Mcta: Heuristics and Search for Timed Systems

Martin Wehrle[1] and Sebastian Kupferschmid[2]

[1] University of Basel, Switzerland
martin.wehrle@unibas.ch
[2] ATRiCS Advanced Traffic Solutions GmbH, Freiburg, Germany
sebastian.kupferschmid@atrics.com

Abstract. Mcta is a directed model checking tool for concurrent systems of timed automata. This paper reviews Mcta and its new developments from an implementation point of view. The new developments include both heuristics and search techniques that define the state of the art in directed model checking. In particular, Mcta features the powerful class of pattern database heuristics for efficiently finding shortest possible error traces. Furthermore, Mcta offers new search techniques based on multi-queue search algorithms. Our evaluation demonstrates that Mcta is able to significantly outperform previous versions of Mcta as well as related state-of-the-art tools like Uppaal and Uppaal/Dmc.

1 Introduction

Model checking of real-time systems is an interesting and important research issue in theory and in practice. In this context, Uppaal [2, 3] is a state-of-the-art model checker for real-time systems that are modeled as timed automata [1]. Uppaal offers several approaches to successfully tackle the state explosion problem. However, to efficiently find short error traces in large concurrent systems of timed automata, additional search techniques are desired.

Mcta [19] is a tool for model checking large systems of concurrent timed automata. Mcta is optimized for *falsification*, i. e., for the efficient detection of short error traces in faulty systems. Therefore, Mcta applies the *directed model checking* approach [9]. Directed model checking is a version of model checking that applies a *distance heuristic* and a special *search algorithm* to guide the search towards error states. Distance heuristics compute a numeric value for every state s encountered during the search, reflecting an estimation of the length of a shortest trace from s to an error state. These values are used by the underlying search algorithm (e. g., the well-known A* algorithm [11, 12]) to guide the search. Overall, most of the proposed distance heuristics can be computed automatically based the description of the input system. Therefore, directed model checking is a fully automatic approach as well. For the special setting when *admissible* distance heuristics are applied (i. e., distance heuristics that are guaranteed to never overestimate the real error distance), directed model checking allows for *optimal* search, i. e., in this case, directed model checking computes *shortest possible* error traces with the A* search algorithm. This is

desirable because shorter error traces allow one to better understand the reason for the bug.

In this paper, we review Mcta and its new developments from an implementation point of view. In particular, we provide an overview of Mcta's lightweight and flexible architecture. This architecture is tailored to engineering an efficient model checker based on heuristic search methods. The current version of Mcta (Mcta-2012.05 or Mcta-2012 for short) supports both optimal and suboptimal search methods. In the setting of optimal search, Mcta-2012 features a powerful admissible *pattern database heuristic*. To get a feeling of the power of Mcta-2012's heuristic search methods in an optimal search setting, we provide a snapshot of Mcta-2012's performance in Table 1. The problems D_1–D_6 stem from an industrial real-time case study (see Sec. 6 for details). A dash indicates that the corresponding tool exceeded the memory limit of 4 GByte. We observe that Mcta-2012 shows superior performance.

Table 1. Snapshot of Mcta-2012's performance in an optimal search setting. The table provides the best runtime in seconds for Mcta-2012, for Mcta-0.1 (corresponding to the predecessor of Mcta-2012 that has been released in 2008 [19]), for Uppaal/Dmc, and for Uppaal-4.0.13.

Instance	Mcta-2012	Mcta-0.1	Uppaal/Dmc	Uppaal-4.0.13
D_1	10.2	81.2	84.7	90.5
D_2	12.2	433.4	255.3	539.0
D_3	12.3	487.0	255.6	548.4
D_4	13.9	288.0	256.7	476.4
D_5	60.1	–	–	–
D_6	66.4	–	–	–

Furthermore, in the setting of suboptimal search, Mcta now features search algorithms that extend classical directed model checking by applying a *multi-queue* approach using *several* open queues instead of only one. This approach can be applied with arbitrary distance heuristics.

Mcta is written in C++ and Python. It is released under the GPL and can be obtained from the website `http://mcta.informatik.uni-freiburg.de/`. The website particularly provides a binary of Mcta, the source code, relevant benchmark problems, and related papers. Subsequently, when we want to distinguish between the new and the earlier version of Mcta, the new version is called Mcta-2012, whereas the earlier version that corresponds to the last tool paper [19] is called Mcta-0.1 (as also indicated on the website).

The remainder of the paper is organized as follows. In Sec. 2, we give the preliminaries that are needed for this work. In Sec. 3, we present Mcta's basic architecture, based on which the newly developed components and their implementation are described in detail in Sec. 4 and Sec. 5. Furthermore, an experimental evaluation is given in Sec. 6. Finally, we conclude the paper in Sec. 7 and give an overview of next development steps.

2 Preliminaries

In this section, we introduce the preliminaries that are needed for this paper. In Sec. 2.1, we give a brief introduction to the timed automata formalism. In Sec. 2.2, we describe the classical directed model checking approach that MCTA is based on.

2.1 Timed Automata

We consider a class of timed automata that is extended with bounded integer variables. A timed automaton $\mathcal{A}$ consists of a finite set of *locations* and a set of *edges* that connect (some of) $\mathcal{A}$'s locations. Every location features a *clock invariant* represented as a conjunction of clock constraints $x \prec n$ for a clock x and an integer $n \in \mathbb{N}$, where $\prec \in \{<, \leq\}$. Furthermore, edges are annotated with *guards*, *effects* and a *synchronization label* from a global synchronization alphabet Σ. The *guard* of an edge consists of a clock and an integer guard, consisting of clock and integer constraints, respectively. The *effect* of an edge consists of a list of clocks (to be reset) and a list of integer assignments. A *(parallel) system* of timed automata is defined as a set $\mathcal{M} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ of timed automata.

The operational semantics of a system $\mathcal{M}$ of timed automata is defined as follows. As the explicit size of $\mathcal{M}$'s state space is infinite, we use a symbolic representation of the state space that is sound and complete. This representation is based on *zones*. In this setting, a *global state* consists of a discrete part and a symbolic part. It is defined as a tuple $s = \langle L, V, Z \rangle$, where L is a function that evaluates for every automaton in $\mathcal{M}$ the current location in s and V is a function that evaluates for every integer variable the current value in s. L and V define the discrete part of s. Furthermore, Z is the *zone* of s, i. e., a conjunction of clock constraints that describes the possible values of the clock variables in s. Z defines the symbolic part of s. We define a *transition* in $\mathcal{M}$ either as a set of one edge that has a special internal void label (asynchronous communication), or as a set of two edges from different automata with the same synchronization label from Σ. Guards and effects of transitions are defined accordingly. A transition t is *applicable* in a state s if the location, integer and clock guards of t are satisfied in s. In this case, the *successor state* $t[s]$ of s is defined as the state s where the locations and the integer values are first changed according to the effect of t, and the zone of $t[s]$ is defined as an update of the zone of s according to the clock guard and the clock resets of t. Finally, the resulting zone of $t[s]$ is maximized while preserving consistency with the location invariants of the destination locations of t. The resulting state space of $\mathcal{M}$ is called the *zone graph* of $\mathcal{M}$.

2.2 Directed Model Checking

In general, depending on the distance heuristic and the search algorithm, directed model checking influences the order in which the state space is traversed. For a

system of timed automata $\mathcal{M}$, directed model checking is performed on the zone graph of $\mathcal{M}$. The basic model checking algorithm of MCTA is shown in Fig. 1.

```
1  function dmc(M, h, φ):
2      open = empty priority queue
3      closed = ∅
4      open.insert(s_0, priority(h, s_0))
5      while open ≠ ∅ do:
6          s = open.getMinimum()
7          if s ⊨ φ then:
8              generateErrorTrace(s)
9          closed = closed ∪ {s}
10         for each transition t of M that is applicable in s do:
11             if t[s] ∉ closed then:
12                 open.insert(t[s], priority(h, t[s]))
13     return True
```

Fig. 1. MCTA's basic directed model checking algorithm

For a given system $\mathcal{M}$, a distance heuristic h, and an error property φ (i. e., a negated invariant property), MCTA performs a reachability algorithm on the zone graph of $\mathcal{M}$. Therefore, MCTA maintains a priority queue *open* that contains encountered states for which the successor states have not yet been computed, and a *closed* list that contains the *explored* states, i. e., the states for which the successor states have already been computed. Starting with the initial state s_0, MCTA iteratively computes successor states and evaluates them with a *priority* function. For all encountered states, the priority function computes a *priority value* which is determined by the distance heuristic h and the applied search algorithm. According to the priority value, MCTA iteratively removes a best state s from *open* and checks if s is an error state (line 6–8). If this is the case, an error trace in generated by back-tracing from s (therefore, MCTA additionally stores information in the states about how they have been reached). If s is not an error state, s is stored in *closed*, and the successors of s are computed, evaluated and inserted into *open* if they are not already explored.

At this point, it is important to note that the algorithm in Fig. 1 should be read on a conceptual, rather than on an implementation level. For example, on an implementation level, the closed list is a special kind of hash table (rather than a set) that supports a certain inclusion test for states. We will come back to these points in Sec. 3, specifically for a discussion on MCTA's data structures, distance heuristics and search algorithms.

3 Mcta's Architecture and Features

In this section, we give an overview of MCTA's overall architecture and MCTA's features. Therefore, in Sec. 3.1, we present a high-level overview of the modules

MCTA consists of. In Sec. 3.2 and Sec. 3.3, we specifically describe MCTA's distance heuristics and search algorithms.

3.1 Mcta's Basic Architecture

MCTA consists of the modules *parser*, *system*, *search*, and *heuristics*. The input of MCTA consists of a file that contains a description of the timed automata system, and a file that contains the property to check. The property to check is a CTL formula of the form $\exists\Diamond\varphi$, where φ is a conjunction of constraints that speaks about automata and variables (i. e., φ describes the error states). Currently, MCTA supports a part of UPPAAL's input language.

The *parser* module of MCTA uses UPPAAL's timed automata parser library (UTAP), which is released under the LGPL and freely available at `http://www.uppaal.org/`. After parsing the input, MCTA generates an internal representation of the input system and the property. The corresponding algorithms and data structures to build this representation are part of the *system* module. The system representation is used by the *search* module which performs a search on the zone graph of this representation using a distance heuristic and a search algorithm (according to the algorithm given in Fig. 1). MCTA offers several kinds of distance heuristics and search algorithms (see Sec. 3.2 and Sec. 3.3 for an overview). In our setting, a distance heuristic is a function $h : \mathcal{S} \to \mathbb{N} \cup \{\infty\}$ that returns for each state of $\mathcal{S}$ an estimation of its error distance. The distance heuristics are implemented in the *heuristics* module. The overall architecture of MCTA is depicted in Fig. 2.

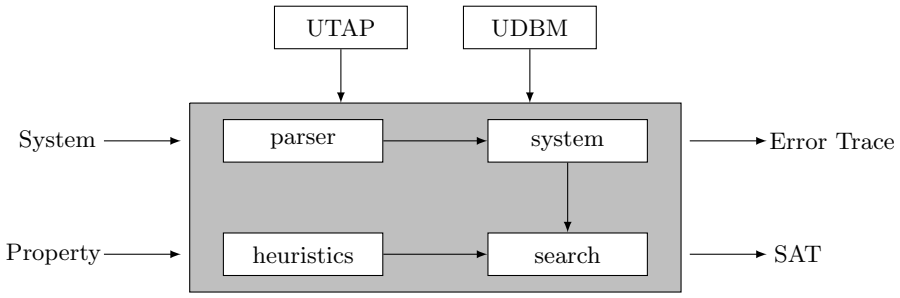

Fig. 2. MCTA's basic architecture

The *search* module is central to MCTA. It consists of the *search engine*, which implements the global while loop of Fig. 1, and uses dedicated data structures for *states*, for the *open* queue and for the *closed* list. For the internal representation of zones, MCTA uses UPPAAL's difference bound matrices library (UDBM), which is released under the GPL and freely available at `http://www.uppaal.org/`. The *open* queue and the *closed* list are special kinds of hash tables. Overall, the design of the search engine is lightweight, which is supposed to simplify

the implementation of new search algorithms. Furthermore, the interface to the heuristics module is intended to simplify the implementation of new distance heuristics.

3.2 The Heuristics Module

The heuristics module of MCTA-2012 features several distance heuristics to guide the search. To estimate the error distance of a state s, the distance heuristics compute an abstract error trace $\pi^{\#}$ that starts in an abstraction of s, and use the length of $\pi^{\#}$ as the estimation for the length of a concrete error trace from s. We give a short description of the different approaches in the following.

1. The d^U and d^L distance heuristics [8] are based on the local graph distances of the automata of the input system. Synchronization, integer variables and clock variables are ignored.
2. The h^L and h^U distance heuristics [17] are based on the *monotonicity abstraction*, which abstracts the original semantics of the system. The monotonicity abstraction assumes that variables are set-valued and, once they obtain a value, keep this value forever. The sets that contain the collected values grow monotonically over transition application, hence the name of the abstraction. The h^L and h^U distance heuristics compute abstract error traces based on this abstraction.
3. A pattern database heuristic based on downward pattern refinement [18]. We do not go into detail here but refer the reader to Sec. 4.

Compared to the earlier version MCTA-0.1, the h^U heuristic and the pattern database heuristic based on downward pattern refinement are new developments.

3.3 The Search Module

In this section, we focus on a description of MCTA's search algorithms that are the essential part of the *search* module. The search algorithms make use of the estimated error distances provided by the distance heuristic.

1. The standard greedy search algorithm [22] and A* search algorithm [11, 12], including the uninformed search algorithms depth-first and breadth-first search.
2. The notion of *useless transitions* provides an approach to evaluate transitions (rather than just evaluating states) [25, 26]. Transitions t are called *useless* in a state s if no shortest error trace starts in s with t. This criterion is approximated such that it can be computed efficiently. For this approach, the current version of MCTA maintains two open queues q_0 and q_1, where q_1 maintains states that are reached by a useless transition, and q_0 maintains the remaining states. The q_1 queue is accessed only if q_0 is empty.

3. Iterative-context bounding [20] is an approach that stems from the area of software model checking. In our setting, it corresponds to an iterative deepening search algorithm that prefers states that are reached with low number of *context switches*, i. e., with a low number of transition applications of different automata.
4. Context-enhanced directed model checking [24] is a further technique to additionally prioritize transitions during directed model checking. Similar to the iterative context bounding algorithm, it gives preference to states that are reached by a transition that interferes with previously applied transitions. In contrast, context switches are defined and exploited in a different way.

In comparison to MCTA-0.1, the implementation of the iterative context-bounding approach and the implementation of context-enhanced directed model checking are new developments. Both of these algorithms are based on multiple open queues. We will describe their implementation in Sec. 5.

4 Mcta's Pattern Database Heuristics

In this section, we describe MCTA's implementation for *pattern database (PDB) heuristics* in general, and the implementation of an extended version of downward pattern refinement in particular. We assume the reader is roughly familiar with pattern databases, and only give a short introduction. Pattern database heuristics are a class of admissible distance heuristics that come from the area of Artificial Intelligence [4, 7]. For an input system $\mathcal{M}$ and a subset $\mathcal{P}$ of the system components of $\mathcal{M}$ (the so-called *pattern*), a pattern database PDB is a data structure that contains the abstract states of $\mathcal{M}|_{\mathcal{P}}$, where $\mathcal{M}|_{\mathcal{P}}$ denotes the projection abstraction of $\mathcal{M}$ that is obtained by abstracting away all systems components that are not contained in $\mathcal{P}$. Furthermore, for all abstract states in the PDB, the corresponding abstract error distance is stored. The PDB is computed once *prior* to directed model checking. *During* directed model checking, the PDB is used as a distance heuristic $h^{\mathcal{P}}$ by mapping every encountered concrete state s to a corresponding abstract state $s^{\#}$. The distance value $h^{\mathcal{P}}(s)$ of s is defined as the corresponding abstract error distance of $s^{\#}$.

4.1 Mcta's Architecture for Pattern Databases

Assuming that a pattern $\mathcal{P}$ is given, we present MCTA's framework for the computation of pattern databases (see Sec. 4.2 how MCTA computes a suitable pattern). MCTA performs three steps to compute the pattern database for $\mathcal{P}$: Abstracting the system, then computing the entire abstract state space $S^{\#}$, and finally, computing the abstract error distances for the abstract states in $S^{\#}$.

Abstracting the System First, for the given input system $\mathcal{M}$ and the pattern $\mathcal{P}$, MCTA computes a *projection abstraction* of $\mathcal{M}$ based on $\mathcal{P}$ by abstracting away all system components that do not occur in $\mathcal{P}$. Therefore, MCTA applies

the *abstractor* tool which also comes with the current MCTA-2012 release. The abstractor tool works as follows. For integer and clock variables v to be abstracted, the abstractor removes v from the guards and effects of the transitions of $\mathcal{M}$. If there is an edge with an effect such that the new value of v depends on a variable in $\mathcal{P}$, then v is abstracted away, too. Moreover, for an automaton $\mathcal{A}$ to be abstracted, the abstractor replaces $\mathcal{A}$ with a new automaton $\mathcal{A}'$ that only consists of one location. Moreover, $\mathcal{A}'$ contains loop edges for all edges of $\mathcal{A}$ where the guard is abstracted, but the effects and synchronization labels are kept. Doing so, we obtain an overapproximation $\mathcal{M}|_{\mathcal{P}}$ of the original system $\mathcal{M}$.

Computing the Abstract State Space Second, for the obtained abstraction $\mathcal{M}|_{\mathcal{P}}$ of the original system $\mathcal{M}$, the entire reachable state space of $\mathcal{M}|_{\mathcal{P}}$ is computed in a forward manner and dumped into a file. For the traversal of the abstract state space, an extended version of the original search engine is used. This extended version specifically takes into account that if a state s that is already in the closed list is encountered again (i. e., on a different trace), the (new) transition that led to s is stored additionally. The abstract states and transitions are stored in a serialized form. Moreover, abstract error states are stored with a special error flag. Overall, we end up with a file that contains all the abstract states (where abstract error states are specifically indicated) together with all the abstract transitions.

Computing the Abstract Error Distances Finally, based on the file that contains the abstract state space, we apply the external tool PDBGEN to generate the final pattern database. PDBGEN comes with the current MCTA-2012 release and computes the abstract error distances for a given abstract state space. This is done in a backwards manner via a version of Dijkstra's algorithm. More precisely, PDBGEN starts by assigning the error distance zero to all the abstract error states, and by assigning infinity to all the other states. In the following, PDBGEN iteratively checks the predecessor states and updates the distance value if it is reached more cheaply than before. The output is a file that contains the serialized abstract states together with their abstract error distances. This is the pattern database which can be fed into MCTA to be used as a distance heuristic. We finally remark that, doing this 3-step process to compute the PDB, we avoid the expensive regression operation on the zone graph.

4.2 Running Mcta with Extended Downward Pattern Refinement

To compute the pattern, MCTA uses the external tool MCTA-PDB that is implemented in Python. MCTA-PDB acts as a wrapper around the pattern generator PDBGEN and MCTA. More precisely, MCTA-PDB first generates the underlying pattern. The pattern is generated with an algorithm based on downward pattern refinement [18]. In addition to the originally proposed h^{dpr} distance heuristic, MCTA-2012 applies explicit search in intermediate abstractions to deal with clock

variables more explicitly. Doing so leads to a more-fine grained approach to select clocks than proposed in the original paper (where *all* clock variables have been selected for the pattern by default). For the resulting pattern, the above described 3-step process is performed. Finally, MCTA-PDB calls MCTA with the resulting pattern database. To apply MCTA with the pattern database heuristic based on downward pattern refinement, select the following command line parameters.

```
mcta-pdb --dprc --astar SYSTEM PROPERTY
```

5 Multi-Queue Search Algorithms

In this section, we describe MCTA's implementation for *multi-queue search algorithms*. After giving a brief conceptual description in Sec. 5.1, we present MCTA's general framework for this approach in Sec. 5.2. In the subsequent sections, we specifically describe the implementation of two multi-queue search algorithms from the literature, namely iterative context bounding [20] in Sec. 5.3, and context-enhanced directed model checking [24] in Sec. 5.4.

5.1 The General Approach

In the setting where not only a distance heuristic, but also an additional quality measure to guide the search is available, there is the question of how to exploit this additional information. In such cases, a popular approach is to maintain multiple open queues instead of only one. Within this approach, states are pushed into different open queues according to the additional quality measure, and ordered in this queue according to the original distance heuristic. The "best" state to explore next is then defined as the "best" state according to the distance heuristic in the "best" open queue according to the additional quality measure. For example, multi-queue approaches have been successfully applied in the area of AI planning for *combining* distance heuristics [23] (i. e., in this case, the additional quality measure is another distance heuristic), or for additionally evaluating *transitions* rather than only evaluating states [13, 14, 25]. Furthermore, in the area of model checking, similar approaches have been proposed to evaluate transitions based on iterative context bounding [20], interference contexts [24], and the notion of *useless* transitions [26].

For the rest of this section, we assume a setting where a distance heuristic (to evaluate states) *and* a technique to evaluate transitions is available. The idea is to exploit this additional information by preferably exploring states that are estimated to be near to an error state (which corresponds to low distance values as before) *and* that are reached by a transition that is estimated to guide the search properly towards an error state. More precisely, the evaluation of transitions determines the open queue in which the resulting successor state is maintained, and (as in the classical approach) the distance heuristic determines the ordering of the states in the queues. Formally, the priority function from the

algorithm in Fig. 1 becomes a function with domain $\mathbb{N} \times \mathbb{N}$, i. e., it does no longer only assign a natural number to states s, but additionally a natural number for the transition that led to s to determine in which open queue s is maintained.

5.2 Mcta's Architecture for Multi-Queue Search Algorithms

The high-level architecture of MCTA to maintain several open queues is best described by the following template functions. They show how an extended open queue that internally consists of multiple open queues is accessed to get and insert states. The algorithmic design is rather straight forward and depicted in Fig. 3.

```
1 function insert(s, h):
2     k = evaluate predecessor transition t of s
3     q_k.insert(s, h(s))

1 function getMinimum():
2     determine open queue q_k to access
3     return q_k.getMinimum()
```

Fig. 3. Multi-queue accessing functions

In the above algorithm, we assume that an upper bound on the number of queues can be computed (see Sec. 5.3 and Sec. 5.4 how this is done for the individual approaches). In contrast to the classical approach in Fig. 1, the *insert* function computes a natural number k to determine the quality of the transition that led to the state s that is inserted. This number in turn determines the index of the queue in which s is inserted. Furthermore, *getMinimum()* returns the best state of the open queue that is accessed next; note that it depends on the applied search algorithm *how* this queue is actually determined.

5.3 Implementation of Iterative Context Bounding

Iterative context bounding (ICB) has been proposed for the purpose of testing multithreaded programs [20]. Roughly speaking, ICB performs an iterative deepening search with the objective to minimize the number of *context switches*, i. e., the number of execution points on a trace where the scheduler forces the active thread to change.

MCTA implements this approach by considering threads as automata. Therefore, a context switch occurs if two consecutive transitions on a trace belong to two different automata. ICB can be combined with arbitrary distance heuristics as well as uninformed search (where the latter corresponds to the original approach). MCTA applies a special search engine that maintains two open queues q_0 and q_1 (i. e., $k \in \{0, 1\}$ in Fig. 3). The *insert* function is implemented as follows. For a state s and an applicable transition t in s, the successor state is

inserted into q_0 if the edge(s) of t and the edge(s) of the predecessor transition of s belong to the same automata. The *getMinimum()* function is implemented by always accessing q_0 until q_0 gets empty. If this is the case, then q_0 and q_1 are exchanged, reflecting that the number of context switches has been increased by one. In case q_1 is empty, too, we report that no error state is reachable.

Running Mcta with Iterative Context Bounding To run MCTA with iterative context bounding, use the `--icb=1` flag when MCTA is called. We remark that MCTA supports additional options to define a context, but we do not go into detail here. A short description of these parameters is given when MCTA is called with the `--help` option.

5.4 Implementation of Context-Enhanced Directed Model Checking

Context-enhanced directed model checking is a further multi-queue search approach [24]. In contrast to iterative context-bounding, contexts are essentially defined based on *interference* of transitions, where transitions t and t' *interfere* if t writes a variable that is read by t', or t' writes a variable that is read by t, or t and t' write a common variable. Moreover, during the search, more than two open queues are maintained in general. The approach is based on preferably exploring states that are reached by transitions that interfere with previously applied transitions. More precisely, states are preferably explored if they are reached with a transition with low *interference distance* to the previously applied transition, where the interference distance of t and t' is defined as the smallest $k \in \mathbb{N}$, $k \geq 1$, such that there are transitions $t_1, \ldots, t_k$ with the property that t interferes with t_1, t_1 interferes with t_2, ..., and t_k interferes with t'.

In MCTA, context-enhanced directed model checking is implemented as follows. First, for every transition t in the system, the interference distance of t is computed to every other transition in the system. This is a all pairs-shortest-path problem for which we apply the *Floyd Warshall* algorithm [10]. The resulting interference distances are stored in a 2-dimensional vector. The maximal interference distance N for two transitions defines the number of open queues (obviously, N is defined because systems have a finite number of transitions). More precisely, we maintain a global open queue Q that consists of a vector of open queues $q_0, \ldots, q_N$. The *insert* function in Fig. 3 is defined as follows. For a state s' that is supposed to be inserted into Q, MCTA determines the interference distance k of the predecessor transition t' of s' and the predecessor transition of the predecessor state of s', where s' is inserted into queue q_k. In the special case that the effect of t' satisfies a constraint of the property that is subject to model checking, s' is inserted in q_0. The *getMinimum()* function determines the smallest non-empty queue in Q to get the next state.

Running Mcta with Iterative Context Bounding To run MCTA with the context-enhanced directed model checking approach, use the `--ce` flag when MCTA is called. We remark that MCTA supports additional options for this

setting, but we do not go into detail here. A short description of these parameters is given when MCTA is called with the `--help` option.

6 Mcta's Performance

In this section, we present an experimental evaluation of MCTA-2012 on large and challenging real-time benchmarks. Specifically, some of these benchmarks stem from industrial real-time case studies. To evaluate the performance of MCTA-2012, a bug has been inserted in all of them.

The case study "Single-Tracked Line Segment" [15] (the problem instances $C_1, \ldots, C_9$ and $D_1, \ldots, D_9$) models a distributed real-time controller for a segments of tracks where trams share a piece of track. The distributed controller is supposed to ensure that never two trams that drive in opposite directions are simultaneously given permission to enter the shared piece of track. The controller was modeled in terms of PLC automata [6], which is an automata-like notation for real-time programs. With the tool MOBY/RT [21], the PLC automata system has been transformed into abstractions of its semantics in terms of timed automata. For the evaluation of MCTA-2012, we chose the property that never both directions are given permission to enter the shared segment simultaneously.

As a further set of benchmarks, we used a case study called "Mutual Exclusion" (problem instances $M_1, \ldots, M_4$ and $N_1, \ldots, N_4$). As suggested by the name, in this case study, mutual exclusion has to be established for real-time systems. It is based on a protocol that is described by Dierks [5]. We refer the reader to the website of MCTA for a more detailed description. All of these benchmarks can also be obtained from the MCTA website.

The experiments have been performed on an AMD Opteron Processor 6174 with 2.2 GHz system and 4 GByte of memory. We compare MCTA-2012 in an optimal search setting with the best technique described in the last tool paper [19] (corresponding to MCTA-0.1). We also provide results for the tools UPPAAL-4.0.13 and UPPAAL/DMC [16]. UPPAAL provides an efficient implementation of breadth-first search, whereas the other tools apply the directed model checking approach. Note that in this paper, we do not compare to UPPAAL's randomized depth first search (rdfs) because rdfs is not guaranteed to find *shortest* possible error traces (see the earlier tool paper of MCTA [19] for a comparison of suboptimal search techniques including UPPAAL's rdfs). We used the options that lead to the best experimental results for each tool. In particular, for MCTA-2012, we used extended downward pattern refinement as described in Sec. 4.2. The results are given in Table 2. For MCTA-12 and UPPAAL/DMC, the best search options to find shortest error traces are PDB approaches; therefore, for these tools, the pure search time in the concrete state space is reported additionally.

The results clearly indicate that MCTA-2012 mostly outperforms the other directed model checking tools on these problems. Moreover, we observe that these problems are large and complex because even UPPAAL, which provides a very efficient implementation of breadth-first search, cannot solve all of them. Furthermore, we observe that the preprocessing of MCTA-2012 often takes most

Table 2. Results with the A* search algorithm. Abbreviations: "MCTA-12": MCTA-2012, "MCTA-08": MCTA-0.1, "U/DMC": UPPAAL/DMC, "runtime": overall runtime including any preprocessing in seconds, "explored states": number of explored concrete states, dashes indicate out of memory (> 4 GByte). For MCTA-12 and U/DMC that rely on PDBs, the pure search time in the concrete (i. e., time without preprocessing) is reported in parenthesis.

	runtime in seconds				explored states				trace
Inst.	MCTA-12	MCTA-08	U/DMC	UPPAAL	MCTA-12	MCTA-08	U/DMC	UPPAAL	length
M_1	2.2 (0.3)	0.6	3.0 (0.2)	0.5	29029	41455	190	14290	47
M_2	2.9 (0.9)	2.6	3.2 (0.2)	2.1	99528	164856	4417	51485	50
M_3	3.7 (1.7)	3.0	3.4 (0.4)	2.2	165336	189820	11006	52987	50
M_4	8.2 (6.2)	13.5	4.0 (1.0)	8.8	549999	724030	41359	186435	53
N_1	2.6 (0.1)	2.7	18.0 (0.4)	3.8	3606	93951	345	28196	49
N_2	3.1 (0.6)	14.7	12.1 (0.5)	17.1	26791	438394	3811	100078	52
N_3	4.2 (1.7)	19.1	14.7 (4.5)	17.5	70439	547174	59062	102124	52
N_4	13.0 (10.4)	95.3	34.3 (27.8)	76.4	388076	2317206	341928	370459	55
C_1	1.3 (0.1)	0.2	0.8 (0.1)	0.2	98	12458	130	21008	54
C_2	1.4 (0.1)	0.7	1.1 (0.7)	0.5	98	32751	89813	55544	54
C_3	1.4 (0.1)	0.8	0.8 (0.0)	0.6	98	37126	197	74791	54
C_4	1.4 (0.1)	7.5	0.9 (0.1)	6.0	312	301818	1140	553265	55
C_5	1.5 (0.1)	60.9	1.0 (0.1)	53.1	1178	2174789	7530	3977279	56
C_6	1.5 (0.1)	605.6	1.1 (0.3)	514.3	2619	20551913	39436	33526538	56
C_7	1.6 (0.1)	–	1.7 (0.8)	–	4247	–	149993	–	56
C_8	1.6 (0.2)	–	1.7 (0.9)	–	5416	–	158361	–	56
C_9	1.7 (0.2)	–	1.7 (0.8)	–	13675	–	127895	–	57
D_1	10.2 (0.3)	81.2	84.7 (65.0)	90.5	2789	1443874	4610240	4048866	78
D_2	12.2 (0.4)	433.4	255.3 (5.4)	539.0	5086	6931937	4223	21478364	79
D_3	12.3 (0.4)	487.0	255.6 (5.4)	548.4	5161	7900038	2993	21553760	79
D_4	13.9 (0.3)	288.0	256.7 (5.4)	476.4	1023	4660652	2031	18487819	79
D_5	60.1 (6.4)	–	–	–	122204	–	–	–	102
D_6	66.4 (10.5)	–	–	–	426571	–	–	–	103
D_7	67.1 (7.9)	–	–	–	180132	–	–	–	104
D_8	68.3 (6.2)	–	–	–	28285	–	–	–	104
D_9	71.4 (6.3)	–	–	–	12186	–	–	–	105

of the overall model checking time. However, the preprocessing time mostly pays off, specifically compared to the uninformed search provided by UPPAAL, but also compared to the other directed model checking tools. For the M instances, the pure search time of MCTA-2012 is still comparable to the search time of most of the other tools. Moreover, in these instances, we observe that the overall number of explored states as well as the number of explored states per second is lower for UPPAAL than for MCTA-2012. Although we do not know the exact reason, we suppose that this is the case because UPPAAL uses a more efficient representation of the zone graph. We finally remark that we have also successfully *verified* correct systems with MCTA-2012. This is possible because admissible heuristics h can be used as a pruning method: If $h(s) = \infty$ for a state s, then the real error distance of s is infinity as well, and hence, s can safely be pruned (recall that admissible heuristics never overestimate the real error distance). For more details, including experimental results, we refer the reader to the literature [18].

7 Conclusion

In this paper, we have reviewed MCTA and its new developments from an implementation point of view. The new developments include heuristics and search techniques for both optimal and suboptimal search. We have observed that MCTA-2012 is very useful in efficiently finding shortest possible error traces in faulty systems. For the future, we specifically aim at developing new admissible distance heuristics. A main issue for research will be to effectively find the sweet spot of the trade-off to be as accurate as possible on the one hand, and as cheap to compute as possible on the other hand.

Acknowledgments Parts of this work have been done while the authors have been employed by the University of Freiburg, Germany. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `http://www.avacs.org/`).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
2. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Uppaal implementation secrets. In: Proc. FTRTFT. pp. 3–22. Springer (2002)
3. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Proc. SFM-RT. pp. 200–236. Springer (2004)
4. Culberson, J.C., Schaeffer, J.: Pattern databases. Comp.Int. 14(3), 318–334 (1998)
5. Dierks, H.: Comparing model-checking and logical reasoning for real-time systems. Formal Aspects of Computing 16(2), 104–120 (2004)
6. Dierks, H.: Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics. Habilitation thesis, University of Oldenburg, Germany (2005)
7. Edelkamp, S.: Planning with pattern databases. In: Proc. ECP. pp. 13–24 (2001)
8. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5(2), 247–267 (2004)
9. Edelkamp, S., Schuppan, V., Bosnacki, D., Wijs, A., Fehnker, A., Aljazzar, H.: Survey on directed model checking. In: Proc. MOCHART. pp. 65–89. Springer (2008)
10. Floyd, R.W.: Algorithm 97: Shortest path. Communications of the ACM 5(6), 345 (1962)
11. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Systems Science and Cybernetics 4(2), 100–107 (1968)
12. Hart, P.E., Nilsson, N.J., Raphael, B.: Correction to a formal basis for the heuristic determination of minimum cost paths. SIGART Newsletter 37, 28–29 (1972)
13. Helmert, M.: The Fast Downward planning system. JAIR 26, 191–246 (2006)
14. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. JAIR 14, 253–302 (2001)

15. Krieg-Brückner, B., Peleska, J., Olderog, E.R., Baer, A.: The UniForM workbench, a universal development environment for formal methods. In: Proc. MF. pp. 1186–1205. Springer (1999)
16. Kupferschmid, S., Dräger, K., Hoffmann, J., Finkbeiner, B., Dierks, H., Podelski, A., Behrmann, G.: Uppaal/DMC – abstraction-based heuristics for directed model checking. In: Proc. TACAS. pp. 679–682. Springer (2007)
17. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In: Proc. SPIN. pp. 35–52. Springer (2006)
18. Kupferschmid, S., Wehrle, M.: Abstractions and pattern databases: The quest for succinctness and accuracy. In: Proc. TACAS. pp. 276–290. Springer (2011)
19. Kupferschmid, S., Wehrle, M., Nebel, B., Podelski, A.: Faster than Uppaal? In: Proc. CAV. pp. 552–555. Springer (2008)
20. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proc. PLDI. pp. 446–455. ACM Press (2007)
21. Olderog, E.R., Dierks, H.: Moby/RT: A tool for specification and verification of real-time systems. J. UCS 9(2), 88–105 (2003)
22. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley (1984)
23. Röger, G., Helmert, M.: The more, the merrier: Combining heuristic estimators for satisficing planning. In: Proc. ICAPS. pp. 246–249. AAAI Press (2010)
24. Wehrle, M., Kupferschmid, S.: Context-enhanced directed model checking. In: Proc. SPIN. pp. 88–105. Springer (2010)
25. Wehrle, M., Kupferschmid, S., Podelski, A.: Useless actions are useful. In: Proc. ICAPS. pp. 388–395. AAAI Press (2008)
26. Wehrle, M., Kupferschmid, S., Podelski, A.: Transition-based directed model checking. In: Proc. TACAS. pp. 186–200. Springer (2009)