UPPAAL/DMC – Abstraction-Based Heuristics for Directed Model Checking

Sebastian Kupferschmid[1], Klaus Dräger[2], Jörg Hoffmann[3], Bernd Finkbeiner[2], Henning Dierks[4], Andreas Podelski[1], and Gerd Behrmann[5]

[1] University of Freiburg, Germany
{kupfersc,podelski}@informatik.uni-freiburg.de
[2] Universität des Saarlandes, Saarbrücken, Germany
{draeger,finkbeiner}@cs.uni-sb.de
[3] Digital Enterprise Research Institute, Innsbruck, Austria
joerg.hoffmann@deri.org
[4] OFFIS, Oldenburg, Germany
dierks@offis.de
[5] Aalborg University, Denmark
behrmann@cs.aau.dk

Abstract. UPPAAL/DMC is an extension of UPPAAL which provides generic heuristics for directed model checking. In this approach, the traversal of the state space is guided by a heuristic function which estimates the distance of a search state to the nearest error state. Our tool combines two recent approaches to design such estimation functions. Both are based on computing an abstraction of the system and using the error distance in this abstraction as the heuristic value. The abstractions, and thus the heuristic functions, are generated fully automatically and do not need any additional user input. UPPAAL/DMC needs less time and memory to find shorter error paths than UPPAAL's standard search methods.

1 Introduction

UPPAAL/DMC is a tool that accelerates the detection of error states by using the directed model checking approach [4,5]. Directed model checking tackles the state explosion problem by using a *heuristic function* to influence the *order* in which the search states are explored. A heuristic function h is a function that maps states to integers, estimating the state's distance to the nearest error state. The search then gives preference to states with lower h value. There are many different ways of doing the latter, all of which we consider the wide-spread method called *greedy search* [8]. There, search nodes are explored in ascending order of their heuristic values. Our empirical results show that this can drastically reduce memory consumption, runtime, and error path length.

Our tool combines two recent approaches to design heuristic functions. Both are based on defining an abstraction of the problem at hand, and taking the heuristic value to be the length of an abstract solution. It is important to note that both techniques are *fully automatic*, i.e., no user intervention is needed to generate the heuristic function. UPPAAL has a built-in heuristic mode, but the specification of the heuristic is entirely up to the user. Inventing a useful heuristic is a tedious job: it requires expert knowledge and a huge amount of time.

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 679–682, 2007.

2 Heuristics

The next two sections give a brief overview of the abstractions used to build our heuristics, and how heuristic values are assigned to search states.

2.1 Monotonicity Abstraction

Our first heuristic [7] adapts a technique from AI Planning, namely *ignoring delete lists* [1]. The idea of this abstraction is based on the simplifying assumption that *every state variable, once it obtained a value, keeps that value forever*. I.e., the value of a variable is no longer an element, but a *subset* of its domain. This subset grows monotonically over transition applications – hence the name of the abstraction.

When applying the monotonicity abstraction to a system of timed automata, then each automaton will (potentially) be in several locations in a state. The system's integer variables will have several possible values in a state. So far clocks are not included in the computation of heuristic values. If we included clocks in the obvious way, every guard or invariant involving a clock would be immediately satisfied. The reason for this is that clock value sets quickly subsume all possible time points.

Our heuristic h^{ma} assigns to each state encountered during search a heuristic value by solving an abstract problem. Such an abstract problem is obtained by applying the monotonicity abstraction to the current state. The length of a solution found in this abstraction is then used as the heuristic estimate for the state's distance to the nearest error state. In a nutshell, an abstract solution is computed by iteratively applying all enabled transitions to the initial abstract state (the state for which we want to estimate the distance), until either the enlarged state subsumes an error state, or a fixpoint is reached. In the former case, an abstract solution can be extracted by backtracking through the state enlargement steps. In case of reaching a fixpoint, we can exclude this state from further exploration: the monotonicity abstraction induces an over-approximation, i.e. so if there is no abstract error path, then there is no real one either.

2.2 Automata-Theoretic Abstraction

The second heuristic [3] aims at a close representation of the process synchronization required to reach the error. Each process is represented as a finite-state automaton. The heuristic h^{aa} estimates the error distance $d(s)$ of a system state s as the error distance of the corresponding abstract state $\alpha(s)$ in an abstraction that approximates the full product of all process automata. The approximation of the product of a set of automata is computed incrementally by repeatedly selecting two automata from the current set and replacing them with an abstraction of their product. To avoid state space explosion, the size of these intermediate abstractions is limited by a preset bound N: to reach a reduction to N states, the abstraction first merges bisimilar states and then states whose error distance is already high in the partial product. In this way, the precision of the heuristic is guaranteed to be high in close proximity to the error, and can, by setting N, be fine-tuned for states further away from the error. In our experiments, fairly low values of N, such as $N = 100$, already significantly speed up the search for the error, and therefore represent a good trade-off between cost and precision.

3 Results

We compare the performance of UPPAAL/DMC's[1] greedy search and UPPAAL's randomized depth first search (rDF), which is UPPAAL's most efficient standard search method across many examples. The results for rDF in Table 1 are averaged over 10 runs. The C_i examples stem from an industrial case study called "Single-tracked Line Segment" [6] and the M_i examples come from another case study, namely "Mutual Exclusion" [2]. An error state was made reachable by increasing an upper time bound in each example.

The results in Table 1 clearly demonstrate the potential of our heuristics. The heuristic searches consistently find the error paths much faster. Due to the reduced search space size and memory requirements, they can solve all problems. At the same time, they find, by orders of magnitude, *much* shorter error paths in *all* cases.

Table 1. Experimental results of UPPAAL's rDF and UPPAAL/DMC's greedy search with h^{ma} and h^{aa}. The results are computed on an Intel Xeon with 3 Ghz and 4 GB of RAM. Dashes indicate out of memory.

	runtime in s			explored states			memory in MB			trace length		
Exp	rDF	h^{ma}	h^{aa}	rDF	h^{ma}	h^{aa}	rDF	h^{ma}	h^{aa}	rDF	h^{ma}	h^{aa}
M_1	0.8	0.1	0.2	29607	5656	12780	7	1	11	1072	169	74
M_2	3.1	0.3	0.9	118341	30742	46337	10	11	11	3875	431	190
M_3	2.8	0.2	0.8	102883	18431	42414	9	10	11	3727	231	92
M_4	12.7	0.8	1.9	543238	76785	126306	22	13	14	15K	731	105
C_1	0.8	0.2	0.5	25219	2339	810	7	9	11	1065	95	191
C_2	1.0	0.3	1.0	65388	5090	2620	8	10	19	875	86	206
C_3	1.1	0.5	1.1	85940	6681	2760	10	10	19	760	109	198
C_4	8.4	2.5	1.8	892327	40147	25206	43	11	23	1644	125	297
C_5	72.4	13.2	4.0	8.0e+6	237600	155669	295	21	28	2425	393	350
C_6	–	10.1	14.9	–	207845	1.2e+6	–	20	67	–	309	404
C_7	–	169.0	162.4	–	2.7e+7	1.3e+7	–	595	676	–	1506	672
C_8	–	14.5	155.3	–	331733	1.2e+7	–	23	672	–	686	2210
C_9	–	1198.0	1046.0	–	1.3e+8	3.6e+7	–	2.5G	1.6G	–	18K	1020

Other heuristics, proposed by Edelkamp et al. [4,5] in the context of SPIN are based on graph distances. The underlying abstraction of these heuristics preserves only edges and locations of an automata system. For an automaton a let $d(a)$ be the distance of a's start location to its target location. Then, the h^{gd}_{max} heuristic is defined as $\max_a d(a)$. The h^{gd}_{sum}heuristic is defined as $\sum_a d(a)$.

Note that h^{gd}_{max} and h^{gd}_{sum} are rather crude approximations of the systems semantics. For example, they completely ignore variables and synchronization. In contrast, the

[1] Two different versions of UPPAAL/DMC (both Linux executables) are available under `http://www.informatik.uni-freiburg.de/~kupfersc/uppaal_dmc/`. One is optimized for Intel Pentium 4 processors, the other one was compiled with default optimization. The page also provides a short description of the used benchmarks, and *all* used model and query files.

h^{ma} and h^{aa} heuristics do *not* do this. Our approximations are more costly, i.e. one call of h^{ma} or h^{aa} takes more runtime than one call of h^{gd}_{max} or h^{gd}_{sum}. The additional effort typically pays off: for example, in the case studies shown in Table 1, greedy search with $\max_a d(a)$ and $\sum_a d(a)$ performs only slightly better than rDF, and much worse than our heuristics; e.g. it cannot solve any of C_6, C_7, C_8, and C_9.

4 Outlook

The most important piece of future work is to explore the value of our tool in the abstraction refinement life cycle. The basic idea is to use heuristics to address the intermediate iterations where (spurious) errors still exist. As our results show, this has the potential to speed up the process *and* yield shorter, and thus more informative error paths. Hence, our technique for error detection will be able to help with actual *verification*.

Acknowledgments

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `http://www.avacs.org/` for more information.

References

1. Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129 (1–2):5–33, 2001.
2. Henning Dierks. Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.
3. Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In *Proceedings of the 13th International SPIN Workshop on Model Checking of Software*, 2006.
4. Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with HSF-Spin. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 57–79, 2001.
5. Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 2004.
6. Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, and Alexander Baer. The UniForM Workbench, a universal development environment for formal methods. In *FM'99 – Formal Methods*, volume 1709 of *LNCS*, pages 1186–1205. Springer, 1999.
7. Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In *Proceedings of the 13th International SPIN Workshop on Model Checking of Software*, 2006.
8. Judea Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley, 1984.