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 that 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 [1, 2]. 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* [3]. 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.

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 [4] adapts a technique from AI Planning, namely *ignoring delete lists* [5]. 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 backtracing 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 [6] aims at a close representation of the process synchronisation 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 greedy search and UPPAAL's randomised 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" [7] and the M_i examples come from another case study, namely "Mutual Exclusion" [8]. 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 heuristis, proposed by Edelkamp et al. [1, 2] 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 synchronisation. In contrast, the 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*.

4.1 Availability of the Tool

At `http://www.informatik.uni-freiburg.de/~kupfersc/uppaal_dmc/`, two Linux executables of UPPAAL/DMC are available. The first version is optimised for Intel Pentium 4 processors, the other one was compiled with default optimisation. The page also provides a short description of the used benchmarks, and *all* used model and query files.

References

1. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit model checking with HSF-Spin. In: Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'2001). (2001) 57–79
2. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit-state model checking in the validation of communication protocols. International Journal on Software Tools for Technology Transfer (2004)
3. Pearl, J.: Heuristics: Intelligent search strategies for computer problem solving. Addison-Wesley (1984)
4. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In: Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN'2006). (2006)
5. Bonet, B., Geffner, H.: Planning as heuristic search. Artificial Intelligence **129**(1–2) (2001) 5–33
6. Dräger, K., Finkbeiner, B., Podelski, A.: Directed model checking with distance-preserving abstractions. In: Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN'2006). (2006)
7. Krieg-Brückner, B., Peleska, J., Olderog, E.R., Baer, A.: The UniForM Workbench, a universal development environment for formal methods. In Wing, J.M., Woodcock, J., Davies, J., eds.: FM'99 – Formal Methods. Volume 1709 of LNCS., Springer (1999) 1186–1205
8. Dierks, H.: Comparing model-checking and logical reasoning for real-time systems. Formal Aspects of Computing **16**(2) (2004) 104–120
9. Dierks, H.: Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics. (2005) Habilitation thesis.
10. Olderog, E.R., Dierks, H.: Moby/RT: A tool for specification and verification of real-time systems. Journal of Universal Computer Science **9**(2) (2003) 88–105

A Appendix

A.1 Outline of the oral presentation

In the talk, we are going to demonstrate our tool on a typical example from our benchmark suite. We will discuss what is difficult about these examples and how the difficulties are addressed by UPPAAL/DMC. We believe that this case study is interesting in its own right. It is publicly available from the UPPAAL/DMCwebpage (see Section 4.1) as a point of reference for the evaluation of other, forthcoming, tools.

The oral presentation of UPPAAL/DMC will roughly be as follows (see Fig. 1). First we will recall what is greedy and A^* search and how these search methods use heuristics to guide the search. Hereinafter we will explain our abstractions and how heurisitc values are computed with concrete example. After explaining the used benchmarks we will compare standard UPPAAL with UPPAAL/DMC. Finaly, we will point out how UPPAAL/DMC can be used for abstraction refinement and talk about the next steps of UPPAAL/DMC.

Fig. 1. preliminary structure of the talk

- Introduction
- Motivation – why DMC at all?
- Heuristic Search
  - Explaination of greedy and A^* search on some small example
- Heuristic Search applied to Model Checking
- Our framework for fully automatically generated heuristics, based on abstractions
  - The Monotonicity Abstraction
  - Demonstration of the effect of the monotonicity abstraction on a small example
  - An Automata-Theoretic Abstraction
  - Demonstration of the effect of this abstraction on a small example
- Benchmarks
  - Short comments on used case studies (see A.2)
  - Comparison UPPAAL and UPPAAL/DMC
- Demo:
  - we are going to demonstrate our tool on a typical example from our benchmark suite. We will discuss what is difficult about these examples and how the difficulties are addressed by UPPAAL/DMC. We believe that this case study is interesting in its own right. It is publicly available from the UPPAAL/DMCwebpage (see 4.1) as a point of reference for the evaluation of other, forthcoming, tools.
- Future Work and an application to Abstraction Refinement

A.2 Case Study Single-tracked Line Segment

The examples C_i, $i = 1, \ldots, 9$, come from a case study called “Single-tracked Line Segment”. This study stems from an industrial project partner of the UniForM-project

[7]. The problem was to design a distributed real-time controller for a segment of tracks where trams share a piece of track. Figure 2 sketches the system architecture.

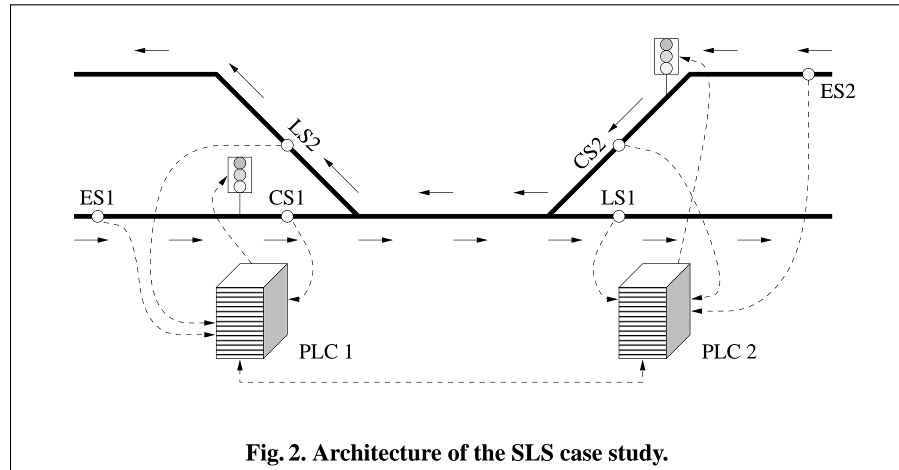

Fig. 2. Architecture of the SLS case study.

The railway lines are marked by thick lines and share a short part. The short arrows describe the directions of trains. The movements of trains are detected by six sensors, three for each direction. The arrival of trains is recognised by entrance sensors ES1 and ES2. The sensors CS1 and CS2 check whether a train enters the critical section, while LS1 and LS2 recognise leaving trains. The information delivered by the sensors are processed by two computing devices called PLCs. These PLCs notice the movements of trains and compute which direction is allowed to use the critical section.

A distributed controller was modelled in terms of PLC-Automata [9], which is an automata-like notation for real-time programs. The system diagram of the controller for the SLS is given in Fig. 3. According to their tasks, we can distinguish five different automata in the system:

- First the signals of the sensors are filtered by six components. The purpose of these filters is to compensate some inherent unreliabilities of the sensor hardware.
- Four counters accumulate the information about passing trains produced by the filters. For each zone of interest we need a counter to determine the number of trains in this zone. There is for each direction a *waiting zone* (zone between the entrance sensor and the critical sensor) and there is a *critical zone* (zone between the critical sensor and the leaving sensor).
  If a counter recognises that the number of trains in its corresponding zone leaves the plausible range, which is $[0,\ 2]$ for the SLS, an error signal is raised.
- One component summarises the error signals of all filters and counters.
- The permissions which direction is allowed to enter the critical section are computed in the main controller. The decision depends on the current values of the counter and the current state of the main controller.

– Two automata produce the signal for the traffic lights for each direction. To this end they need the information whether there is a train in its waiting zone, whether its direction has got the permission to enter the critical section, and whether there is an error in the system or not.

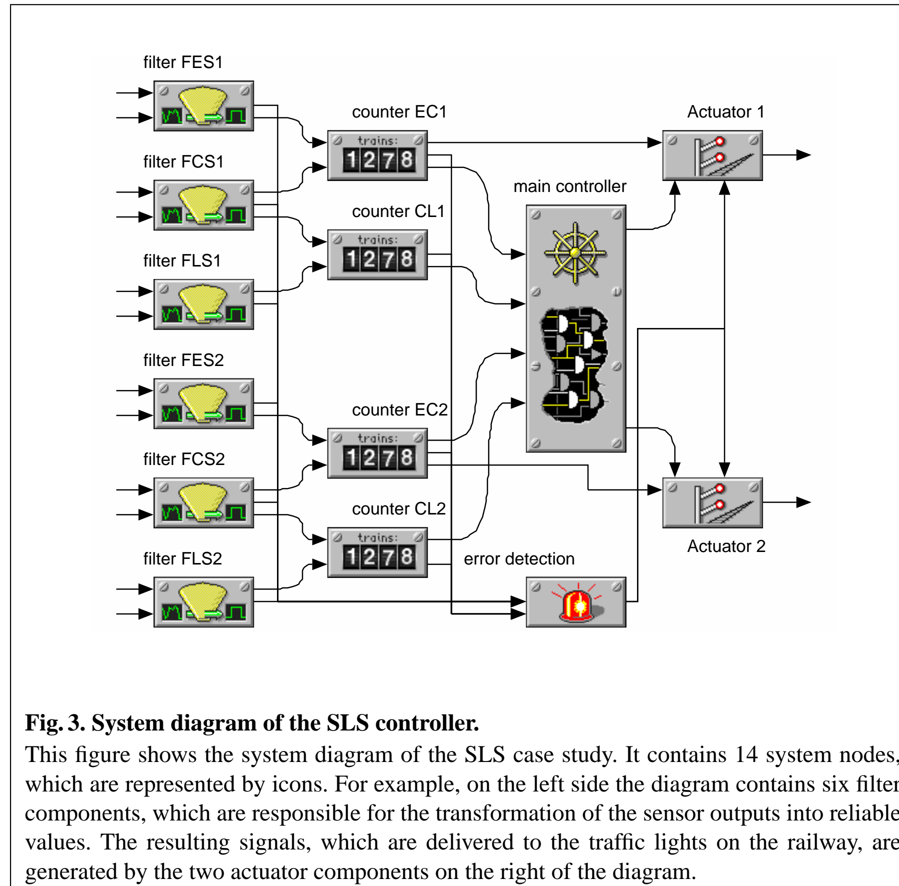

Fig. 3. System diagram of the SLS controller.
This figure shows the system diagram of the SLS case study. It contains 14 system nodes, which are represented by icons. For example, on the left side the diagram contains six filter components, which are responsible for the transformation of the sensor outputs into reliable values. The resulting signals, which are delivered to the traffic lights on the railway, are generated by the two actuator components on the right of the diagram.

This system of PLC-Automata can be transformed into (abstractions of) their semantics in terms of timed automata with the tool Moby/RT [10]. For the evaluation of our approach we choose the property that never both directions are given permission to enter the shared segment simultaneously. This property is ensured by 3 PLC-Automata (main controller, actuator 1, actuator 2) of the whole controller, and we injected an error by manipulating a delay so that the asynchronous communication between these automata is faulty. In Moby/RT abstractions are offered for the translation into the timed automata. The given set of PLC-Automata has eight input variables. We constructed nine models with increasing size by decreasing the number of abstracted inputs.