Model Checking Graph Transformations: A Comparison of Two Approaches

Arend Rensink[1], Ákos Schmidt[2], and Dániel Varró[2]

[1] University of Twente
P.O. Box 217, Enschede, 7500 AE, The Netherlands
rensink@cs.utwente.nl
[2] Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok körútja 2, Budapest, Hungary
varro@mit.bme.hu

Abstract. Model checking is increasingly popular for hardware and, more recently, software verification. In this paper we describe two different approaches to extend the benefits of model checking to systems whose behavior is specified by graph transformation systems. One approach is to encode the graphs into the fixed state vectors and the transformation rules into guarded commands that modify these state vectors appropriately to enjoy all the benefits of the years of experience incorporated in existing model checking tools. The other approach is to simulate the graph production rules directly and build the state space directly from the resultant graphs and derivations. This avoids the preprocessing phase, and makes additional abstraction techniques available to handle symmetries and dynamic allocation.
In this paper we compare these approaches on the basis of three case studies elaborated in both of them, and we evaluate the results. Our conclusion is that the first approach outperforms the second if the dynamic and/or symmetric nature of the problem under analysis is limited, while the second shows its superiority for inherently dynamic and symmetric problems.

Keywords: logic properties of graphs and transformations, analysis of transformation systems, semantics of visual techniques, model checking

1 Introduction

Graph transformation [6, 18] represents a rich line of research in computer science. Recently, a wide range of applications have been found especially in the theoretical foundations of diagrammatic specification formalisms such as UML. The main advantage of using graph transformation lies in the fact that not only the (static) program state of these UML-related models can be stored as graphs, but it is quite obvious and natural to define the evolution of these models by transformations on those graphs.

However, software engineers may implant bugs into the system under design even if they use such a high-level and executable specification methodology as graph transformation. In this respect, one has to verify automatically and with mathematical preciseness that the system model fulfills all its requirements.

Model checking is one of the few verification techniques that, in some areas of computer science, have shown their benefits in practice and have been adopted by industry. However, the successes are mainly limited to hardware verification. It has been long recognized that software has features that make the problem inherently harder. Primary among those features is the *dynamic nature* of software, which typically relies heavily upon the dynamic allocation and deallocation of portions of memory to data structures (the heap) and control flow (the stack).

We argue in the paper that the strengths of graph transformation are precisely there where the weaknesses of current model checking approaches lie: namely, in the description of the dynamic nature of software. We have therefore sought to combine the two, by using graph transformations for the specification, and model checking for the verification of systems. This paper describes and compares two, quite different approaches towards this goal, namely, CheckVML [20, 24] and GROOVE [13, 16].

The reason we have chosen these approaches that tackle the model checking problem for graph transformations for a comparison is twofold: a) they represent the two obvious main roads (i.e. to compile graphs into an off-the-shelf tool or to write a state space generator for graphs) b) currently, they have the most extensive tool support.

Related work on model checking graph transformations. The theoretical basics of verifying graph transformation systems by model checking have been studied thoroughly by Heckel et al. in [9] (and subsequent papers). The authors propose that graphs can be interpreted as states and rule applications as transitions in a transition system, which idea is used in both approaches in the paper.

A theoretical framework by Baldan et al. [2] aims at analyzing a special class of hypergraph rewriting systems by a static analysis technique based on approximative foldings and unfoldings of a special class of Petri nets. Recently, this work has been extended in [1] to provide a precise (McMillan-style) unfolding strategy. This is essentially different from both approaches discussed in the current paper in that symmetric situations are only identified on a single path (thus they are might be investigated several times on different paths). But detecting that a certain situation has already been examined on a single path can be much cheaper in general compared to total isomorphism checks (as done in GROOVE).

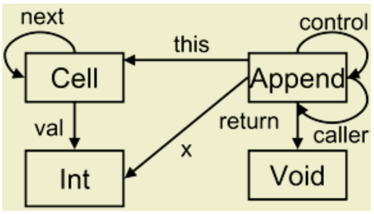
Dotti et al. [5] use object-based graph grammars for modeling object-oriented systems and define a translation into SPIN to carry out model checking. The main difference (in contrast to CheckVML) is that the authors allow a restricted structure for graph transformation rules that is tailored to model message calls in object-oriented systems. Therefore, CheckVML is more general from a pure graph transformation perspective (i.e. any kind of rules are allowed) However, the framework of [5] relies on higher-level SPIN/Promela constructs (processes and channels), which might result better run-time performance.

Structure of the paper. The rest of the paper is structured as follows. Section 2 introduces the basic concepts of graph transformation systems and model checking on a motivating example. Section 3 and 4 provides an overview of the CheckVML and the GROOVE approach, respectively. We present the results of three case studies in Sec. 5. Finally, Section 6 concludes the paper.

2 Model Checking Graph Transformation Systems

2.1 A Motivating Example: The Concurrent Append Problem

As a motivating running example for the paper, we consider the "Concurrent Append" problem for the Java program listed in Fig. 1, which implements an append method on a list of cells. Given an integer value x as parameter, the program appends a new tail cell to the list if x is not contained in any of the existing cells. An example correctness criterion is that the list of cells must not contain the same value more than once. However, we allow that different threads may access the list concurrently by calling the append method, which might result in undesired race conditions without certain assumptions on atomicity in case of the Java program below.



```
class Cell {
    Cell next;
    int val;
    void append(int x) {
        if (x == this.val)
            return;
        else if (this.next == null) {
            this.next = new Cell();
            this.next.val = x;
        } else
            this.next.append(x);
    }
}
```

Fig. 1. A Java program and its metamodel/type graph.

In the paper, we model this problem by using typed graphs [3] (or metamodels in UML terms) for describing the static structure. For instance, the metamodel in Fig. 1 expresses that a node of type Int may be connected to a node of type Cell via an edge of type val (that straightforwardly correspond to the Java attribute val). Furthermore, a next edge is leading from a cell point to the next cell in the list (if there is any). Each invocation of the append method is denoted by an Append node where we register the this pointer (which points to a cell), the caller invocation (which is another Append node), and the return value (of type Void) by edges of corresponding types. Finally, the program counter in each invocation of the append method is denoted by a control loop (self-edge).

All valid instance graphs (or models in UML terms) that represent specific invocations of the append method should comply to this metamodel in a type conforming way for both nodes and edges.

2.2 An Informal Introduction to Graph Transformation

The dynamic behavior of the recursive append method is captured by graph transformation rules. Graph transformation provides a visual, rule and pattern-based manipulation of graph models with solid mathematical foundations [6,18].

A graph transformation rule r consists of a left-hand side (LHS) and a right-hand side (RHS) graph and, potentially, some negative application conditions (NAC) which are traditionally denoted by (red) crosses. Informally, the execution of a rule on a given host graph G (i) finds a matching of the LHS in G, (ii) checks whether the matching can be extended to the matching of NAC (in which case the original matching of the LHS is invalid), (iii) removes all the graph elements from G which has an image in the LHS but not in the RHS, and (iv) creates new graph elements and embeds them into G to provide an image for rule elements that appear only in the RHS but not in the LHS. In other terms, the LHS and NAC graphs denote the precondition while the RHS denotes the postcondition for rule application.

In the paper, we use the rule notation of GROOVE (that is very similar to the notation used in the Fujaba [12]), which abbreviates the different LHS, RHS and NAC rule graphs into a single graph with the following conventions:

– *Reader* nodes and edges (i.e. elements that are part of LHS and RHS) are shown in solid thin (black) lines
– *Eraser* elements (that are part of the LHS but not the RHS) are depicted in dashed (blue) lines.
– *Creator* elements (that are part of the RHS but not the LHS) are depicted in solid thick (green) lines.
– *Embargo* elements (from the NAC) are shown in dotted (red) lines.

A sample graph transformation rule stating how to append a new element to the end of the cell list is depicted in Fig. 2 in both the traditional and the GROOVE notation[1]. The dynamic behavior of this highly recursive append problem is defined by four graph transformation rules (see Figs. 2 and 3).

Append a New Cell. Rule *Append* is responsible for appending a new cell to the list if the control reaches the last cell (see the negative condition inhibiting the existence of a next edge pointing to a Cell) and the value stored at this last cell is not equal to the method parameter. Furthermore, the append method returns one level up in the recursive call hierarchy as simulated by removing the bottom-most Append node and adding a return edge.

[1] Note that node identities are not allowed in the GROOVE tool, we only use them for presentation reasons.

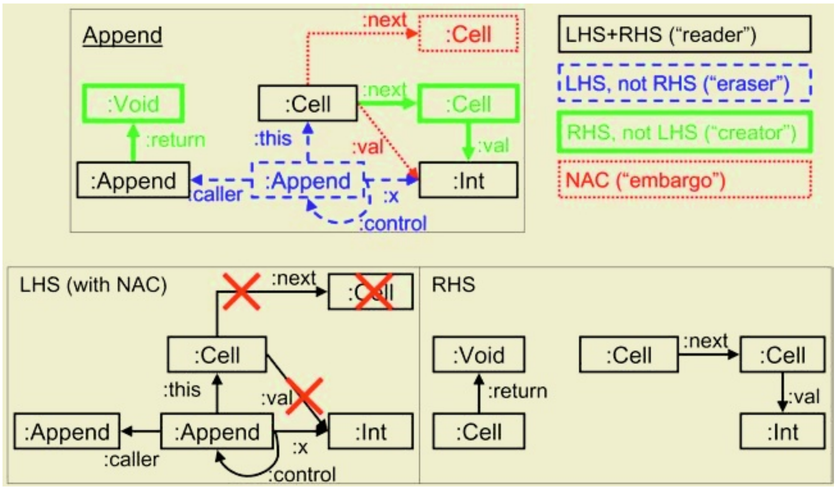


Fig. 2. The *Append* graph transformation rule in different notations.

Go to Next Cell. Rule *Next* checks whether the method parameter is *not* equal to the value stored at the current cell and makes a recursive call then for checking the next cell by generating a new Append node and passing the control to it.

Value Found in List. Rule *Found* checks if the method parameter is equal to the value stored at the current cell and, if so, returns the control to its caller append invocation node p in such a case.

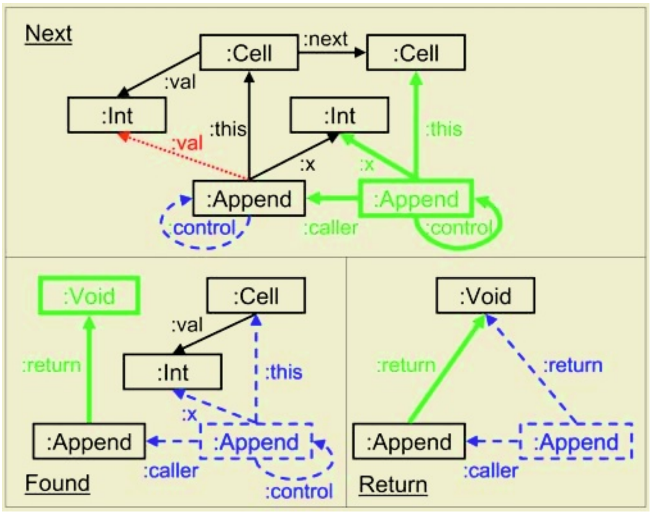


Fig. 3. Additional graph transformation rules for the concurrent append problem.

Return Result. Finally, rule *Return* simply removes an append invocation node (from the stack of recursive calls) if it has already calculated the result.

2.3 The Model Checking Problem of Graph Transformation Systems

The *model checking problem* is to automatically decide whether a certain correctness property holds in a given system by systematically traversing all enabled transitions in all states (thus all possible execution paths) of the system. The correctness properties are frequently formalized as LTL formulae.

In graph transformation systems, a state is a graph, while a transition corresponds to the application of a rule for a certain matching of the left hand side in such a graph. Traversing all enabled transitions then means applying all rules on all possible matchings. During this process, it is important to realize whether a certain state has been investigated before; therefore the model checker has to store all the graphs that it has encountered. Furthermore, ideally a model checker should exploit the symmetric nature of a problem by investigating isomorphic situations only once. The two approaches compared in the paper introduce very different techniques to tackle these problems.

For the current paper, we restrict our investigation to the verification of *safety* and *reachability* properties. A safety property defines a desired property that should always hold on every execution path or (equivalently) an undesired situation which should never hold on any execution paths (which we will call a *danger* property below). A reachability property describes, on the contrary, a desired situation which should be reached along at least one execution path. From a verification point of view, safety and reachability properties are dual: the refutation of a safety property is a counter-example which satisfies the reachability property obtained as the negation of the safety property. On the other hand, if a safety property holds (or a reachability property is refuted) the model checker has to traverse the entire state space.

A safety or reachability property can be interpreted as a special graph pattern (called *property graph* in the sequel) which immediately terminates the verification process if it is matched successfully. We have shown in [14] that the properties expressible in this way are equivalent to the $\exists\neg\exists$ fragment of ($\forall$-free) first order logic with binary predicates. For instance, the property that there exists an element that is shared among two list cells, expressed by the first-order logic property $\exists\ v\colon \mathsf{Int}, c_1, c_2\colon \mathsf{Cell}\ .\ \mathsf{val}(c_1, v) \wedge \mathsf{val}(c_2, v) \wedge c_1 \neq c_2$ is alternatively encoded in the left graph of Figure 4.

The other property graphs in Fig. 4 are *Isolated* stating that every Int-object is either a method parameter or contained in the list, and *Terminated* expressing that there are no Append-methods left. Since different interleavings of append method calls access the list concurrently, we need model checking to ensure that these properties hold.

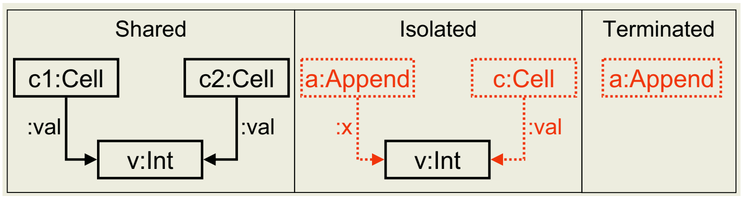


Fig. 4. Danger and reachability property graphs.

3 The CheckVML Approach

Main concepts. The main idea of the CheckVML approach [20, 23, 24] is to exploit off-the-shelf model checker tools like SPIN [11] for the verification of graph transformation systems. More specifically, it translates a graph transformation system parameterized with a type graph and an initial graph (via an abstract transition system representation) into its Promela equivalent to carry out the formal analysis in SPIN. Furthermore, property graphs are also translated into their temporal logic equivalents.

Traditional model checkers are based on so-called *Kripke structures*, which are state-transition models where the structure of a state consists of a subset of a finite universe of propositions. This determines the storage structures used (usually Binary Decision Diagrams or a variant thereof), the logic used to express properties (propositional logic extended with temporal operators, usually LTL or CTL) and the model checking algorithms (automata-based or tableau-based).

Since graph transformation is a meta-level specification paradigm (i.e. it defines how each instance of a type graph should behave) while the Kripke structure (transition system) formalism of Promela is a model-level specification language (i.e. a Promela model describes how a specific model should behave), the main challenge in this approach is *rule instantiation*, i.e. to generate one Promela transition for all the potential application of a graph transformation rule *in a preprocessing phase* at compile time.

The potential benefits of the CheckVML approach are the following:

1. It considers typed and attributed graphs which fits well to the metamodeling philosophy of UML and other modeling languages.
2. The size of the state vector depends only on the dynamic model elements (i.e., elements that can be altered by at least one graph transformation rule) while immutable static parts of a model are not stored in the state vector. This is a typical case for data-flow like systems (dataflow networks, Petri nets, etc).
3. It can be easily adapted to various back-end model checker tools.

The essential disadvantage of the approach is that dynamic model elements (that are not restricted by static constraints) easily blow up both the verification model and state space; moreover, symmetries in graphs can be handled for only very limited cases. Further research is necessitated in these directions.

Graphs and transformation rules. CheckVML uses directed, typed and attributed graphs (or MOF metamodels and models) as model representation (see the example presented in Sec. 2.1-2.2). Inheritance between node types is also supported.

Concerning the rule application strategy, CheckVML prescribes that a matching in the host graph should be an injective occurrence of the LHS (and NAC) graphs. Arbitrary creation and deletion of edges are allowed while there is an *a priori* upper bound for the number of nodes (of a certain type) potentially created during a verification run, which is passed as a parameter to the translator. Moreover, all dangling edges are implicitly removed when deleting a node.

New (unpublished) features. Several new features of CheckVML have been added as an incremental improvement since the previous papers [20, 24]. In order to improve performance, the entire tool has been rewritten, and the translator now uses relational database technology for generating all potential matches.

The main novelty is the automated translation of property graphs into LTL formulae; thus the users do not need SPIN-specific knowledge for stating properties. Since property graphs denote safety or reachability properties, thus this translation should find all potential matchings of this pattern in a similar way as done for instantiating rules.

Furthermore, in order to handle certain isomorphic situations, node identifiers have been ordered and made reusable. When a new node is created, the smallest available identifier is assigned to it, therefore, the same node can be reassigned several times. As a result, certain (but not all) isomorphic host graphs are handled only once.

Input / Output formats. CheckVML uses the GXL format [21] to store all host graphs, rule graphs (LHS, RHS, NAC) and property graphs. An XML configuration file is responsible for declaring the role of a certain graph (rule, host or property), and the user can set several translation parameters as well (e.g. upper bound for nodes of a certain type). In the near future, we plan to port CheckVML to a graph transformation tool with visual graph and rule editing facilities. The AGG tool [8] is a primary candidate due to the similarities between both the graph models and XML formats.

CheckVML generates a Promela model by instantiating rules on the host graph, and the SPIN representation of LTL formulae (which can be copy-pasted into the XSPIN framework). As a result, the users can work with high-level graph models and no (significant) SPIN-specific knowledge is required for modeling. However, counter-examples obtained as results of a verification run are currently available only in SPIN (for instance, in the form of scenarios/sequence diagrams), therefore, SPIN specific knowledge is required for the interpretation of analysis results. In the future, we also plan to investigate the possibilities of back-annotating analysis results so that they could be simulated (played back) in a graph transformation tool. Unfortunately, existing graph transformation tools provide very little support for importing entire execution traces.

Note that the overall ideas behind the CheckVML approach are not restricted to SPIN. In fact, thanks to a recent extension, CheckVML also yields an XML format for the generated transition system. Since the majority of model checker tools use transition systems as the underlying mathematical model (naturally, in their own dialect), this XML output can easily be adapted to various back-end model checkers, e.g. by XSLT scripts.

4 The GROOVE Approach

Main concepts. The idea behind the GROOVE approach (see [15] for further details on the project and downloads) is to use the core concepts of graphs and graph transformations all the way through during model checking. This means that states are explicitly represented and stored as graphs, and transitions as applications of graph transformation rules; moreover, properties to be checked should be specified in a graph-based logic, and graph-specific model checking algorithms should be applied.

This approach implies that very little of the theory and tool development for traditional model checkers can be applied immediately, since the most basic concept, namely the underlying model, has been extended drastically.

Currently only the state space generation part of GROOVE has been fully implemented. However, by the nature of graph transformation, this already implies the ability to express and check safety and reachability of graph properties, since they can be be formulated as rules with an identity morphism. Such a rule is applicable (idempotently) at precisely those states where the property holds. It is then straightforward to use such properties in controlling the state space generation process.

In particular, when treating a safety/danger property as an invariant, the state space generation halts with unexplored states exactly if the property is violated; when treating the inverse of a reachability property as an invariant, it halts precisely if the property is satisfied.

The GROOVE state space generator implements the process described in Sec. 2 to match each newly generated state against existing states up to isomorphism. While an isomorphism check is in principle quite expensive, for the examples we have worked out it stays within practical bounds.

The potential benefits of the GROOVE approach are the following:

1. There is no *a priori* upper bound to the size of the graphs;
2. There is an implicit symmetry check through the identification of isomorphic graphs;
3. No pre- or post-processing is necessary to apply the GROOVE tool to a given graph transformation system, or to translate the results of the model checking back into graphs;
4. Existing graph transformation theory can be directly brought to bear upon the tool, for instance, to discover rule independence or local confluence.

The essential disadvantage of this approach is that the huge body of existing research in traditional model checking is only indirectly applicable. In each of the areas where this applies, we aim to develop alternative techniques that are based directly on graphs.

1. *Storage techniques* (e.g., Binary Decision Diagrams). Rather than storing each graph anew, we store only the differences with the graph that it was derived from, in terms of the nodes and edges added and removed. This does mean that the actual graph has to be reconstructed when it is needed, e.g., for checking isomorphism; to alleviate the resulting time penalty this minimal representation is combined with caching.
2. *State space reduction techniques*, such as partial order reduction and abstraction. For state space reduction, we intend to use confluence properties of graph transformation rules (see advantage 3 above), or graph abstraction in the sense of *shape graphs* (see [19]). A first step towards the latter was reported in [17].
3. *Logics and model checking algorithms.* To replace the propositional logic used in traditional model checking, we have proposed a predicate graph logic in [13] for the purpose of formulating the properties to be checked. Some preliminary ideas on model checking such properties can be found in [4].

Graphs and transformation rules. GROOVE uses untyped, non-attributed, edge-labeled graphs without parallel edges. Node labels are not supported; however, we simulate them using self-edges (which indeed are also depicted by writing the labels inside the nodes). Furthermore, GROOVE implements the single pushout rewrite approach [7] (which means that dangling edges are removed while non-injective matching of the LHSs is allowed). It supports the use of negative application conditions. These can be used to specify, among other things, injectivity constraints; thus we can also simulate transformation systems in which the matchings are intended to be injective.

For the purpose of graph transformation, the lack of typing in GROOVE is not a serious drawback, since type information is not used to control the transformation process (although it may be used to optimize it). The absence of attributes is a potentially greater drawback. The examples presented here have been chosen such that attributes to not play a significant role, and so they can be simulated using ordinary edges. In fact, an extension to "true" attributes is not planned; rather, we plan to interpret data values as a special class of nodes, with ordinary edges pointing to them, as in [10].

Input/output formats GROOVE uses the GXL format [21] to store host graphs and rules. Each rule is saved as a single graph, combining the information in LHS, RHS and NACs by adding structure on the edges (in the form of a prefix) that indicates their role – or, in the case of nodes, by adding special edges for this purpose. A graph transformation system consists of all the rules in a single directory as well as its subdirectories (which are treated as separate namespaces, thus giving rise to a simple hierarchy of rules). In the future we plan to support the special-purpose format GTXL (see [22]).

Table 1. Feature comparison for CheckVML and GROOVE.

	Aspects of comparison	GROOVE	CheckVML
Graph model	Directed graphs	+	+
	Labeled graphs	+	
	Typed and attributed graphs		+
GT rules	NAC	+	+
	Node creation	arbitrary number	a priori upper bound
	Edge creation/removal	+	+
	Dangling edges	removed	removed
	Pattern matching	non-injective	injective
Input / Output	Graphical input (editor)	+	
	XML input	+	+
	Graphical output (trace)	built-in	MSCs in XSPIN
	XML output	+	+
	Property to be proved	graph constraint	graph constraint or LTL in SPIN
		safety / reachability	safety / reachability
Verification	Exploration strategies	extensible library	SPIN
	Symmetry recognition	graph isomorphism	reusable object ids
	Preprocessing	none	translation to SPIN

Alternatively, the GROOVE tool packages a stand-alone graph editor that can be used to construct graphs and rules and save them in the required format, or to read and edit graphs obtained from elsewhere.

State transition systems generated as a result of state space generation are also saved as GXL graphs, in which the nodes correspond to states (hence, graphs) and the edges to rule applications, labeled by the rule names.

State spaces can be generated either using a graphical simulator or using a command-line tool.

– The simulator, described before in [16], supports state space traversal by allowing the user to select and apply rules and matchings, all the while building up the transition system. Alternatively, the user can apply one of the available automatic state space exploration strategies (branching, linear, bounded, invariant). Graphs and transition system can be inspected by showing and hiding edges based on regular expressions over their labels.
– The command-line tool applies a pre-chosen strategy and generates and saves the resulting transition system.

Finally, Table 1 provides a brief summarizing comparison of the two tools.

5 Experimental Comparison

We have carried out three different case studies to compare the two model checking approaches, namely, (1) the Concurrent Append example of the current paper, (2) the dining philosophers problem as discussed in [24], and (3) a mutual exclusive resource allocation example taken from [9]. In the following we briefly describe the salient features of these cases.

Dining philosophers = Symmetries + No dynamic allocation. We have chosen this example because it is a traditional one, which has already been subject of a study for the CheckVML approach. For the purpose of GROOVE, this is an interesting case because with n philosophers, the example obviously has symmetry degree n, and this should then also be the reduction factor in number of states and transitions. On the other hand, the example has no dynamic allocation, and in this sense is not typical of the sort of problem for which we expect a graph transformation-based approach to be superior to traditional model checkers. We checked a safety property stating that no forks are ever held by more than one philosophers.

Concurrent append = Dynamic allocation + No symmetries. This is the running example of the paper. We have chosen it because it combines features that we believe to be typical of the "hard" problems in software verification. On the one hand, it contains dynamic allocation (list cells are created and append method frames are created and deleted), and on the other hand, it specifies concurrent behavior (several append methods are running in parallel). Note that, in the representation chosen here, the example has few non-trivial symmetries. In particular, all Int-objects in the list are distinguished by their value. We checked the property expressing that the list of cells is not allowed to contain the same value more than once.

Mutual exclusion = Dynamic allocation + Symmetries. In this example, processes try to access shared resources by using a token ring. We have chosen this example because it combines dynamic allocation (processes and resources can be created and deleted arbitrarily) and symmetry (processes and resources cannot be distinguished from one another). Moreover, a graph-based description of the protocol is very natural: an argument can be made that the specification of this protocol using graph transformation rules is superior to any other. The verified requirement was that at most one process may be allowed to access each resource at a time.

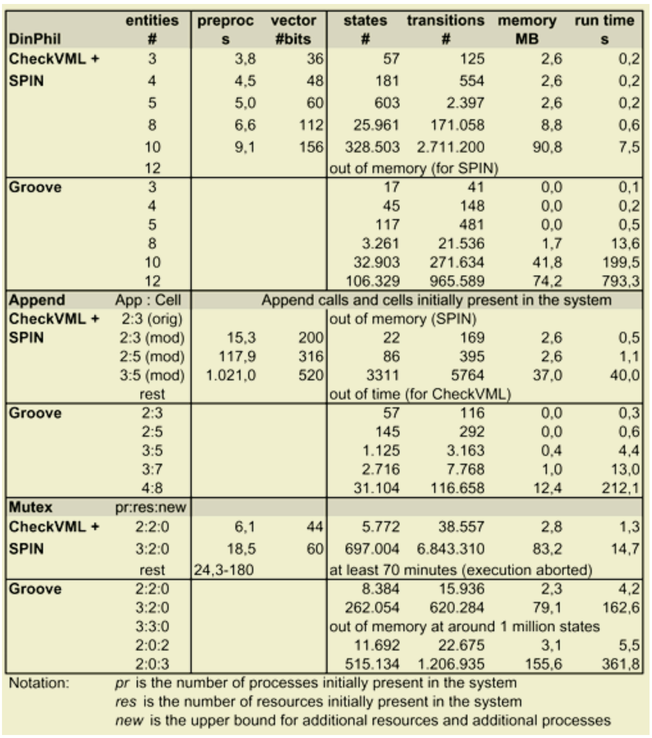
Of the examples presented here, this is the only one for which the state space is actually infinite (there is no upper bound to the numbers of processes and resources). Therefore, an artificial upper bound has to be imposed for the purpose of state space generation.

Results. In Table 2, we compare (a) the number of states traversed by the model checker during a successful verification run, (b) the number of transitions in the (reachable) state space, (c) the size of memory footprint of the state space, and (d) the execution time for the verification run. Furthermore, we also present the preprocessing time required for CheckVML to translate graph transformation systems into SPIN and the size of state vectors in SPIN.

We have done our best to produce the results of both approaches on an equal basis. We briefly list the characteristics of the experiments:

Memory Usage and Run-Time Performance. Experiments were run on a 3 GHz Pentium IV processor with 1 GB of memory. For the GROOVE experiments, Java Virtual Machine was started with an initial memory size

Table 2. Comparison of verification runs for CheckVML+SPIN and GROOVE.



DinPhil	entities #	preproc s	vector #bits	states #	transitions #	memory MB	run time s
CheckVML +	3	3,8	36	57	125	2,6	0,2
SPIN	4	4,5	48	181	554	2,6	0,2
	5	5,0	60	603	2.397	2,6	0,2
	8	6,6	112	25.961	171.058	8,8	0,6
	10	9,1	156	328.503	2.711.200	90,8	7,5
	12			out of memory (for SPIN)			
Groove	3			17	41	0,0	0,1
	4			45	148	0,0	0,2
	5			117	481	0,0	0,5
	8			3.261	21.536	1,7	13,6
	10			32.903	271.634	41,8	199,5
	12			106.329	965.589	74,2	793,3
Append	App : Cell	Append calls and cells initially present in the system					
CheckVML +	2:3 (orig)			out of memory (SPIN)			
SPIN	2:3 (mod)	15,3	200	22	169	2,6	0,5
	2:5 (mod)	117,9	316	86	395	2,6	1,1
	3:5 (mod)	1.021,0	520	3311	5764	37,0	40,0
	rest			out of time (for CheckVML)			
Groove	2:3			57	116	0,0	0,3
	2:5			145	292	0,0	0,6
	3:5			1.125	3.163	0,4	4,4
	3:7			2.716	7.768	1,0	13,0
	4:8			31.104	116.658	12,4	212,1
Mutex	pr:res:new						
CheckVML +	2:2:0	6,1	44	5.772	38.557	2,8	1,3
SPIN	3:2:0	18,5	60	697.004	6.843.310	83,2	14,7
	rest	24,3-180		at least 70 minutes (execution aborted)			
Groove	2:2:0			8.384	15.936	2,3	4,2
	3:2:0			262.054	620.284	79,1	162,6
	3:3:0			out of memory at around 1 million states			
	2:0:2			11.692	22.675	3,1	5,5
	2:0:3			515.134	1.206.935	155,6	361,8

Notation: *pr* is the number of processes initially present in the system
res is the number of resources initially present in the system
new is the upper bound for additional resources and additional processes

of 100 MB and maximum size of 1 GB. Although the space used for the actual storage of the state space is under 200 MB for all the cases reported here, during state space generation the tool heavily relies of caching and limiting the amount of available memory dramatically worsens the run-time performance.

Bounding the State Space. For the mutual exclusion example, we had to put a bound to the state space (as mentioned above). The way this is implemented in both tools is different. In GROOVE, all states which violate the bounding constraint are first generated and added to the transition system, after which the violation is detected and they are ignored for further exploration. In the CheckVML approach, on the other hand, the violation is checked first and hence those states are not generated at all. It turns out that the "spurious" states in the GROOVE results comprise about 85% of the state space and about 25% of the number of transitions.

Activating vs. Creating Nodes. For the original concurrent append example, SPIN failed even on very small examples due to the fact that each node and edge type is dynamic[2]. However, verification times for CheckVML +

[2] CheckVML generates the cross-product of all nodes and edges in the preprocessing phase even though the number of edges are only linear in the number of nodes.

SPIN could be reduced by a modeling trick, i.e. altering the models and the rules by adding an *isActive* attribute for each node type and only activating a node by changing this attribute instead of "real" node creation. This way, many graph elements that were originally dynamic are turned into static elements and thus abstracted by CheckVML during preprocessing. This example thus also demonstrated some pros and contras of graph attributes. However, the experimental results for the two approaches are not directly comparable (as denoted by the "(mod)" postfix after the append test cases in Table 2).

Evaluation. Based on Table 2, we come to the following overall conclusions:

– The space needed to store the transition system generated by both tools is comparable. Yet the techniques are very different: for GROOVE it is based on storing the differences between successive states, in terms of nodes and edges added and removed, whereas SPIN (and hence CheckVML) stores states as bit vectors that encode the entire graphs.
– The time needed to generate the states spaces is in a different order of magnitude: on the cases reported here, CheckVML typically takes under a tenth of the time that GROOVE does. For this we offer three possible explanations: (a) SPIN clearly shows the benefits of a more mature technology: over a decade of research has gone into improving its implementation. (b) Over the years, SPIN has been heavily optimized towards its implementation in C, whereas GROOVE has been implemented entirely in Java. (c) The approach taken by GROOVE, involving explicit graph matching and graph isomorphism checks, is inherently more complex.
– For each of the problems studied the GROOVE approach can handle a larger dimension than the CheckVML approach (which dimension is unquestionably significant for the append and mutual exclusion examples). This shows that the potential advantages of the approach, in terms of symmetry checking and dealing with dynamic allocation, also really show up in practice.

6 Conclusions

In the paper, we tackled the problem of model checking graph transformation systems by two different approaches. CheckVML exploits traditional model checking techniques for verification by translating graph transformation systems into SPIN, an off-the-shelf model checker. GROOVE, on the other hand, uses the core concepts of graphs and graph transformations all the way through during model checking.

We compared the two approaches on three case studies having essentially different characteristics concerning the dynamic and symmetric nature of the problem. Our overall conclusion is the following:

– If the problem analyzed lends itself well to be modeled in SPIN; that is, if dynamic allocation and/or symmetries are limited, it is to be expected that the CheckVML approach will always remain superior.

– On the other hand, for problems that are inherently dynamic, the GROOVE approach is a promising alternative.

Our conclusions also imply certain directions for future work. Obviously, CheckVML would yield a much more succinct state vector if further constraints on the metamodels (such as multiplicities) were handled in the preprocessing phase. For GROOVE, it is an interesting issue to make isomorphism checks optional (thus serving as an intelligent compression technique). However, the main line of research should find sophisticated abstraction techniques especially for infinite state graph transformation systems.

References

1. P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR '04*. Springer-Verlag, 2004. LNCS, to appear.
2. P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: First International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 14–29. Springer, Barcelona, Spain, 2002.
3. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, vol. 26(3/4):pp. 241–265, 1996.
4. D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In R. Baeza-Yates, U. Montanari, and N. Santoro (eds.), *Foundations of Information Technology in the Era of Network and Mobile Computing*, vol. 223 of *IFIP Conference Proceedings*, pp. 435–447. Kluwer Academic Publishers, 2002.
5. F. L. Dotti, L. Foss, L. Ribeiro, and O. M. Santos. Verification of object-based distributed systems. In *Proc. 6th International Conference on Formal Methods for Open Object-based Distributed Systems*, vol. 2884 of *LNCS*, pp. 261–275. 2003.
6. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.
7. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *In [18]*, chap. Algebraic Approaches to Graph Transformation – Part II: Single pushout approach and comparison with double pushout approach, pp. 247–312. World Scientific, 1997.
8. C. Ermel, M. Rudolf, and G. Taentzer. *In [6]*, chap. The AGG-Approach: Language and Tool Environment, pp. 551–603. World Scientific, 1999.
9. R. Heckel. Compositional verification of reactive systems specified by graph transformation. In *Proc. FASE: Fundamental Approaches to Software Engineering*, vol. 1382 of *LNCS*, pp. 138–153. Springer, 1998.
10. R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: First International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 161–176. Springer, Barcelona, Spain, 2002.
11. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, vol. 23(5):pp. 279–295, 1997.
12. U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*. ACM Press, Limerick, Ireland, 2000.

13. A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti (eds.), *Proceedings of the* 3rd *Workshop on Automated Verification of Critical Systems*, Technical Report DSSE–TR–2003–2, pp. 150–160. University of Southampton, 2003.
14. A. Rensink. Canonical graph shapes. In D. A. Schmidt (ed.), *Programming Languages and Systems – European Symposium on Programming (ESOP)*, vol. 2986 of *LNCS*, pp. 401–415. Springer-Verlag, 2004.
15. A. Rensink. Graphs for object-oriented verification, 2004. See `http://www.cs.utwente.nl/~groove`.
16. A. Rensink. The GROOVE simulator: A tool for state space generation. In M. Nagl, J. Pfalz, and B. Böhlen (eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, vol. 3063 of *LNCS*. Springer-Verlag, 2004.
17. A. Rensink. State space abstraction using shape graphs. In *Automatic Verification of Infinite-State Systems (AVIS)*, ENTCS. Elsevier, 2004. To appear.
18. G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
19. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, vol. 24(3):pp. 217–298, 2002.
20. Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In P. Stevens, J. Whittle, and G. Booch (eds.), *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, vol. 2863 of *LNCS*, pp. 92–95. Springer, San Francisco, CA, USA, 2003.
21. A. Schürr, S. E. Sim, R. Holt, and A. Winter. The GXL Graph eXchange Language. `http://www.gupro.de/GXL/`.
22. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In J. Padberg (ed.), *UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques*, vol. 44 (4) of *ENTCS*. 2001.
23. D. Varró. Towards symbolic analysis of visual modelling languages. In P. Bottoni and M. Minas (eds.), *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, vol. 72 (3) of *ENTCS*, pp. 57–70. Elsevier, Barcelona, Spain, 2002.
24. D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, vol. 3(2):pp. 85–113, 2004.