Model Checking Dynamic States in GROOVE

Harmen Kastenberg* and Arend Rensink

Department of Computer Science, University of Twente,
P.O. Box 217, AE 7500, Enschede, The Netherlands
{h.kastenberg, rensink}@cs.utwente.nl

Abstract. Much research has been done in the field of model-checking complex systems (either hardware or software). Approaches that use explicit state modelling mostly use bit vectors to represent the states of such systems. Unfortunately, that kind of representation does not extend smoothly to systems in which the states contain values from a domain other than primitive types, such as reference values commonly used in object-oriented systems.

In this paper we report preliminary results on applying CTL model checking on state spaces generated using graph transformations. The states of such state spaces have an internal graph structure which makes it possible to represent complex system states without the need to know the exact structure beforehand as when using bit vectors.

1 Introduction

Verifying complex systems is a big field of research. For hardware systems, model checking techniques have proven to be quite successful. Lately, researchers are trying to also apply model-checking techniques for the verification of software systems.

In the Groove-project we focus on the use of model checking techniques for verifying object-oriented systems, where the states of the system are modelled as *graphs*, instead of bit vectors as in most explicit state representing approaches. We think 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 state spaces on which we perform the model checking process are generated from so-called graph production systems using the GROOVE Simulator [9]. This results in a so-called graph transition system. These are then translated to ordinary Kripke structures after which we are able apply standard CTL model checking.

* The author is employed in the GROOVE project funded by the Dutch NWO (project number 612.000.314).

A. Valmari (Ed.): SPIN 2006, LNCS 3925, pp. 299–305, 2006.

2 State Space Generation

In our approach we model systems by representing their states as graphs and their behaviour as graph transformations [13]. In this work, a *graph* G consists of a finite set N of *nodes* and a finite set $E \subseteq N \times L \times N$ of *edges* (where L is a global set of labels). We use $\mathcal{G}$ to denote the set of all graphs, ranged over by G, H. Fig. 2.1 shows an example graph representing a specific state of a circular buffer containing three cells.[1]

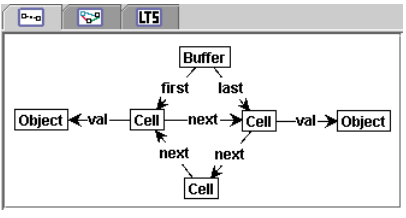


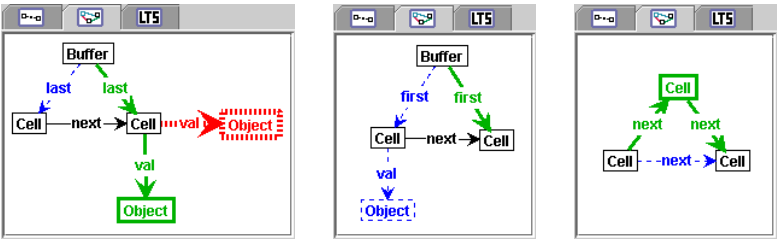
Fig. 2.1. A circular buffer having 2 filled cells out of 3

The state space representing the entire behaviour of the system can be generated from a *graph production system* (GPS), which consists of a graph I representing the initial state of the system and a set of *graph transformation rules* $\mathcal{R}$. A graph transformation rule specifies how the system evolves when going from one state to another. A graph transformation rule $p \in \mathcal{R}$ is identified by its *name* ($N_p \in \mathcal{N}$, where $\mathcal{N}$ is a global set of rule names) and consists of a *left-hand-side graph* (L_p), a *right-hand side graph* (R_p), and a set of so-called *negative application conditions* (NAC_p, which are supergraphs of L_p) [4]. The application of a graph transformation rule p transforms a graph G, the *source graph*, into a graph H, the *target graph*, by looking for an occurrence of L_p in G (specified by a graph matching m that cannot be extended to an occurrence of any graph in NAC_p) and then replacing that occurrence with R_p, resulting in H. Such a rule application is denoted as $G \xrightarrow{p,m} H$. A precise technical specification of the graph transformation process can be found in [13, 4].

Fig. 2.2 shows three screen-shots from our tool (see below) displaying three graph transformation rules: put for inserting a newly created object into the buffer, get for getting an object out of the buffer (and deleting it), and extend for enlarging the capacity of the buffer with one. Note that these transformation rules specify the behaviour of a circular buffer. This means that performing a put and get operation subsequently, moves both the first and the last pointer one cell further. Performing an equal number of puts and gets (without extending the buffer) results in isomorphic states (which are identified within the tool).

The different shapes (and colours) of the nodes and edges refer to the different roles of the elements within the rule. The thin solid elements (black in a coloured

[1] In order to improve the readability of the graphs, we show the labels of self-edges as labels of the corresponding nodes.



(a) put rule. (b) get rule. (c) extend rule.

Fig. 2.2. Graph transformation rules specifying the behaviour of the circular buffer

print-out) are part of both L and R. They need to be present in the source graph in order for the rule to apply and will be preserved during transformation. The thin dashed elements (blue) are also part of L but not of R, and will be removed. The solid fat gray elements (green) are part of R but not of L and will be created. The dashed fat gray elements (red) represent the NACs, whose presence in the source graph prevent the rule from being applied.

Each GPS $P = \langle \mathcal{R}, I \rangle$ specifies a (possibly infinite) state space which can be generated by repeatedly applying the graph transformation rules on the states, starting from the initial state I. This results in a *graph transition system* (GTS):

Definition 1 (graph transition system). *The* graph transition system $T = \langle S, \rightarrow, I \rangle$ *generated by* $P = \langle \mathcal{R}, I \rangle$ *consists of a set* S *of states, which are actually graphs* ($S \subseteq \mathcal{G}$)*; a transition relation* $\rightarrow \subseteq S \times \mathcal{R} \times [\mathcal{G} \rightarrow \mathcal{G}] \times S$*, such that* $\langle G, p, m, H \rangle \in \rightarrow$ *iff there is a rule application* $G \xrightarrow{p,m} H'$ *with* H' *isomorphic to* H*; and an initial state* $I \in S$*.*

The graph transformation process is implemented in the Groove Simulator [9]. This tool is implemented in Java, and currently consists of 18 packages comprising approximately 400 classes, and 75,000 lines of code. The tool can handle arbitrary

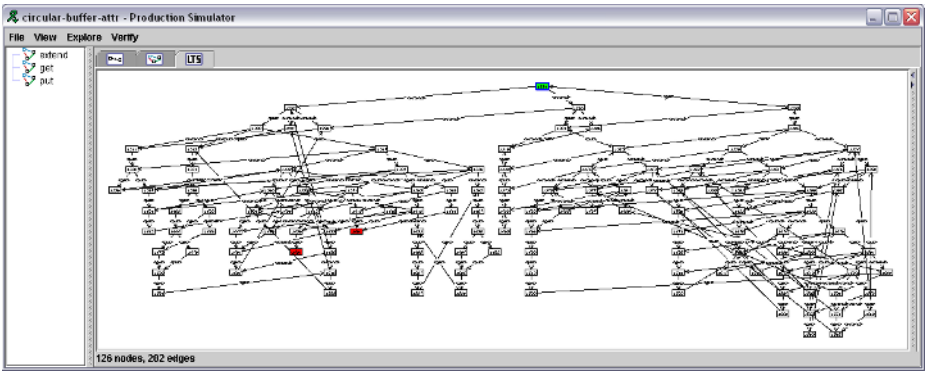


Fig. 2.3. State space of a circular buffer with capacity extending to 5

GPSs, but can obviously only generate a finite part of the corresponding graph transition system. Some performance figures were reported in [11]; as an indication, in its current form the tool can handle up to 200,000 states for average graph size of 50 nodes. Two intrinsically complex parts of the state space generation are: finding occurrences of left hand sides, and determining isomorphism of states.

Fig. 2.3 shows a finite part of the graph transition system for the transformation rules from Fig. 2.2 and the initial graph of Fig. 2.1, where the states are limited to those where the number of buffer cells is 5. The resulting state space consists of 126 states and 282 transitions.

3 CTL Model Checking

In the approach reported here, we have chosen to express properties in the temporal logic CTL [3]. The main reason for choosing CTL and not, for example, LTL, is the simplicity of the former, both in terms of complexity (the model checking problem for CTL is well known to be linear in both the size of the state space and the size of the formula) and in terms of the actual algorithm.

In order to perform model checking on the graph transition systems generated in the previous section we need to translate them to *Kripke structures*, which are defined over a finite set AP of *atomic propositions*.

Definition 2 (Kripke structure). *A* Kripke structure $K = \langle S, \rightarrow, I, L \rangle$ *consists of a set S of states, a* total *transition relation $\rightarrow \subseteq S \times S$, a set $I \subseteq S$ of initial states, and a labelling function $L : S \rightarrow 2^{AP}$, which maps each state to the subset of atomic propositions holding in that state.*

When translating a GTS T to a Kripke structure K_T, two issues need to be taken care of: (1) $\rightarrow_T$ must be made *total* (if this is not yet the case) and (2) the labelling function L_K must be defined. As atomic propositions we use the rule names, $\mathcal{N}$. Thus, a GTS T gives rise to a Kripke structure K_T such that:

$$
\begin{aligned}
S_K &= S_T \\
I_K &= \{S_I\} \\
\rightarrow_K &= \{\langle G, H \rangle \mid \exists p, m : \ G \xrightarrow{p,m}_T H\} \cup \{\langle G, G \rangle \mid \nexists p, m, H : \ G \xrightarrow{p,m}_T H\} \\
L_K(G) &= \{N_p \mid \exists m, H : G \xrightarrow{p,m}_T H\}, \text{ for all } G \in S_K
\end{aligned}
$$

From the construction process described above it becomes clear that the labelling function of the resulting Kripke structure, in graph transformation terms, actually maps each state on the set of names of the graph transformation rules that were applicable in that state. This means that for each transformation rule p, L_p and NAC_p constitute a property of graphs that can be used as an atomic proposition named N_p[2]. In the special case where L_p and R_p are identical, the rule actually specifies a *state property* instead of a graph transformation, since such rules have no structural effect on any state.

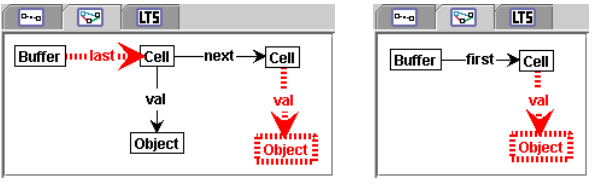
[2] In [10] we show that properties specified this way may correspond precisely to a certain fragment of First-Order logic.

Two example properties to check for on the circular buffer example are:

$$\mathsf{AG}(\neg \mathsf{gap}) \tag{1}$$

$$\mathsf{AG}(\mathsf{EF}(\mathsf{empty})) \tag{2}$$

Property 1 is a safety property specifying that the buffer may not contain a *gap*, which is an empty cell following a non-empty cell that is not the last cell of the buffer. Fig. 3.1 (a) specifies the gap-proposition in the form of a rule (with identical left and right hand side). Property 2 is a liveness property specifying that the state representing the empty buffer must reachable infinitely often. The buffer is empty when the first cell does not contain a value, as shown in Fig. 3.1 (b).



(a) gap. (b) empty.

Fig. 3.1. Graph structures as properties

It turns out that the system of Fig. 2.3 actually does *not* satisfy Property 1. This is because we have not specified the extend-rule correctly: it puts no constraints on the places where the buffer may be extended, and hence may well introduce a gap. After fixing this, the system indeed satisfies both properties.

Results. In order to compare our tool with existing ones, we also implemented our running example as a (more or less) equivalent SPIN-program [7]. A naive translation results in a SPIN-program using a bit-array (with the maximum allowed capacity as its length) storing 1's (representing full cells) and 0's (representing empty cells). A more sophisticated SPIN-program can use the built-in channels to store the buffer values. Note that this no longer is a real circular buffer. In both cases we implement the possible operations as being atomic. In the naive implementation, the first and last pointer travel along the array resulting in many more states. In the sophisticated implementation there is no need for a first and last pointer.

Statistics about the state spaces generated by the three programs are given in Table 3.2. In this table we list for each implementation the number states, the state space generation time (GT) and the memory needed to store the states. From this table we can conclude that GROOVE cannot compete with SPIN regarding time-performance. The main reason for this is because checking for isomorphic graphs and constructing the graph matchings is very expensive; indeed, for the buffer of 200 cells, over 90% of the time is spent in isomorphism checking. However, isomorphism does result in automatic symmetry reduction, as can be seen be comparing the numbers of states in the GROOVE and naive SPIN implementations. Concerning memory usage for state storage, both tools perform

Table 3.2. Performance statistics

	GROOVE			SPIN naive			SPIN smart		
Max cap.	States	GT (s)	Memory (MB)	States	GT (s)	Memory (MB)	States	GT (s)	Memory (MB)
25	345	1.5	< 1	5,845	< 1	0.2	345	< 1	< 1
50	1,320	6.9	< 10	44,195	< 1	3.1	1,320	< 1	< 1
100	5,145	60.6	< 20	343,395	< 1	42.6	5,145	< 1	0.6
200	20,295	636.5	< 20	2.7+e6	20	606.3	20,295	< 1	4.5

comparable. From the object-oriented point of view, the example showed that GROOVE provides a natural way of dealing with reference-pointers, whereas the encoding in SPIN resorts to built-in static data types.

As mentioned before, we have implemented the standard CTL algorithm (with backwards state traversal). Currently, the verification process is performed sequentially after the state space generation. By combining both phases, so called *on-the-fly* model checking, we could also run the algorithm on graph production systems that yield potentially infinite state spaces, and get a result if it can be computed on a finitely representable fragment of the graph transition system.

4 Conclusion

We have shown how to apply CTL model checking on state spaces generated from graph production systems. The innovation in this approach lies not in the model checking itself but in the use of graphs for explicit dynamic state representation, which, as we have argued before, gives rise to an alternative to bit vectors that is potentially more flexible. We have shown some statistics on how our tool performs when compared to SPIN. The choice of CTL is not important in this respect.

Within the area of software model checking, a large number of other software verification tools have been developed, e.g. Java PathFinder [5], BLAST [6], SLAM [1], and MAGIC [2]. The last three focus on the verification of C programs instead of OO-systems like our tool and Java PathFinder. Representing states as graphs, instead of using arrays and lists, as is done in Java PathFinder, provides a more natural way of dealing with reference values, and symmetry reduction boils down to checking for isomorphic graphs. While Java PathFinder uses the byte code of a program, we represent the source code as graphs, taking the abstract syntax of the language as a starting point [8].

In the future we plan to do more experiments using the technique described in this paper. Next to that, there is a lot of further work to be done on improving the state space generation part. For one thing, currently no advantage is taken whatsoever of the potential for partial order reduction. In the running example of this paper, partial order reduction would already pay off, because the put- and get-rules are actually provably *confluent*. Alternatively, in [12] we describe an *abstraction* technique for graph transformation that results in smaller (in fact, finite) state spaces, at the price of false negatives in the model checking phase.

Acknowledgements

We would like to thank the anonymous referees for their detailed comments and constructive suggestions.

References

1. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–3. ACM Press, 2002.
2. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Softw. Eng.*, 30(6):388–402, 2004.
3. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1982.
4. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.
5. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.
6. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In T. Ball and S. K. Rajamani, editors, *SPIN Workshop on Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.
7. G. J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2003.
8. H. Kastenberg, A. Kleppe, and A. Rensink. Engineering object-oriented semantics using graph transformations. Technical report, Department of Computer Science, University of Twente, 2005. Pre-final version available at `http://www.cs.utwente.nl/~rensink/papers/taal-draft.pdf`.
9. A. Rensink. The GROOVE Simulator: A tool for state space generation. In J. L. Pfaltz, 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, 2004.
10. A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335. Springer, 2004.
11. A. Rensink. Time and space issues in the generation of graph transition systems. In *International Workshop on Graph-Based Tools (GraBaTs)*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 127–139, 2005.
12. A. Rensink and D. Distefano. Abstract graph transformation. In *International Workshop on Software Verification and Validation (SVV)*, Electronic Notes in Theoretical Computer Science, 2005. To appear. Technical report version: CTIT TR–CTIT–05–04, University of Twente.
13. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.