

Faster than UPPAAL ?

(Tool Paper)

Sebastian Kupferschmid, Martin Wehrle, Bernhard Nebel, and Andreas Podelski

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Freiburg, Germany
{kupfersc,mwehrle,nebel,podelski}@informatik.uni-freiburg.de

Abstract. It is probably very hard to develop a new model checker that is faster than UPPAAL for verifying (*correct*) timed automata. In fact, our tool MCTA does not even try to compete with UPPAAL in this (i. e., UPPAAL's) arena. Instead, MCTA is geared towards analyzing *incorrect* specifications of timed automata. It returns (shorter) error traces faster.

1 Our tool: MCTA

We present MCTA, a model checking tool for real-time specifications modeled as timed automata. Although the tool can be used for verification, MCTA is rather optimized for falsification, i. e., detecting violations against safety properties fast and returning short error traces. Several types of traces can be generated, including an option to find a (guaranteed) shortest error trace. There is also the possibility to examine MCTA's traces with UPPAAL's graphical user interface.

MCTA accepts input models in the form of the UPPAAL input language (cf. [1]). So far only a fraction thereof is supported, e. g. there is no support for urgent channels, arrays, etc. yet. Internally, MCTA uses UPPAAL's timed automata parser library. For the representation of zones, MCTA uses UPPAAL's difference bound matrices library. Both libraries are released under the terms of the LGPL or GPL, respectively, and are freely available at <http://www.uppaal.com/>. All other data structures and all algorithms (and their implementation) used are genuine to MCTA.

MCTA is free software and also released under the terms of the GPL. Pre-compiled Linux executables and a snapshot of the source code of our tool are also freely available at <http://www.informatik.uni-freiburg.de/~kupfersc/mcta/>.

2 The ingredients of MCTA

MCTA accelerates the detection of error states by using the well-known directed model checking approach [4, 5]. In this approach, an abstract distance value is computed for each state encountered during the state space traversal. The abstract distance values determine the *order* in which the states are explored. Among possible successor states, the ones with a lower value are preferred. There are many different strategies to explore

the state space. MCTA allows the user to choose between two strategies based on two wide-spread search methods called A* and *greedy search*. The first explores states s with lowest value of $c(s) + h(s)$ first, where $c(s)$ is the length of the path from the initial state through which s was reached. Under certain conditions on the abstract distance values, one is guaranteed a shortest error path. In the second strategy, states are explored by increasing value of $h(s)$. Doing so, the length of the detected error path is not guaranteed to be as short as possible, but tends to explore less states in practice.

MCTA generates the abstract distance values *fully automatically* for each input given by a timed automaton and a safety property. This is done by efficiently computing a rather coarse abstraction (the user can choose among several kinds of abstraction, see below) and taking the distance in the abstract state space. MCTA in addition offers the possibility to automatically recognize which transitions should be penalized during the state space traversal; this is a new technique presented in [8].

Monotonicity Abstraction Currently, MCTA comes with several kinds of abstractions as the basis for computing the abstract distance values. Here, we will only explain the distance function which is based on the *monotonicity abstraction* [7]. This abstraction is mainly an adaption of a technique from AI Planning namely *ignoring delete lists* [2]. The idea of the corresponding abstraction is to have *every state variable, once it has 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 this abstraction.

MCTA assigns to each state encountered during the state space traversal an abstract distance value by applying the monotonicity abstraction to the part of the state space that is rooted in the current state, and traversing the abstract state space. The length of the abstract error trace is the state’s abstract distance value. If there is no abstract error path, then there is no concrete one either.

3 Results

We compare the performance of MCTA and UPPAAL for detecting error traces in incorrect specifications of timed automata. For both tools we chose the most powerful options. We used the current version of UPPAAL (4.0.6) with the option *randomized depth first search* (rDF). The results for rDF in Table 1 are averaged over 10 runs. For MCTA the specific options are: the strategy for the state space traversal being based on greedy search, the abstraction for the abstract distance values being the monotonicity abstraction, and the recognition of “useless transitions” and the state space traversal penalizing recognized transitions [8].

The examples C_1, \dots, C_9 stem from an industrial case study called “Single-tracked Line Segment” [6]. It models a distributed real-time controller for a segment of tracks where trams share a piece of track. The examples M_1, \dots, M_4 and N_1, \dots, N_4 come from a case study namely “Mutual Exclusion” [3]. It models a real-time protocol to ensure mutual exclusion of states in a distributed system via asynchronous communication. Both case studies are part of the AVACS project benchmark suite.

The results in Table 1 (visualized in Fig. 1) clearly demonstrate that the algorithms employed by our tool are useful for analyzing incorrect timed automata. In comparison

Table 1. Experimental results of UPPAAL’s and MCTA’s most powerful options. The results are computed on an Intel Xeon with 2.66 Ghz. Dashes indicate out of memory (more than 4 GB).

Exp	explored states		runtime in s		memory in MB		trace length	
	UPPAAL	MCTA	UPPAAL	MCTA	UPPAAL	MCTA	UPPAAL	MCTA
M_1	8343	4256	0.37	0.07	38	56	829	97
M_2	27156	8186	1.53	0.10	40	57	3245	146
M_3	24368	10650	1.39	0.12	40	58	2991	91
M_4	70906	22412	4.93	0.24	45	64	11728	136
N_1	11115	5689	0.93	0.10	38	59	607	108
N_2	45998	15377	4.99	0.25	41	62	3788	152
N_3	31725	16332	3.31	0.26	41	65	3302	91
N_4	220262	44199	25.31	0.71	51	84	14003	118
C_1	15407	1658	0.23	0.12	38	56	945	91
C_2	31308	1333	0.32	0.16	39	56	820	91
C_3	45443	1153	0.36	0.13	39	56	541	91
C_4	366056	1001	2.90	0.19	49	57	1690	121
C_5	2629269	833	23.54	0.22	120	57	2345	114
C_6	21940802	833	230.08	0.29	761	57	3237	114
C_7	–	829	–	0.35	–	57	–	114
C_8	–	816	–	0.28	–	57	–	95
C_9	–	13423	–	3.24	–	71	–	90

with UPPAAL which does not employ such algorithms (based on automatically generated abstract error distances), our tool finds the error paths faster. It explores less states and uses less memory and thus scales to larger benchmarks. At the same time, it returns shorter error paths.

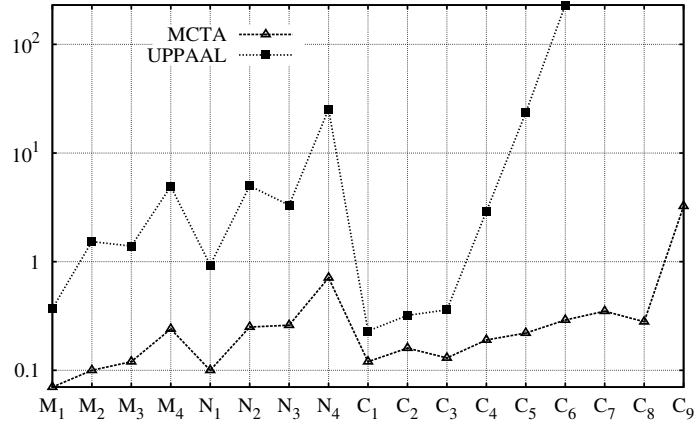


Fig. 1. Comparing UPPAAL and MCTA on incorrect specifications of timed automata (“MCTA’s arena”): runtime (in s) for detecting error traces. MCTA is orders of magnitude faster.

4 Outlook

In the future, MCTA will evolve by supporting more and more language constructs for defining (extensions of) timed automata, and by providing more and more kinds of abstractions for computing abstract distance values. Eventually, we hope, the results and the practical experience with MCTA for analyzing incorrect specifications will flow into tools that were originally geared towards analyzing *correct* timed automata.

Acknowledgements

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.

We also thank the UPPAAL team for making their DBM and their parser library freely available.

References

1. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *International School on Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer-Verlag, 2004.
2. Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.
3. Henning Dierks. Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.
4. Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2–3):247–267, 2004.
5. Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with HSF-Spin. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, 2001.
6. Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, and Alexander Baer. The Uni-ForM workbench, a universal development environment for formal methods. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods (FM 1999)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1186–1205. Springer-Verlag, 1999.
7. Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, 2006.
8. Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Useless transitions are useful. Reports of SFB/TR 14 AVACS 39, SFB/TR 14 AVACS, 2008.