* describes the evolution of models in a visual language in a general way (i.e. on the metalevel). Formally, a graph transformation rule $\mathsf{r} = (\mathsf{LHS}, \mathsf{RHS}, \mathsf{NAC})$ contains a left-hand side graph LHS, a right-hand side graph RHS, and negative application condition graphs NAC.

The *application* of a rule r to a *host model (instance graph) M* replaces a matching of the LHS in M by an image of the RHS. This is performed in two phases

- **Pattern matching:**
  1. find a matching of LHS in M (by graph pattern matching),
  2. check the negative application conditions NAC (which prohibit the presence of certain objects and links)
- **Updating:**

3. remove a part of the model M that can be mapped to LHS but not to RHS yielding to the context model,
4. glue 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) to obtain the *derived model* M'.

A *graph transformation* is a sequence of rule applications from an initial model M_I.



(a) releaseRule

(b) giveRule

(c) requestRule

(d) takeRule

Figure 3. A subset of rules describing the mutual exclusion algorithm

Example. The mutual exclusion algorithm [17] can be described by 13 graph transformation rules of which a subset is presented in Fig. 3. For instance, releaseRule (Fig. 3(a)) states that in its LHS a process p is held_by a resource r, while in its RHS the same nodes (p and r) are connected by an edge of type release. The releaseRule has a negative application condition which expresses that the process p cannot have any requests issued on any resources.

This rule can be applied on the model that has been presented in Fig. 2. Let us suppose that in the pattern matching phase, p, hb and r of the releaseRule are mapped to p1, h1 and r1 of the model, respectively; thus the first phase has been terminated successfully. Since the selected process p1 does not have any associated requests, the negative application condition does not prohibit the execution of the rule. In the updating phase edge h1 is removed from the model, and a new edge rl1 of type release is created. If the same rule is applied three more times (once on each remaining pair of processes and resources), then we obtain the model presented in Fig. 4(b).

3. Benchmark features

We propose the terminology and the most common features of benchmarking for graph transformation systems.

The aim of benchmarking is to systematically measure the performance of a system under different, precisely defined and deterministic (reproducable) circumstances. The criterion of determinism has a strong impact on test set definition, since theoretically, both (i) the next rule to be applied and (ii) the matching on which the rule is applied are nondeterministically selected. In order to avoid both kinds of nondeterminism, (i) we define “checkpoints”, where the instance model must be the same for all runs. Moreover, (ii) only an iterative execution of one rule is allowed between two checkpoints. Naturally, the end of the whole transformation sequence should also be a checkpoint.

3.1. Definitions of benchmarking

By a *scenario* we mean a broad application field where the paradigm of graph transformation is applicable. In [30] we mention three scenarios such as (i) model analysis, (ii) model transformation, and (iii) simulation of visual languages with dynamic operational semantics. A scenario typically has some informal characteristics (e.g. “the structure of the system does not significantly change during the transformation process”)

A *benchmark example* is a well-known problem serving as an incarnation of a scenario as it fulfills all the informal characteristics. For instance, the Mutex defined with its metamodel of Fig. 1 and graph transformation rules of Fig. 3 can be considered as a benchmark example for the simulation of visual languages as argued in Sec. 4. In technical terms, the metamodel and the graph transformation rules of the problem are fixed for a benchmark example, but instance models and concrete transformation sequences are left undefined.[1]

A benchmark example may consist of several *test sets*. A test set is a complete, deterministic, but parametric specification. In this sense, the structure of both the instance model and the transformation sequence is fixed up to numerical parameters, which characterize, for instance, the size of the model, the length of the transformation sequence, etc. Moreover, we do not decide yet which optimization strategies for different tool features (see Sec. 3.3) are turned on/off in a test set.

[1]To be precise, minor variations can be allowed in the metamodel within the same benchmark example due to practical reasons.

In a *test case*, characteristics of the model and the transformation are still parametric, but we fix which optimization strategies (for details see 3.3) to turn on.

Finally, a test case is called a *run*, when even the runtime parameters are set. Thus, a run conforms to the requirements of determinism for benchmarking, since it is completely characterized by all its parameters and it is reproducable.

3.2. Paradigm features for graph transformation

A *paradigm feature* describes a characteristics of a problem. A *feature value* is a symbolic value corresponding to a numerical interval. Thus, each test set, test case and run is defined by representative feature values assigned to paradigm features. In case of graph transformation we identified the following paradigm features and feature values:

- *Pattern size*, or in other words the number of nodes and edges in the LHS graph, is a highly critical factor in the runtime behaviour of the pattern matching phase. According to the theoretical background, the complexity of graph pattern matching algorithms is exponential in the size of the pattern graph. On the other hand, in contrast to the size of patterns, RHS graph sizes do not have strong influence on the runtime performance.

  Feature values: Since a benchmark problem may have several rules, the upper bound for the pattern sizes of all rules will be used as the value of the feature. A large (small) pattern consists of at least (at most) 15 nodes and edges. This size can be considered as a typical value for separating small and large categories in software engineering problems.

- The *maximum degree of nodes (fan-out)* in the model is the number of edges that are adjacent to a certain node. This feature has a significant impact on the complexity of a pattern matching algorithm which starts at a certain node and extends the match by examining its direct neighbourhood. To be more precise, only adjacent edges of the *same type* matter, since type checking typically precedes the enumeration of potential continuations during the pattern matching phase.

  Feature values: Values for this feature are also grouped into a small and a large category, which mean at most and at least 100 outgoing edges, respectively. This limit is typically exceeded, if containment relation appears on the modeling level.

- The third feature of a test set is the *number of matchings*. In some cases it is enough to calculate only the first matching of a rule, but in other situations all the successful matchings have to be determined. It is obvious that in the latter case, this feature directly and seriously influences the overall runtime of the pattern matching.

  Feature values: The value of the feature is again the upper bound for the number of successful matchings in the pattern matching phases of all rule applications. The term small (large) is used, if at most (at least) 10 successful matchings exist.

- The *length of the transformation sequence* also affects the overall execution time. The more rule applications are performed, the longer it will take. However, this feature does no longer influence the average time needed for a single rule execution.

  Feature values: The value of this feature is the number of atomic rule executions performed. Terms short (long) sequence are used, if the length is at most (at least) 1000.

Given a complete test set description, values for paradigm features can be determined as a function of runtime parameters. For instance, the length of a transformation sequence in the ALAP test set of the Mutex benchmark (see Sec. 4.1) is $4N$, where N is a parameter corresponding to the number of processes, thus, this paradigm feature is parameter dependent.

3.3. Tool features

Up to this point, features were completely dependent only on problem descriptions. Now we identify *tool features*, which are categories for typical optimization supported by different tools. For the moment four tool features are identified.

- In case of *parallel rule execution*, all matchings of a rule are calculated in the pattern matching phase, and then updates are performed as a transaction block on the collected matchings without re-evaluating valid matchings during the transaction. For parallel rule executions we assume that the individual matchings are independent of each other.

- *'As long as possible' (ALAP) rule application* means an iterative execution of the selected rule. A standard graph rewriting step (with a pattern matching and an updating phase) is performed in each iteration as long as a matching can be found. A possible optimization strategy is to calculate independent matchings concurrently, and then to call the same procedure recursively.

  The termination of the iteration should be guaranteed by the system designer. Thus, in order to avoid infinite loops, it must be ensured that the number of matching patterns always decreases, which is a sufficient criteria for termination.

- *Multiplicity based optimization* is used, when a tool applies a different (and usually more powerful) strategy in order to find matching model elements for an edge with 0..1 multiplicity. A typical strategy is to traverse 0..1 edges first in the pattern matching phase, since it yields a search tree that is narrower at the topmost levels.

- *Parameter passing* provided between consecutive rule applications means pattern matching in the subsequent rewriting steps is accelerated by directly reusing model elements passed as parameters without recalculating them in the later steps.

Naturally this set of tool features cannot be complete, since new heuristics can be discovered in the future, futhermore, it ignores features that are specific to a single tool.

3.4. Feature matrix

Paradigm features	Mutex		
	Short TS	Long TS	ALAP execution
LHS size (small/large)	small	small	small
fan-out (small/large)	PD	small	small
matchings (few/many)	PD	PD	PD
transformation sequence length (short/long)	PD	PD	PD

Tool features	Mutex		
	Short TS	Long TS	ALAP execution
parameter passing	***ON/OFF***	OFF	NA
0..1 multiplicities	***ON/OFF***	OFF	OFF
parallel execution	OFF	OFF	***ON/OFF***
as long as possible	NA	OFF	OFF

Table 1. Paradigm and tool features of the mutual exclusion benchmark example

A *feature matrix* (see Table 1) summarizes the features of test sets. Rows of the upper and the lower table correspond to paradigm and tool features, respectively. Columns represent test sets. Moreover, these test sets can be grouped to form a benchmark example. A field in the table contains the feature value that characterizes the given feature of a test set.

As the domain of feature values differ for paradigm and tool features, the possible values in the feature matrices are also different. Paradigm features may have values that have been defined in Sec. 3.2, or they may be parameter dependent (PD), if their category depends on the runtime parameter. Tool features can be characterized by three values. Label ON (OFF) means that the corresponding optimization strategy is applicable and it is switched on (off) in our measurements. Label NA denotes that the optimization strategy for the tool feature is not applicable.

We selected test cases according to the following principles.

1. All possible combinations of switching on and off tool optimization strategies should be avoided, since this method would be practically infeasible as it requires unacceptably high effort even for a single test set.

2. In order to obtain a feasible method, each optimization strategy is enabled only for the test set, where the effect of optimization is the most significant.

By following the above guidelines, only 7 test cases are required for our measurements instead of the original 32 test cases (that correspond to all possible combinations of ONs and OFFs). Note that measurements for the ALAP style rule execution is omitted here, since no optimization strategies are built into existing tools.

4. A benchmark example: Distributed mutual exclusion algorithm

For the current paper, we selected a benchmark example for the scenario of simulation of visual languages with dynamic operational semantics. This scenario can be characterized (i) by a nearly static graph structure, where only a small part of the model is modified, and (ii) by short rewriting sequences that are executed many times during a simulation run. Test sets are defined as rule application sequences that describe different possible runtime behaviours of the system.

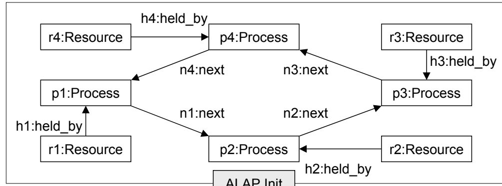
The benchmark example discussed in the paper is a distributed mutual exclusion algorithm (see the full specification in [17]). Despite the fact that this algorithm lacks some typical characteristics of model transformation scenarios such as large pattern sizes, a necessarily small number of matchings and short transformation sequences, we selected this benchmark due to its compactness: it makes possible to discuss all the tool features and to measure the effects of different optimization strategies in a single example. Moreover, this mutual exclusion algorithm is well-known and it can be easily implemented, which is also an argument in favour of this benchmark.

As shown in the feature matrix of Table 1, three test sets have been defined for this benchmark example. This means that runs on all test sets have been executed for our measurements in Sec. 5. However, due to space restrictions, only the test set of 'as long as possible' rule application is presented in details. For all the other descriptions, see [30].

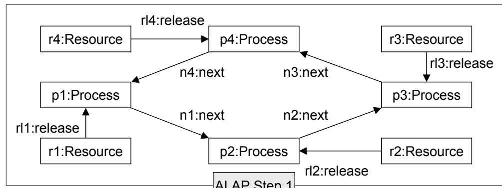
4.1. The 'as long as possible' test set

In order to illustrate a detailed test set description, we selected the 'as long as possible' (ALAP) test set, since it

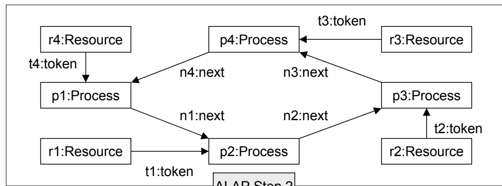
can demonstrate the usage of several tool features i.e. the as long as possible and the parallel rule execution on a single example. Despite the fact that this test set enables the evaluation of the effects of the 'as long as possible' style rule application (as it is also shown by its name), we have not examined this optimization possibility in our measurements, since no existing tools support this construct with optimized strategies.



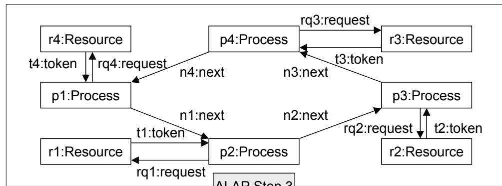
(a) Initial model with parameter $N = 4$



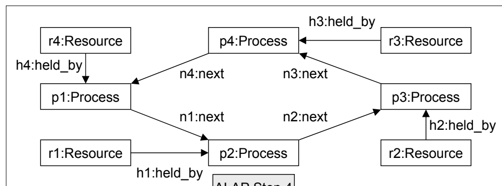
(b) Model after the 1st step



(c) Model after the 2nd step



(d) Model after the 3rd step



(e) Model after the 4th step

Figure 4. Models in different phases of the ALAP test set

The test set can be characterized by small LHS graphs, and small number of fan-outs of model nodes. Furthermore, the length of transformation sequences and the number of matchings depend on the single runtime parameter N, which denotes the number of processes in the system. More precisely, the transformation sequence length and the number of matchings can be expressed as $4N$ and N, respectively.

The initial instance model consists of $2N$ nodes (N processes and N resources) and $2N$ edges. Process nodes are arranged into a token ring along N edges of type next. Furthermore, each resource is reserved by at most one process and each process holds at most one resource at a time. In the model, this property is expressed by N edges of type held_by. A sample initial instance model is presented in Fig. 4(a) for the case $N = 4$.

4.2. Test case with sequential rule application

The transformation sequence of the test set consists of 4 macro steps. Each macro step is an iterative execution of a single rule.

1. During the first step, releaseRule is executed N times, yielding a model (see Fig. 4(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 again N times. This rule enables the next process in the ring to reserve the resource by giving the token to the process. The result model is depicted in Fig. 4(c).

3. The iterative execution of requestRule initiates a process to issue a request on the resource for which the process already has a token. As a result of an iteration of length N, we obtain the model of Fig. 4(d).

4. Finally, takeRule is executed N times.This rule assigns a process to a resource if the process has already reserved a token for the requested resource. The final instance model is isomorphic to the initial model. However, in the final model, a given resource is held by the next process in the token ring (see Fig. 4(e) vs. Fig. 4(a)).

4.3. Test case with parallel rule execution

Since the order of rule applications inside a macro step is irrelevant, the specific rule can be applied concurrently (in parallel) on all processes of the system. As a consequence, if each macro step consists of the parallel execution of the prescribed rule, then parallel and sequential transformations yield equivalent results.

5. Measurement results

We selected four graph transformation tools for our measurements. Our primary aim in selecting tools was to include those with essentially different pattern matching strategies and heterogeneous execution environments.

- AGG is an interpreted graph transformation tool written in Java, which directly follows a category theory based implementation. Its algorithm interprets pattern matching as a constraint satisfaction problem to be solved.
- PROGRES is one of the first graph transformation tools. It operates on an additional underlying graph based database (GRAS). PROGRES can run in compiled mode as it can generate C code from the specification. The strategy of PROGRES for pattern matching performs local search.
- FUJABA also uses the local search technique for pattern matching. It also belongs to compilation based tools, but in this case JAVA code is generated.
- The database (DB) approach operates on a standard relational database by issuing join based queries for pattern matching, which rank this approach among the interpreted graph transformation tools. It communicates with the database via the standard JDBC interface.

We examine one by one which tool features and optimization strategies of Sec. 3.3 are supported by these tools. (i) Parameter passing is supported by all four tools taking part in the measurements. (ii) Parallel rule application is possible in all tools except for AGG. (iii) Fujaba and PROGRES provide different methods for traversing edges with bounded multiplicity, while no such optimization strategies exist for AGG and the DB approach. Since (iv) none of these tools supports ALAP rule application with optimized procedures, investigations on measuring the effects of this tool feature are omitted from the current paper.

In order to assess the performance of graph transformation tools, tests were performed on a 1500 MHz Pentium machine with 768 MB RAM. A Linux kernel of version 2.6.7 served as the underlying operating system.

All the runs were executed without the GUI of tools, so rule applications were guided by JAVA programs (except for the measurements for PROGRES, where C programs were used). This way, we were doing programmed graph rewriting in each case for batch transformations.

Our general guideline for the comparison of tools was to use the standard services available in the default distribution, fine-tuned according to the suggestions of different tool developers. For instance, we exploited a parameter passing strategy of AGG, which is only available

	Mutex (short TS)		Proc. #	Model size #	TS length #	AGG match msec	AGG update msec	PROGRES match msec	PROGRES update msec	Fujaba match msec	Fujaba update msec	DB match msec	DB update msec
giveRule	multiplicity opt.	OFF	10	32	49	2.89	2.24	0.40	0.09	0.18	0.15	5.02	31.42
	param. passing	OFF	100	302	499	5.18	10.58	0.31	0.20	0.27	0.14	7.10	33.38
	parallel exec.	OFF	1000	3002	4999	-	-	0.63	0.26	0.35	0.03	4.26	32.13
	multiplicity opt.	ON	10	32	49			0.28	0.18	0.17	0.13		
	param. passing	OFF	100	302	499			0.14	0.09	0.19	0.15		
	parallel exec.	OFF	1000	3002	4999			0.49	0.28	0.08	0.03		
releaseRule	multiplicity opt.	OFF	10	32	49	3.16	1.54	0.26	0.06	0.19	0.22	18.99	48.38
	param. passing	OFF	100	302	499	3.12	9.11	11.71	0.18	0.70	0.17	12.87	55.86
	parallel exec.	OFF	1000	3002	4999	-	-	249.23	1.25	2.11	0.04	32.86	49.99
	multiplicity opt.	ON	10	32	49			0.20	0.16	0.19	0.22		
	param. passing	OFF	100	302	499			0.10	0.08	0.63	0.12		
	parallel exec.	OFF	1000	3002	4999			0.48	0.29	2.10	0.04		
	multiplicity opt.	OFF	10	32	49	2.65	1.87	0.16	0.08	0.14	0.23	7.32	51.48
	param. passing	ON	100	302	499	2.89	13.19	0.30	0.17	0.15	0.18	11.96	48.85
	parallel exec.	OFF	1000	3002	4999	-	-	0.49	0.40	0.03	0.03	-	-
	multiplicity opt.	ON	10	32	49			0.31	0.15	0.14	0.22		
	param. passing	ON	100	302	499			0.12	0.07	0.15	0.12		
	parallel exec.	OFF	1000	3002	4999			0.47	0.23	0.03	0.03		
	(long TS)												
release	multiplicity opt.	OFF	4	21	2500	1.86	17.55	0.62	0.15	0.15	0.09	4.15	34.01
	param. passing	OFF											
	parallel exec.	OFF	1000	5001	60001	1116.34	871.32	269.58	0.62	0.26	0.03	20.47	29.35
	(ALAP)												
releaseRule	multiplicity opt.	OFF	10	50	40	19.37	5.93	0.34	0.08	2.21	0.18	7.38	33.56
	param. passing	OFF	100	500	400	7.02	7.57	20.08	0.37	0.56	0.17	8.81	50.52
	parallel exec.	OFF	1000	5000	4000	81.34	148.91	242.95	0.85	0.60	0.10	24.12	62.06
	multiplicity opt.	OFF	10	50	40			0.10	0.19	2.17	0.19	1.23	0.78
	param. passing	OFF	100	500	400			0.16	0.08	0.19	0.17	0.54	1.65
	parallel exec.	ON	1000	5000	4000			0.38	0.06	0.09	0.11	0.81	0.90

Table 2. Experimental results

in programmed mode. In case of FUJABA, the models themselves were slightly altered to provide better performance. We used GRAS as being the default underlying graph-oriented database for the PROGRES tests, and in addition, the Prolog-style cuts in the specification to make the execution deterministic. Moreover, the standard interpreter of PROGRES was completely ignored during the measurements as we prepared the compiled version of the specification. In case of database tests, MySQL (version 4.1.7) with the default configuration was used as the underlying relational database using the built-in query optimization strategies.

Table 2 shows the execution times of three test sets (having different characteristics and optimization strategy combinations) carried out on our mutual exclusion benchmark example. The head of a row (i.e. the first two columns) shows the name of the rule and the optimization strategy configuration on which the average is calculated. (Note that a rule is executed several times in a run.) The third column (Proc) depicts the number of processes in the run, which is, in turn, the runtime parameter N for the test case. The fourth and fifth columns show the concrete values for the model size and the transformation sequence length, respectively. Values in match and update columns depict the average times needed for a single execution of a rule in the pattern matching and updating phase, respectively. Execution times were measured on a microsecond scale, but a millisecond scale is used in Table 2 for presentation purposes. Light grey areas denote the lack of support for a combination of optimization strategies by a given tool.

From the measurements of Table 2, we can make the following observations.

- Runtime performance of a single tool in the pattern matching phase may significantly differ depending on the structure of LHS. This means that further paradigm features (e.g. the number of nodes in the NAC) need to be identified in the future to refine our measurements.
- A larger model results in a larger number of update operations; therefore, the constant overhead (e.g. for compiling Java byte code) is distributed over a larger number of rule applications, which yields a decreasing series of average values for update operations as the model size increases.
- The update phase of AGG shows a significantly increasing trend as the model size increases. In fact, in case of large models, the update phase of AGG takes at least as much time as the pattern matching phase itself which is quite unexpected. The reason for that is a compilation step from graphs to categories that is carried out in each graph transformation step.
- The updating phase of the DB approach is significantly longer than in case of other tools, which is a consequence of the extra work that is performed to free the table allocated for the result set.
- The effect of multiplicity based optimization is not significant in case of Fujaba. In contrast, PROGRES may have a heavy decrease in the execution time if a different strategy is used for bounded multiplicities. However, this speed-up depends on the given rule.
- The gain from parameter passing is noticable for Fujaba and PROGRES, while AGG and DB approaches cannot benefit from this tool feature. In case of AGG, parameter passing is not officially supported, i.e. it was programmed manually for the measurements. In the DB case, optimizations for parameter passing are carried out automatically by the query optimizer of the database.
- The effect of parallel rule execution is noticable in case of all tools that support this feature, but significant speed-up is produced only by PROGRES and the DB approach.

6. Conclusion

In the paper, we proposed a benchmarking framework for assessing the performance of different graph transformation tools. For this purpose, we first identified and categorized features of graph transformation problems and tools. Based on tool-specific features we identified various optimization strategies that are present in several graph transformation tools. Moreover, we designed benchmark examples for different problems in a systematic way to enable precise and repeatable performance measurements. After selecting a benchmark example, we carried out measurements on four different graph transformation tools by using different parameter settings and optimization strategy combinations. Finally, we compared and analyzed the performance results of our measurements.

Based on our observations, we conclude that

- our initial set of paradigm and tool features was a good choice as they acknowledged our expectations that these features were significant from the viewpoint of performance measurements,
- the initial set of paradigm features is not complete as not only the pattern size, but, for instance, the pattern structure and the appearance of negative application conditions also influence the performance of graph transformation tools, and
- interesting trends could be observed on different aspects of graph transformation tools.

For developers of graph transformation tools, we recommend to focus on developing more efficient techniques for the processing of multiple matchings in situations where the straightforward parallel matching approach no longer works.

Our upcoming tasks in the future include (i) the extension of the measurements to other tools and benchmark examples to provide a wide range comparison to the community, (ii) the extension of the set of paradigm features to be able to analyze the behaviour of graph transformation tools more thoroughly, and (iii) the adaptation of our benchmark examples and benchmarking approach to support the benchmarking of general model transformation tools.

7. Acknowledgements

Authors are very grateful to Marita Breuer (PROGRES – Aachen), Gabi Taentzer (AGG – TU Berlin), Albert Zündorf, and Christian Schneider (Fujaba – Uni-Kassel) for giving valuable comments and assistance in fine-tuning different tools. Moreover, Katalin Friedl (TU Budapest) is also highly acknowledged for reading initial versions of the paper and giving valuable feedback.

References

[1] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, and A. Vizhanyo. The design of a simple language for graph transformations. *Journal in Software and System Modeling*, 2005. In review.

[2] W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY

system: Integrating object-oriented design and formal methods. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering. 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 2002, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.

[3] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Sci. Comput. Program.*, 34(1):1–54, 1999.

[4] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *Online Proc. of the OOPSLA'03 Workshop on Generative Techniques in the Context of the MDA*, 2003. `http://www.softmetaware.com/oopsla2003/bezivin.pdf`.

[5] P. Boocock. *Jamda: The Java Model Driven Architecture*, May 2003. `http://sourceforge.net/projects/jamda/`.

[6] A. Borkowski, E. Grabska, and J. Szuba. On graph-based knowledge representation in design. In A. D. Songer and J. C. Miles, editors, *Computing in Civil Engineering*, pages 1–10, 2002.

[7] P. Bottoni, S.-K. Chang, M. F. Costabile, S. Levialdi, and P. Mussio. On the specification of dynamic visual languages. In *Proc. IEEE Symposium Visual Languages'98*, pages 14–21, 1998.

[8] D. A. Brant, 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)*, pages 287–296, 1991.

[9] D. Carlson. *Modeling XML Applications with UML: Practical e-Business Applications*. Addison Wesley Professional, 2001.

[10] P. R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, Massachusetts, 1995.

[11] G. Costagliola, A. D. Lucia, S. Orefice, and G. Tortor. A parsing methodology for the implementation of visual systems. *IEEE Trans. Softw. Eng.*, 23(12):777–799, 1997.

[12] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Online Proc. of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003. `http://www.softmetaware.com/oopsla03/czarnecki.pdf`.

[13] H. Ehrig, 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.

[14] H.-E. Eriksson and M. Penker. *Business Modeling with UML: Business Patterns at Work*. John Wiley & Sons, Inc., February 2000.

[15] C. Ermel, M. Rudolf, and G. Taentzer. *In [13], chapter The AGG-Approach: Language and Tool Environment*, pages 551–603. World Scientific, 1999.

[16] T. Fischer, 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)*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, 1998.

[17] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering: First International Conference, FASE*, volume 1382 of *LNCS*, pages 138–153. Springer-Verlag, 1998.

[18] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, April 2003.

[19] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM04)*, Vancouver, British Columbia, Canada, October 2004.

[20] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.

[21] OMG Document: ad/03-08-11. *Interactive Objects and Project Technology, MOF Query/Views/Transformations Revised Submission*, August 2003.

[22] M. Peltier, J. Bézivin, and G. Guillaume. MTRANS: A general framework based on XSLT for model transformations. In *Proc. of the Workshop on Transformations in UML*, pages 93–97, Genova, Italy, April 2001.

[23] QVT Partners. *Revised submission for MOF 2.0 Query/Views/Transformations RFP*, August 2003. `http://qvtp.org/`.

[24] A. Rensink. 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)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

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

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

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

[28] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

[29] G. Varró, K. Friedl, and D. Varró. Graph transformation in relational databases. In *Int. Workshop on Graph-Based Tools (GraBaTs)*, 2004. `http://tfs.cs.tu-berlin.de/grabats/`.

[30] G. Varró, A. Schürr, and D. Varró. Benchmarking for graph transformation. Technical report, Budapest University of Technology and Economics, 2005. `http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf`.

[31] N. Zhu, J. C. Grundy, and J. G. Hosking. Pounamu: a metatool for multi-view visual language environment construction. In *Proceedings of the 2004 International Conference on Visual Languages and Human-Centric Computing*, pages 254–256, Rome, Italy, September 2004.