Graph Transformation and AI Planning

Stefan Edelkamp
Computer Science Department
University of Dortmund
stefan.edelkamp@cs.uni-dortmund.de

Arend Rensink
Computer Science Department
University of Twente
rensink@cs.utwente.nl

Abstract

This document provides insight to the similarities and differences of Graph Transformation and AI Planning, two rising research fields with different publication organs and tools.

While graph transformation systems can be used as a graphical knowledge engineering front-end for designing planning problems, AI planning technology (especially heuristic search) can accelerate the exploration process in graph transformation benchmarks.

Introduction

Graph transformation systems have shown to be suitable representations for software and hardware systems and extend traditional transition systems by relating states with graphs and transitions with partial graph morphisms. Intuitively, a (partial) graph morphism associated to a transition represents the relation between the graphs associated to the source and the target state of a transition. More specifically, such system models the merging, insertion, addition and renaming of nodes or edges.

Figure 1: Vidio game creation in *StageCast*.

Copyright © 2007, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

An example for a simple graph transformation system is *StageCast*[1] to create a visual programming environment for 2D (jump'n run) computer games based on behavioral rules associated to graphical objects, visual pattern matching, simple control structures, and intuitive rule-based behavior modeling (see Figure 1).

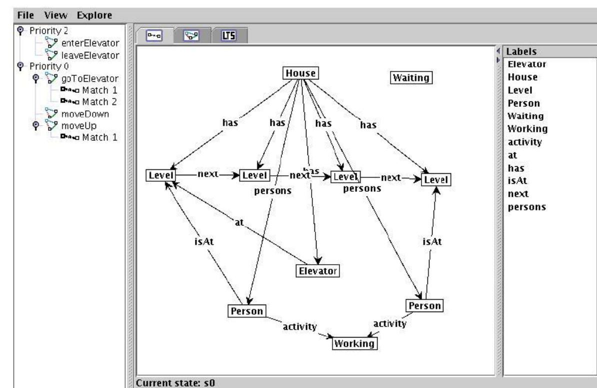

Figure 2: Simulation of graph transformation in GROOVE.

A more advanced tool for graph transformation is GROOVE (see Figure 2). It takes the point of view that graphs are a good basis for both design-time and run-time models of (software) systems. Graphs have the advantage of a visual representation (even though this tends to remain practical only for small-scale examples) and, more importantly, a rich formal foundation; in addition, they are flexible enough to deal with various kinds of models. Finally, in the long-standing theory of graph transformation we find a mathematical tool to formalize most types of transformation (Ehrig *et al.* 2006).

This paper serves three different purposes. First, it introduces graph transformation[2] as an option to model planning problems in a graphical user interface with the planning

[1] www..stagecast.com

[2] There is a notational discrepancy of *graph transformation* - for modeling and simulation - and *graph transition* for the verification of properties and systematic exploration of the system's state space. As our main concern is analysis and search, we often prefer the term *graph transition* to the term *graph transformation*.

goal as a left-hand side for an error specification in the graph transformation system.

Second, we show that for some graph transformation domains, AI action planning systems show better exploration efficiencies by using planner-specific but problem-independent state-to-goal heuristics, and discuss the advances of graph transformation of effective handling of dynamic environments with inherent symmetries.

Third, the paper provides basic information necessary to deal with the graph transformation benchmark that featured the International Competition on Knowledge Engineering for Planning & Scheduling, which included GROOVE as one simulator environment for studying different plan models.

Graph Transition Systems

In order to get a mathematical flavor, what graph transformation is about, the following set of definitions briefly introduces the theory of graph transition function for the *single-pushout* approach based on partial graph morphisms using a left and right rule application pair[3].

Definition 1 *A* graph *is a tuple* $G = \langle V_G, E_G, src_G, tgt_G \rangle$ *where* V_G *is a set of nodes,* E_G *is a set of edges,* $src_G, tgt_G : E_G \to V_G$ *are a source and target functions.*

Graphs usually have a distinguished start state which we denote with s_0^G, or just s_0 if G is clear from the context.

Definition 2 *A* path *in a graph* G *is an alternating sequence of nodes and edges represented as* $u_0 \xrightarrow{e_0} u_1 \ldots$ *such that for each* $i \geq 0$ *we have* $u_i \in V_G$, $e_i \in E_G$, $src_G(e_i) = u_i$ *and* $tgt_G(e_i) = u_{i+1}$, *or, shortly* $u_i \xrightarrow{e_i} u_{i+1}$.

An initial path is a path starting at s_0^G. Finite paths are required to end at states. The length of a finite path p is denoted by $|p|$. The concatenation of two paths p, q is denoted by pq, where we require p to be finite and end at the initial state of q.

Definition 3 *A* graph morphism $\psi : G_1 \to G_2$ *is a pair of mappings* $\psi_V : V_{G_1} \to V_{G_2}$, $\psi_E : E_{G_1} \to E_{G_2}$ *such that we have* $\psi_V \circ src_{G_1} = src_{G_2} \circ \psi_E$, $\psi_V \circ tgt_{G_1} = tgt_{G_2} \circ \psi_E$. *A graph morphism* $\psi : G_1 \to G_2$ *is called* injective *if so are* ψ_V *and* ψ_E*;* identity *if both* ψ_V *and* ψ_E *are identities, and* isomorphism *if both* ψ_E *and* ψ_V *are bijective. A graph* G' *is a* subgraph *of graph* G, *if* $V_{G'} \subseteq V_G$ *and* $E_{G'} \subseteq E_G$, *and the inclusions form a graph morphism.*

A partial graph morphism $\psi : G_1 \to G_2$ *is a pair* $\langle G'_1, \psi_m \rangle$, *where* G'_1 *is a subgraph of* G_1, *and where* $\psi_m : G'_1 \to G_2$ *is a graph morphism.*

The composition of (partial) graph morphisms results into another (partial) graph morphism. Now, we define a notion of transition system.

[3]For the *double-pushout* approach each rule consists of a triple of left-hand side, invariant and a right-hand side. A rule specifies that an occurrence of the left-hand side L in a larger graph G can be rewritten into the right hand side R preserving the interface K. For more information see (Corradini *et al.* 2006).

Definition 4 *A* transition system *is a graph* $M = \langle S_M, T_M, in_M, out_M \rangle$ *whose nodes and edges are respectively called* states *and* transitions, *with* in_M, out_M *representing the source and target of an edge respectively.*

Finally, we are ready to define graph transition systems, which are transition systems together with morphisms mapping states into graphs and transitions into partial graph morphisms.

Definition 5 *A* graph transition system *is a pair* $\langle M, g \rangle$, *where* M *is a transition system and* $g : M \to \mathcal{U}(\boldsymbol{G}_p)$ *is a graph morphism from* M *to the graph underlying* $\boldsymbol{G}_p$, *the category of graphs with partial graph morphisms. Therefore* $g = \langle g^S, g^T \rangle$, *and the component on states* g^S *maps each state* $s \in S_M$ *to a graph* $g^S(s)$, *while the component on transitions* g^T *maps each transitions* $t \in T_M$ *to a partial graph morphism* $g^T(t) : g^S(in_M(t)) \Rightarrow g^S(out_M(t))$.

The GROOVE Simulator

The GROOVE project (Kastenberg & Rensink 2006) aims at the usage of model checking techniques for verifying object-oriented systems, where the states of the system are modeled as graphs, instead of bit vectors as in most explicit state representing approaches. This approach creates new opportunities to specify and verify systems in which the states mainly depend on a set of reference values instead of values of primitive types (with a finite domain) only. Due to frequent (de)allocation of reference values, the states of such systems are highly dynamic, due to their variable size. Graphs provide a natural way of representing the states of such systems and specifying interesting properties. The tool[4] follows the single-pushout approach as introduced above.

The GROOVE simulator (Rensink 2004) does a small part of the job of a model checker: it attempts to generate the full state space of a given graph grammar. This entails recursively computing and applying all enabled graph production rules at each state. Each newly generated state is compared to all known states up to isomorphism; matching states are merged (Rensink 2003). No provisions are currently made for detecting or modeling infinite state spaces. Alternatively, one may choose to simulate productions manually. The tool is designed for extensibility. The internal and visual representation of graphs are completely separated, and interfaces abstract from graph and morphism implementations.

The GROOVE simulator is implemented in Java. The tool can handle arbitrary production rules, but can obviously only generate a finite part of the corresponding graph transition system. The simulator uses non-attributed, edge-labeled graphs without parallel edges. Node labels are actually labels of self-edges. The most performance critical parts of the simulator are: *finding rule matchings*, and *checking graph isomorphism*. The first problem is, in general, NP-complete; the second is in NP but its precise time complexity classification is unknown (Wegener 2003).

The modularity of GROOVE also extends to the serialization and storage of graphs and graph grammars. Currently the tool uses GXL (Winter, Kullbach, & Riediger

[4]`sourceforge.net/projects/groove`

2002), but in an ad hoc fashion: production rules are first encoded as graphs and then saved individually; thus, a grammar is stored as a set of files. A sample GXL specification is provided in the Appendix. Some performance figures on GROOVE were reported in (Rensink 2004).

Modeling Planning Problems in GROOVE

We have seen that graph transformation systems enable users to encode domain models including states and transition rules in form of graphs. This section gives insights and examples of applying graph transformation techniques for specifying and simulating action planning domains in a graphical interface. We further illustrate the impact of planners in solving some graph transition problems faster than with current technology. For the sake of simplicity, we restrict to propositional planning even if GROOVE has been recently extended to deal with numeric attributes.

Action planning refers to a state space description in logic. A number of atomic propositions describe what can be true or false in each state. By applying actions in a state, we arrive at another state where different atoms might be true or false. Usually, only some few atoms are affected by an action, and most of them remain the same. A concise representation the STRIPS formalism (Fikes & Nilsson 1971), an acronym for an early planning system developed at Stanford University. STRIPS planning assumes a closed world. Everything that is not stated as being true is assumed to be false. Therefore, the denotation of the a state is shorthand for the value assignment to *true* of the propositions that are included in the state, and to *false* of the ones that are not. Advances to STRIPS have let to the problem domain description language PDDL (McDermott 2000; Long & Fox 2003; Hoffmann & Edelkamp 2005).

Blocksworld

In Blocksworld a robot tries to reach a target state by actions that stack and unstack blocks, or put them on the table. In Fig. 3 we show a GROOVE model the four operations *stack*, *unstack*, *pickup*, and *putdown*. Nodes represent objects types (in this case there is only one) and edges represent predicates that combine object types. Edges attached with `new` correspond to add effects, while edges labeled by `del` abbreviate delete effects. Delete edges together with persistent edges (none for Blocksworld) form the preconditions of the rule.

Figure 4 illustrates the specification of the initial state in GROOVE. We find block B on top of block B and a singleton block A on the table. The robot arm is empty. Finally, Figure 5 depicts the (entire) graph transition system that is generated by continousely applying the 4 rules starting from the initial state, where rule application is established by matching the rule to the graph representing the current state.

Logistics

An example for a more complex STRIPS domain is Logistics. The task is to transport packages within cities using trucks, and between cities airplanes. Locations within a city are connected and trucks can move between any two such locations. In each city there are one truck and one airport.

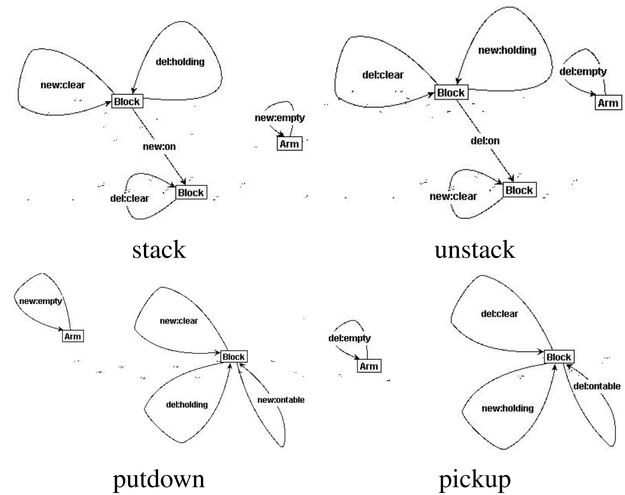

Figure 3: The graph transformation rules for Blocksworld.

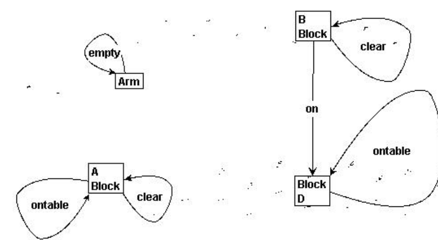

Figure 4: The initial state for Blocksworld.

Fig. 6 shows the modeling of the actions as a graph transformation rule in GROOVE. With Trucks, Airplane, Location and Packages four object types are used. Context edges like do not incur changes such as *in* for the *drive* action.

Running a Planner on GROOVE Models

In this section we address the capability of modern planning in solving one of the problems that have been denoted as a challenge to graph transformation (Rensink 2006).

The Girl's Gossip Problem

There is a number of n girls, each of which has her own secret and one call action. On a call both participating girls divulge all the secrets they know. The question is, what is the minimal number of alls after which all girls know all secrets. The know optimum is $2n - 4$ calls.

The problem has been modeled as a graph transition system. In the following, we give an intuitive PDDL description for it utilizing conditional effects and bounded quantification. For the only action specification we have

```
(:action call
 :parameters (?g1 ?g2 - girl)
 :effect (and (forall (?s - secret)
     (when (has-secret ?g1 ?s) (has-secret ?g2 ?s)))
       (forall (?s - secret)
     (when (has-secret ?g2 ?s) (has-secret ?g1 ?s)))))
```

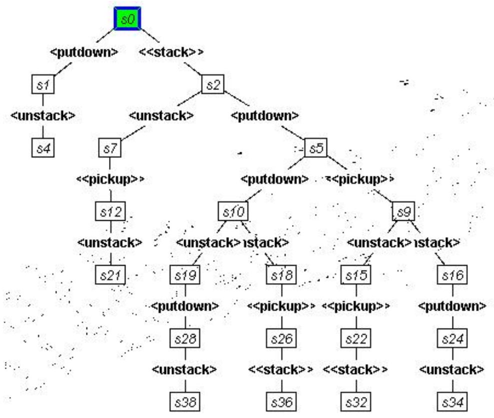

Figure 5: The state space generated from the initial state by the rule set.

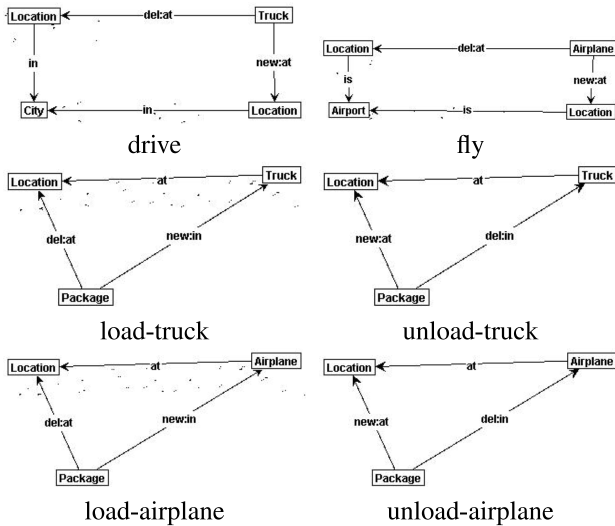

Figure 6: The graph transformation rules for Logistics.

The goal condition is that all girls know all secrets and corresponds to the requirement

```
(:goal
 (and
  (forall (?g - girl)
     (forall (?s - secret)
       (has-secret ?g ?s)))))
```

Table 1 shows the result of running a planner in comparison with two documented versions of GROOVE to illustrate the exponential gain of heuristic search state space search[5]. In the heuristic search planner FF (Hoffmann & Nebel 2001) we choose best-first instead of enforced hill-climbing, as the former delivers plans that are non-optimal. We have applied weighted A*, with an inadmissible heuristic weight factor of 2. Hence, there was no guarantee that the established

	GROOVE Plain		GROOVE Quant.		FF	
girls	states	sec	states	sec	states	sec
5	381	2	.	1	.	1
6	4,448	11	.	1	.	1
7	80,394	240	.	1	.	1
8	2,309,763	13,308	60,990	1,400	.	1
9	-	-	2,132,210	87,302	.	1
11	-	-	-	-	635	1
21	-	-	-	-	5,170	39

Table 1: Comparison of GROOVE with the AI planner FF on the girl's gossiping problem.

solution has a minimal number of steps. As we knew the minimal number of steps beforehand, we could, however, validate that the generated solutions of the planner were indeed optimal.

The Dining Philosophers

In Dijkstra's *dining philosophers problem* n philosophers sit around a table to have lunch. There are n plates, one for each philosopher, and n forks located to the left and to the right of each plate. Since two forks are required to eat the spaghetti on the plates, not all philosopher can eat at a time. Moreover, no communication except taking and releasing the forks takes place. The task is to devise a local strategy for each philosopher that lets all philosophers eventually eat. The simplest solution to access the left fork followed by the right one, has an obvious problem. If all philosopher wait for the second fork to be released there is no possible progress; a dead-end has occurred.

Studies in model checking (Edelkamp, Leue, & Lluch-Lafuente 2004) show that heuristic search for deadlocks scales well for this example. Using an automated translation from Promela inputs[6], a PDDL model for the dining philosopher problem was generated fully automatically. This benchmark has then be used in the 4th International Planning Competition (Hoffmann *et al.* 2006). Here we provide a simplified PDDL model for the dining philosophers, equivalent to the one that is provided in GROOVE.

```
(:action get-left
   :parameters (?p1 ?p2 - phil ?f - fork)
   :precondition
   (and (right ?p1 ?f) (left ?p2 ?f) (hungry ?p2)
        (not (hold ?p1 ?f)))
   :effect (and (hasLeft ?p2) (hold ?p2 ?f)
        (not (hungry ?p2))))

(:action get-right
   :parameters (?p1 ?p2 - phil ?f - fork)
   :precondition
   (and (right ?p1 ?f) (left ?p2 ?f) (hasLeft ?p1)
        (not (hold ?p2 ?f)))
   :effect (and (not (hasleft ?p1)) (hold ?p1 ?f)
        (eat ?p1)))
```

[5]The experiments were run on a Windows Laptop with 512 MB and 3 GHz Pentium 4 processor.

[6]Promela is the input language of the model checker SPIN (Holzmann 2003)

```
(:action go-hungry
   :parameters (?p - phil)
   :precondition (and (think ?p))
   :effect (and (not (think ?p)) (hungry ?p)))

(:action release-left
   :parameters (?p - phil ?f - fork)
   :precondition (and (eat ?p) (hold ?p ?f)
        (left ?p ?f))
   :effect (and (not (hold ?p ?f)) (not (eat ?p))
        (hasright ?p)))

(:action release-right
   :parameters (?p - phil ?f - fork)
   :precondition (and (hold ?p ?f) (right ?p ?f)
        (hasright ?p))
   :effect (and  (think ?p) (not (hold ?p ?f))
        (not (hasright ?p)))
```

Competition Usage

Graph transformation was one of the benchmark for the second International Knowledge Engineering Competition. Competitors and tools did not had to operate on the generic domain of graph transition systems, but to work on the individual graph transformation benchmarks instead. The declaration of the according graph transformation systems was be made available to the competitors.

This section is the manual of the graph transformation GROOVE simulator as used in the competition. As with GROOVE, its simulator wrapper is implemented in Java, while the server interface is written in c/c++. We briefly describe the installation and invocation of the GROOVE simulator as a server application.

Running the Simulator

The simulator is invoked as follows:

```
GROOVE <plan> <prefix> <directory>
```

where `<plan>` is the file containing the plan description in form of a sequence of graph transformation rules, `<prefix>` is an filename prefix for the generated output files, and `<directory>` is the directory of the input files, describing the rules and initial graph of the graph transformation problem.

The simulator then simulates the plan on the graph transformation problem and prints a report of the simulation to some files starting with prefix. The simulator executable is implemented as a wrapper in form of a `bash` script.

Most benchmarks are optimization problems minimizing the number of steps. The importance here is that there are more than one match to a rule such that a plan actually spans a tree of possible plan executions. So instead of the GROOVE simulator that interactively allows to execute a sequence of matches, actually the GROOVE transition system generator is invoked. A plan is reported to be valid if one of the execution traces ends up in a final state. Moreover, all reachable states that can be generated given the generated plan are returned to the client.

Example

For example, we take the *list append problem* (Rensink 2004) with the three rules, `next`, `append` and `return`. The example shows many features typical for the dynamics of object-oriented programs. Methods are modeled by nodes, with local variables as outgoing edges, including a this-labeled edge pointing to the object executing the method. Each method invocation results in a fresh node, with a caller edge to the invoking method. Upon return, the method node is deleted, while creating a return edge from its caller to a return value. It follows that the execution stack is represented by a chain of method nodes. List graph contain concurrently enabled method invocations; we expect the planner to tell us whether these invocations may interfere. A plan file might look as follows

```
next next next append return
```

This simple plan (ordered list of transitions) syntax is the one ingested by the simulator. As such plan has many possible outcomes all are reported. By applying the sequence in the example problem two possible states are generated and stored in the files files `<prefix>-<statenr>` according the GXL-Format (cf. Appendix).

Difficulty Level

We have selected graph transformation benchmark domains of rising difficulty.

- `level_1`: *Solitaire*, *Girls' Gossiping*
- `level_2`: *List Manipulation*, *Circular Buffers*
- `level_3`: *Balanced Search Trees*
- `level_4`: *Virtual Machine*

The description of the individual domains and the corresponding graph transformation systems were made available to the competitors. While *Girls' Gossiping*, *List Manipulation*, *Circular Buffers* existed before the competition, *Solitaire* and *Balanced Search Trees* are new. The *Virtual Machine* is the most complex graph transition model and has been described in (Kastenberg, Kleppe, & Rensink 2006).

Conclusion and Discussion

This paper introduces to the interconnection between planning and exploration in graph transformation systems. We used the GROOVE simulator as a case study, and showed that some traditional STRIPS planning benchmarks can be specified intuitively[7]. The interface for defining graph transformation system uses XML[8].

The paper also delivers first data for possible synergies opposite direction, i.e., how planner can accelerate the exploration process for graph transformation systems for a specific target graph. We have seen a PDDL description for a

[7]Besides Blocksworld and Logistics we also successfully modeled the *Grid* and *Gripper* domain in GROOVE.

[8]An overview on the status quo on the language development of GXL and GTXL can be found at `http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html`.

problem that has been identified as a challenge and shown that planners can be superior to state-of-the-art technologies.

In (Edelkamp, Jabbar, & Lluch-Lafuente 2005) the modeling of graph transition systems in PDDL and the application of heuristic search planning for their analysis are discussed for the first time applying different heuristics in a case study on the arrow distributed directory protocol (Demmer & Herlihy 1998).

Nonetheless the expressiveness of planners and model checkers based on graph transformation are different. On the one hand modern PDDL planners cover numerical attributes for optimizing objective functions, they allow to attach duration to rules for finding schedules of when to apply which rules to minimize the *makespan*. Recent additions to PDDL also cover additional soft constraints to be imposed on the set of feasible plans.

On the other hand, graph transformation systems like GROOVE feature untyped domain objects to allow the generation of a symmetry reduced state space. Two isomorphic graphs are represented only ones. Assigning names to objects, as done in planning actually destroys these symmetries. Moreover, GROOVE covers dynamic aspects such as objects and predicate creation that are not dealt with yet in the PDDL language.

We chose GROOVE as it is self-contained, platform independent, relatively stable, available to public domain and because the simulator is attached to a state space generator. Another simulator for graph transformation systems is AGG[9]. AGG features many different options such as efficient matching, consistency checking, and a critical pair analysis. An alternative for the verification of systems with graph transformation is AUGUR (König & Kozioura 2005). Recent developments in AUGUR unfold and approximate the graph transformation system in form of a Petri-Net (König & Kozioura 2006).

Recent work by (Zaks 2007) shows that finding counter-example in Petri-Net approximations within a graph transformation refinement loop is seemingly faster when exporting the domain to PDDL and use a from-shelf action planner planner. In his work, the author compares the planner MetricFF with the Petri Net reachability analyzer BWRA. The following table shows selected run-time results (in seconds).

Problem	BWRA	MetricFF	Error
red-black	947	1	yes
red-black	948	1	yes
firewall2	7	1	yes
firewall2	338	1	yes
firewall2	6	1	yes
server2	1	–	no
server2	6	–	no
server2	545	1	yes

For International Knowledge Engineering Competition we summarize the important contributions of our work, which makes graph transformation an appropriate candidate for a benchmark domain.

- Graph transformation systems provide an flexible, intuitive input specification for systems of change with a sound mathematical basis. The geometric layout of the graphs associates semantics to the syntactical structures. Vied that way, simulators like GROOVE can be seen as a knowledge engineering framework for specifying planning problems. With this respect, we establish a tight connection to the KE tool *ItSimple* (Vaquero *et al.* 2006; Vaquero, Tonidande, & Silva 2007).

  GROOVE has been recently extended to certain form of arithmetics on the attributes (Kastenberg 2005), so that more expressive numerical plan models can be simulated.
- Graph transformation system bridge the gap between specification and software verification as they allow to validate certain types of UML designs and software models. With the incorporation of heuristic/local search planners, bug finding as proposed in the *directed model checking paradigm* can be accelerated. A planner input compilation includes tool-inherent guidance to the search for free, while heuristics for graph transformation systems have not yet been implemented (Edelkamp, Jabbar, & Lluch-Lafuente 2006)
- The main objective of this article is to provide a challenging domain for the design of plan models. The number of graph transformation benchmark problems is continuously rising[10] even though no forum yet exists[11]

To steer GROOVE remotely, a file or protocol-base simulation has been provided. This has imposed only a minor extension to the public release of GROOVE.

Acknowledgments

The first author thanks DFG for support in the projects *Directed Model Checking* and *Heuristic Search*, Reiko Heckel for the link to *StageCast*, and Barbara König for joint supervision of the master thesis of Maxim Zaks.

References

Corradini, A.; Ehrig, H.; Montanari, U.; Ribeiro, L.; and Rozenberg, G., eds. 2006. *Graph Transformation. Proceedings of the 3rd International Conference*. Springer, Lecture Notes in Computer Science, Volume 4178.

Demmer, M. J., and Herlihy, M. 1998. The arrow distributed directory protocol. In *In International Symposium on Distributed Computing (DISC)*, 119–133.

Edelkamp, S.; Jabbar, S.; and Lluch-Lafuente, A. 2005. Action planning for graph transition systems. In *ICAPS-Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems (VVPS)*, 58–66.

[9] See http://tfs.cs.tu-berlin.de/agg

[10] In the first graph transformation contest on AGTIVE-07 *Ludo*, *UML-to-CSP*, *Sierpinsky triangles* were posed.

[11] Examples in GXL language (single push-out) like *list manipulation* and *buffer manipulation* come with the GROOVE distribution, while (double pushout) examples like *mutual exclusion*, *dining philosophers* and *red-black trees*, defined in GTXL can be obtained on http://www.ti.inf.uni-due.de/research/augur_1/examples/index.shtml

Edelkamp, S.; Jabbar, S.; and Lluch-Lafuente, A. 2006. Heuristic search for the analysis of graph transition systems. In *International Conference on Graph Transformation (ICGT)*, 414–429.

Edelkamp, S.; Leue, S.; and Lluch-Lafuente, A. 2004. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology* 5(2-3):247–267.

Ehrig, H.; Ehrig, K.; Prange, U.; and Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Hoffmann, J., and Edelkamp, S. 2005. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research* 24:519–579.

Hoffmann, J., and Nebel, B. 2001. Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J.; Edelkamp, S.; Thiebaux, S.; Englert, R.; Liporace, F.; and Trüg, S. 2006. Engineering realistic benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research* 453–542.

Holzmann, G. J. 2003. *The SPIN Model Checker*. Addison-Wesley.

Kastenberg, H., and Rensink, A. 2006. Model checking dynamic states in GROOVE. In *Model Checking Software (SPIN)*, 299–305.

Kastenberg, H.; Kleppe, A.; and Rensink, A. 2006. Defining object-oriented execution semantics using graph transformations. In *Formal Methods for Open Object-Based Distributed System (FMOODS)*, 186–201.

Kastenberg, H. 2005. Towards attributed graphs in Groove. 154(2):47–54.

König, B., and Kozioura, V. 2005. Augur—a tool for the analysis of graph transformation systems. *EATCS Bulletin* 87:125–137. Appeared in The Formal Specification Column.

König, B., and Kozioura, V. 2006. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *TACAS*, 197–211.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Overview and results. *JAIR* 20. Special issue on the 3rd International Planning Competition.

McDermott, D. 2000. The 1998 AI Planning Competition. *AI Magazine* 21(2).

Rensink, A. 2003. Towards model checking graph grammars. Technical Report DSSETR20032, Proceedings of the 3rd Workshop on Automated Verification of Critical Systems. University of Southampton.

Rensink, A. 2004. The Groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, 479–485.

Rensink, A. 2006. Nested quantification in graph transformation rules. In *International Conference on Graph Transformation (ICGT)*, 1–13.

Vaquero, T. S.; Tonidande, F.; de Barrow, L. N.; and Silva, J. R. 2006. On the use of UML.P for modeling a real application as a planning problem. In *International Conference on Automated planning and Scheduling (ICAPS)*, 434–437.

Vaquero, T. S.; Tonidande, F.; and Silva, J. R. 2007. itsimple 2.0: An integrated tool for designing planning domains. In *International Conference on Automated planning and Scheduling (ICAPS)*. To appear.

Wegener, I. 2003. *Komplexitätstheorie*. Springer.

Winter, A.; Kullbach, B.; and Riediger, V. 2002. An overview of the GXL graph exchange language. In *Software Visualization*, 324–336. Springer, LNCS.

Zaks, M. 2007. Efficient algorithms for the analysis of graph transformation systems. Master's thesis.

Appendix

GXL of the stack rule of the Blocksworld domain.

```
<?xml version="1.0" encoding="UTF-8"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd">
    <graph id="graph" role="graph" edgeids="false"
     edgemode="directed">
        <node id="n49"/> <node id="n50"/>
        <node id="n51"/>
        <edge from="n49" to="n49">
            <attr name="label">
            <string>del:clear</string> </attr>
        </edge>
        <edge from="n49" to="n49">
            <attr name="label">
            <string>Block</string> </attr>
        </edge>
        <edge from="n50" to="n50">
            <attr name="label">
            <string>Block</string> </attr>
        </edge>
        <edge from="n50" to="n49">
            <attr name="label">
            <string>new:on</string> </attr>
        </edge>
        <edge from="n50" to="n50">
            <attr name="label">
            <string>new:clear</string> </attr>
        </edge>
        <edge from="n50" to="n50">
            <attr name="label">
            <string>del:holding</string> </attr>
        </edge>
        <edge from="n51" to="n51">
            <attr name="label">
            <string>new:empty</string> </attr>
        </edge>
        <edge from="n51" to="n51">
            <attr name="label">
            <string>Arm</string> </attr>
        </edge>
    </graph>
</gxl>
```