

Context-Enhanced Directed Model Checking

Martin Wehrle and Sebastian Kupferschmid

University of Freiburg
Department of Computer Science
Freiburg, Germany
{mwehrle, kupfersc}@informatik.uni-freiburg.de

Abstract. Directed model checking is a well-established technique to efficiently tackle the state explosion problem when the aim is to find error states in concurrent systems. Although directed model checking has proved to be very successful in the past, additional search techniques provide much potential to efficiently handle larger and larger systems. In this work, we propose a novel technique for traversing the state space based on *interference contexts*. The basic idea is to preferably explore transitions that interfere with previously applied transitions, whereas other transitions are deferred accordingly. Our approach is orthogonal to the model checking process and can be applied to a wide range of search methods. We have implemented our method and empirically evaluated its potential on a range of non-trivial case studies. Compared to standard model checking techniques, we are able to detect subtle bugs with shorter error traces, consuming less memory and time.

1 Introduction

When model checking safety properties of large systems, the ultimate goal is to prove the system correct. However, for practically relevant systems this is often not possible because of the state explosion problem. Complementary to verify a system correct, finding reachable *error states* is a potentially easier task in practice. An error state can be found by only exploring a small fraction of the entire reachable state space. Especially for this purpose, directed model checking has found much attention in recent years [1–9]. Directed model checking is tailored to the fast detection of reachable error states. This is achieved by focusing the state space traversal on those parts of the state space that show promise to contain reachable error states. A heuristic function is used to assign each state that is encountered during the traversal of the state space a heuristic value. Typically, a heuristic function approximates a state’s distance to a nearest error state. These values are used to determine which state to explore next. An advantage of directed model checking is that the heuristic functions are usually abstraction based and computed fully automatically based on the declarative description of the system. Usually, distance heuristics are computed by solving a simplified problem, and then using the length of the abstract error trace as an estimation for the actual error distance in the concrete. They differ in the way of how the given problem is simplified. Overall, the performance of directed model checking methods has proved to be often much better than the performance of uninformed search methods like breadth-first or depth-first search. However, for large systems, even error detection is very challenging.

To cope with larger and larger systems, additional techniques to tackle the state explosion problem are needed. Among these, approaches that additionally evaluate the relevance of *transitions*, rather than just states, are very promising. Such methods have proved to further alleviate the state explosion problem as the additional information improves the search guidance. As a consequence, the number of states that have to be explored before an error state is encountered can be significantly reduced in practice. Techniques following this approach have first been proposed in the areas of AI planning and directed model checking. For instance, *helpful actions* [10], *preferred operators* [11], and *useless transitions* [12, 13] are powerful instantiations of this paradigm. All these techniques have in common that they label applicable transitions with a Boolean flag. This flag indicates whether a transition is relevant or not. States that are reached by a relevant transition are preferred during the traversal of the state space. Another technique that exploits certain properties of transitions is *iterative context-bounding* [14]. This algorithm was proposed in software model checking for error detection in multithreaded programs. The algorithm searches for error traces that exhibit a minimal number of *context switches*, i. e., execution points where the scheduler forces the active thread to change. Transitions that do not induce a context switch are preferred. We will detail these approaches in the section on related work.

In this paper, we introduce context-enhanced directed model checking. Roughly speaking, the main idea of this approach is the following: If there is a transition t that is part of a shortest error trace π , then there often is a subsequent transition in π that *profits* from t . Therefore, we propose to preferably explore states that have been reached by a transition that profits from the effect of the previously applied transition. As a consequence, the search process avoids to “jump” while traversing the state space, i. e., it prefers transitions that belong to the same part of the system. We use the notion of *interference context* to determine how much a transition profits from the execution of another transition. With the above mentioned approaches, our technique shares the property that it labels transitions and defers the expansion of states reached by less relevant transitions. In contrast to these approaches, our method does not assign a Boolean flag to a transition, but an integer value. Our approach is embedded in a multi-queue search algorithm that is well-suited to respect the different levels of relevance. Another distinguishing property is that our approach can also be successfully applied to uninformed search, which is not possible with the other approaches, except for iterative context bounding. We have implemented our approach and applied it to uninformed search as well as directed model checking with several distance heuristics from the literature. We also compare our technique with the useless transitions approach as well as iterative context bounding as outlined above. The experiments reveal that our approach scales much better than the previous approaches in many challenging problems coming from real-world case studies.

The remainder of this paper is organized as follows. Section 2 introduces the preliminaries for this work. In the subsequent section, we introduce context-enhanced directed model checking. Afterwards, in Sec. 4, we discuss related work. In the following section, we empirically evaluate our approach on a number of benchmarks and compare it with plain directed model checking as well as previously proposed techniques for prioritizing transitions. Section 6 concludes the paper and discusses future work.

2 Preliminaries

In this section, we give the preliminaries needed for this work. In Sec. 2.1, we introduce our notation and computational model. This is followed by an introduction to directed model checking in Sec. 2.2.

2.1 Notation

Our approach is applicable to a broad class of transition systems, including parallel systems with interleaving, synchronization and linear arithmetic. We only require that the transitions resemble *guarded commands*, i. e., a transition consists of a *precondition* and an *effect*. Therefore, we define our computational model in a general way, consisting of a finite set of bounded integer variables V and a finite set of *transitions* T . For the sake of presentation, we restrict the form of transitions as stated in the next definition.

Definition 1 (System). A system \mathcal{S} is a tuple $\langle V, T \rangle$, where V is a finite set of bounded integer variables and T is a finite set of transitions. A transition t is a tuple $\langle pre, eff \rangle$, where

- pre is the precondition of t . It is a conjunction over constraints of the form $v \bowtie c$, where $v \in V$, $c \in \mathbb{Z}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.
- eff is the effect of t . It is a set of assignments of the form $v := c$, where $v \in V$ and $c \in \mathbb{Z}$.

For a transition $t = \langle pre, eff \rangle$, we denote its precondition and effect with $pre(t)$ and $eff(t)$, respectively. A *system state* is a function that assigns each variable $v \in V$ a value of v 's domain. A transition t is *applicable* in a state s if t 's precondition is satisfied by s , i. e., if $s \models pre(t)$. The successor state $s' = t(s)$ reached by applying t in s is obtained by updating all variables according to the effect of t . Formally, the state space of a system is defined as follows.

Definition 2 (State space of a system). Let $\mathcal{S} = \langle V, T \rangle$ be a system. The state space of \mathcal{S} is defined as a labeled transition system $\mathcal{T}(\mathcal{S}) = (S, \Delta)$, where S is a set of states and $\Delta \subseteq S \times T \times S$ is a transition relation. There is a transition $(s, t, s') \in \Delta$ iff $s \models pre(t)$ and $s' = t(s)$.

For transitions $(s, t, s') \in \Delta$, we will also write $s \xrightarrow{t} s'$. We define a model checking task as a system together with an initial state and a target formula describing the set of error states.

Definition 3 (Model checking task). Let \mathcal{S} be a system and $\mathcal{T}(\mathcal{S}) = (S, \Delta)$ be its state space. A model checking task is a tuple $\langle \mathcal{S}, s_0, \varphi \rangle$, consisting of a system \mathcal{S} , an initial state $s_0 \in S$ and a target formula φ . Target formulas have the same form as preconditions of transitions. The task is to find a sequence $\pi = t_1, \dots, t_n$ of transitions, with $s_{i-1} \xrightarrow{t_i} s_i \in \Delta$ for $1 \leq i \leq n$ and $s_n \models \varphi$.

Informally speaking, the target formula describes a set of states with an undesirable property, the so called *error states*. Therefore, we call a sequence π as defined as in the last definition an *error trace*.

2.2 Directed Model Checking

In a nutshell, directed model checking is the application of heuristic search [15] to model checking. The main idea of directed model checking is to explore those parts of the state space first that show promise to contain reachable error states. As a consequence, it is possible to detect error states in systems whose entire state space is too huge for brute force methods. In directed model checking, the state space traversal is guided (“directed”) towards error states based on specific criteria. Ideally, these guidance criteria are *automatically* extracted from the declarative description of the system under consideration by taking an abstraction thereof. Based on such an abstraction, a heuristic function h is computed that typically approximates a state’s distance to a nearest error state. During the search process, h is used to assign each encountered state s a heuristic value $h(s)$. These values are used to influence the *order* in which state to explore, hereby completeness is *not* affected. Figure 1 shows a basic directed model checking algorithm.

```

1 function dmc( $\mathcal{S}, s_0, \varphi, h$ ):
2    $open = \text{empty priority queue}$ 
3    $closed = \emptyset$ 
4    $priority = \text{evaluate}(s_0, h)$ 
5    $open.insert(s_0, priority)$ 
6   while  $open \neq \emptyset$  do:
7      $s = open.getMinimum()$ 
8     if  $s \models \varphi$  then:
9       return False
10     $closed = closed \cup \{s\}$ 
11    for each transition  $t$  applicable in  $s$  do:
12       $s' = t(s)$ 
13      if  $s' \notin closed \cup open$  then:
14         $priority = \text{evaluate}(s', h)$ 
15         $open.insert(s', priority)$ 
16  return True

```

Fig. 1. A basic directed model checking algorithm

The algorithm takes a model checking task $\langle \mathcal{S}, s_0, \varphi \rangle$ and a heuristic function h as input. It returns *False* if there is a reachable error state, i. e., a state that satisfies φ , otherwise it returns *True*. The state s_0 is the initial state of \mathcal{S} . The algorithm maintains a priority queue $open$ which contains visited but not yet explored states. When $open.getMinimum$ is called, $open$ returns a minimum element, i. e., one of its elements with minimal priority value. States that have been expanded are stored in $closed$. Every state encountered during search is first checked if it satisfies φ . If this is not the case, its successors are computed. Every successor that has not been visited before is inserted into $open$ according to its priority value. The $evaluate$ function depends on the applied version of directed model checking, i. e., if applied with A* or greedy search (cf. [15, 16]). For A*, $evaluate(s, h)$ returns $h(s) + c(s)$, where $c(s)$ is the length of the

path on which s was reached for the first time. For greedy search, it simply evaluates to $h(s)$. When every successor has been computed and prioritized, the process continues with the next state from *open* with lowest priority value.

To be able to report found error traces, every state stores a pointer to its immediate predecessor state and transition. We finally remark that, depending on the implementation details of the used priority queue, depth-first search and breadth-first search are instances of the algorithm from Fig. 1 when h is the constant zero function.

3 Context-Enhanced Directed Model Checking

In this section, we introduce context-enhanced directed model checking which is based on *interference contexts*. To compactly describe the main idea of our approach, we need the notion of *innocence*. A transition t is innocent if for each state s where t is applicable, there is no constraint of the target formula φ that is satisfied by $\text{eff}(t)$. This means, if there is a conjunct c of φ so that $t(s) \models c$, then c was already satisfied by s . Note that, if the initial state of a system is not an error state, then only applying innocent transitions will never lead to error states.

The main idea of context-enhanced directed model checking is the following. Let s be a state and let t be an innocent transition enabled at s . If t is the first transition of a shortest error trace starting from s , then there must be applicable transitions at $s' = t(s)$ that *profit* from the effect of t . Otherwise t cannot be part of a shortest error trace. Hence, our algorithm focuses on transitions that belong to the same part of the system. In this work, we use *interference contexts* to determine whether a transition profits from a preceding transition.

3.1 Interference Contexts

Our notion of interference contexts is based on the well-known concept of interference. Similar to other work (for example, in the area of partial order reduction [17]), we use a definition that can be statically checked. Roughly speaking, two transitions interfere if they work on a common set of system variables. For a transition t , we will use the notation $\text{var}(\text{pre}(t))$ and $\text{var}(\text{eff}(t))$ to denote the set of variables occurring in the precondition and the effect of t , respectively.

Definition 4 (Interference). *Two transitions t_1 and t_2 interfere, iff at least one of the following conditions holds:*

1. $\text{var}(\text{eff}(t_1)) \cap \text{var}(\text{pre}(t_2)) \neq \emptyset$,
2. $\text{var}(\text{eff}(t_2)) \cap \text{var}(\text{pre}(t_1)) \neq \emptyset$,
3. $\text{var}(\text{eff}(t_1)) \cap \text{var}(\text{eff}(t_2)) \neq \emptyset$.

Informally, this means that two transitions interfere if one writes a variable that the other is reading, or both transitions write to a common variable. We next give a pruning criterion based on interference. To formulate this, we first define the notion of *interference contexts*. Roughly speaking, for a transition t and $n \in \mathbb{N}_0$, the interference context $C_n(t)$ contains all transitions for which there is a chain of at most n transitions where neighboring transitions interfere.

Definition 5 (Interference Context). Let $\langle V, T \rangle$ be a system, $t \in T$ be a transition and $n \in \mathbb{N}_0$. The interference context $C_n(t)$ is inductively defined as follows:

$$\begin{aligned} C_0(t) &= \{t\} \\ C_n(t) &= C_{n-1}(t) \cup \{t' \in T \mid \exists t'' \in C_{n-1}(t) : t'' \text{ interferes with } t'\} \end{aligned}$$

As we are dealing with finite transition systems, there exists a smallest $N \leq |T|$, so that $C_N(t) = C_{N+1}(t)$ for all transitions $t \in T$. We will denote this context with $\overline{C}(t)$. Note that this context induces an equivalence relation on the set of system transitions T . It partitions T into subsets of transitions which operate on pairwise disjoint variables. Based on this notion, we now give a pruning criterion which is guaranteed to preserve completeness. Informally speaking, if a state is part of a shortest error trace and has been reached via an innocent transition t , then it suffices to only apply transitions from $\overline{C}(t)$ at s .

Proposition 1. Let $\langle \mathcal{S}, s_0, \varphi \rangle$ be a model checking task for a system $\mathcal{S} = \langle V, T \rangle$ and a target formula $\varphi = \bigwedge (v_i \bowtie c_i)$, where $\bowtie \in \{<, \leq, =, \geq, >\}$, $v_i \in V$ and $c_i \in \mathbb{Z}$. Let s be a state in \mathcal{S} that is part of a shortest error trace from s_0 and has been reached by a transition sequence with last transition t . Further assume that t is innocent. Then there is a shortest error trace from s that starts with a transition from $\overline{C}(t)$.

Proof. As s is part of a shortest path from the initial state, and t is innocent, there is a shortest error trace π that starts in s which contains a transition t'_π that needs the effects of t , i. e., $\text{var}(\text{eff}(t)) \cap \text{var}(\text{pre}(t'_\pi)) \neq \emptyset$. Otherwise, t would not have been needed and s would not be part of a shortest error trace. We transform π into a shortest error trace π' so that the first transition in π' is contained in $\overline{C}(t)$. Let t_1, \dots, t_n be the prefix of π so that t_n is the first transitions in π that interferes with t , i. e., t_1, \dots, t_{n-1} do not interfere with t . Then π' is constructed by an inductive argument. If $n = 1$, i. e., t_1 interferes with t , then $\pi' = \pi$ as $t_1 \in \overline{C}(t)$. For the induction step, let t_{n+1} be the first interfering transition with t . If t_{n+1} and t_n do not interfere, then they can be exchanged in π' as non-interfering transitions can be applied in any order, leading to the same state. If they do interfere, then by definition $t_n \in \overline{C}(t)$, which proves the claim.

From Prop. 1, we can derive the following pruning criterion: If there is a shortest error trace from s that starts with a transition from $\overline{C}(t)$, then all transitions that are not contained in $\overline{C}(t)$ can be pruned, without losing completeness. Note that the condition for s to be on a shortest error trace can be assumed without loss of generality: If s is *not* part of a shortest error trace, *every* successor can be pruned.

In practice, this pruning criterion does not seem to fire very often: Benefits are only obtained if the system's transitions can be partitioned into at least two sets of transitions that operate on disjoint variables. In such cases, also compositional model checking is applicable. These concepts are known and by no means new. In fact, the pruning criterion can be seen as a version of compositional model checking, where completely independent parts of the system are handled individually. However, the formulation of Prop. 1 lends itself to a way of how to approximate the pruning criterion. This leads to the heuristic approach to prioritize transitions that we are introducing next.

3.2 The Context-Enhanced Search Algorithm

In this section, we describe how we approximate the pruning criterion. The basic idea is actually quite simple: Instead of considering the exact closure $\overline{C}(t)$ of a transition t , we substitute it with the interference context $C_n(t)$ for some bound $n < N$, where N is the smallest number such that $C_N(t) = \overline{C}(t)$ for all transitions $t \in T$. Suppose for a moment that we know how to choose a good value for n . Obviously, Prop. 1 does not hold anymore if we replace $\overline{C}(t)$ with $C_n(t)$. As a consequence thereof, states that are reached via transitions that are not contained in $C_n(t)$ cannot be pruned without loosing completeness. The approximation therefore becomes a heuristic criterion for the *relevance* of transitions. Instead of pruning these states, we suggest to defer their expansion. This can be done by extending the search algorithm from Fig. 1 with a second open queue. States whose expansions we want to defer are inserted in this queue. All other states are inserted in the standard open queue. States from the second queue are only expanded if the standard open queue is empty. The question remains which value we should use for the parameter n . From Def. 5, we know that $C_i(t) \subseteq C_j(t)$ for all transitions t iff $i \leq j$. On the one hand, the higher the parameter, the better $\overline{C}(t)$ is approximated. However, as we already argued (and also describe in the experimental section), the exact pruning criterion does not fire very often in practice. On the other hand, the lower the parameter the more missclassifications may occur, which hampers the overall performance of the model checking process.

Instead of choosing a constant parameter, we propose to use a multi-queue search algorithm that maintains $N + 1$ different open queues, where $N \in \mathbb{N}$ is defined as above. Overall, we obtain a family of open queues q_0, \dots, q_N , where q_i is accessed (according to the given distance heuristic) iff q_0, \dots, q_{i-1} are empty, and q_i is not. The basic directed model checking algorithm from Fig. 1 can be converted into our multi-queue search method by replacing the standard open queue with a multi-queue and the corresponding accessing functions as given in Fig. 2. In these queues, states are maintained as follows.

```

1 function insert( $s'$ , priority):
2   if  $t$  is innocent then:
3     if  $t' \notin \overline{C}(t)$  then:
4       prune  $s'$ 
5     else:
6       determine smallest  $n \in \mathbb{N}$  such that  $t' \in C_n(t)$ 
7        $q_n$ .insert( $s'$ , priority)
8   else:
9      $q_0$ .insert( $s'$ , priority)

1 function getMinimum():
2   determine smallest  $n$  such that  $q_n \neq \emptyset$ 
3   return  $q_n$ .getMinimum()

```

Fig. 2. Multi-queue accessing functions for context-enhanced directed model checking

Let s be a state that was reached with transition t , and let $s' = t'(s)$ be the successor state of s under the application of the transition t' . If t is innocent and $t' \notin \overline{C}(t)$, then we can safely prune s' . If t is innocent and $t' \in \overline{C}(t)$, then the successor state s' is maintained in queue q_1 if $t' \in C_1(t)$, and maintained in q_i if $t' \in C_i(t)$ and $t' \notin C_{i-1}(t)$. If t is not innocent, then s' is stored in q_0 .

According to the given distance heuristic, *getMinimum* returns a state with best priority from the queue q_i with minimal index i that is not empty. The multi-queue is empty iff all queues are empty. Obviously, our approach remains complete, as only the order in which the states are explored is influenced. The advantage of this multi-queue approach is that we do not have to find a good value for n , which strongly depends on the system. By always expanding states from the lowest non-empty queue, the algorithm also respects the quality of the estimated relevance.

4 Related Work

Directed model checking has recently found much attention, and various distance heuristics to estimate a state's distance to a nearest error state have been proposed in this context [1–9]. Given a declarative description of the system under consideration, these distance heuristics are usually computed fully automatically based on abstractions. Overall, directed model checking has demonstrated to significantly outperform uninformed search methods like breadth-first or depth-first search.

To efficiently handle larger and larger systems, additional search enhancements have recently been proposed for directed model checking as well as for AI planning. In particular, techniques to additionally prioritize transitions (rather than only states) are very promising. In the area of AI planning, *helpful actions* [10] and *preferred operators* [11] have been proposed. Both approaches heuristically select transitions that should be preferred during search. However, these concepts are specifically designed for certain distance heuristics.

In the context of directed model checking, a similar approach called *useless transitions* has been proposed [13]. A transition is considered as useless in a state s if it does not start a shortest error trace from s . This criterion is approximated to identify such transitions, which are less preferred during the search. In this approach, the distance heuristic itself is used to estimate whether a transition is useless or not. Hence, the quality of this approach strongly depends on the informedness of the distance heuristic. Furthermore, combining this approach with uninformed search methods is not possible as no distance heuristic is applied there. Complementary to this, context-enhanced directed model checking is independent of the distance heuristic. As we shall see in the experimental section, our approach is successfully applicable to uninformed search.

Musuvathi and Qadeer work in the area of software model checking. They propose a technique for bug detection based on *context bounding* [14]. For multithreaded programs, they propose an algorithm that limits the number of *context switches*, i. e., the number of execution points where the scheduler forces the active thread to change. Their algorithm is actually a kind of iterative deepening search, where the number of context switches that may occur in each trace is increased in each iteration. They define a context essentially as a thread, whereas in our work, a context switch would correspond to

a state where a transition is executed which does not interfere with the preceding transition. In our setting, a context switch corresponds to exploring a state from a deferred queue. However, our criterion is stricter than the context switch criterion proposed by Musuvathi and Qadeer: Exploring a state from a deferred queue corresponds to a context switch in the sense of Musuvathi and Qadeer, but not vice versa. Moreover, we handle the different levels of interference with a fine-grained multi-queue search algorithm. Musuvathi and Qadeer propose an algorithm that is guaranteed to minimize the number of context switches. Contrarily, our search algorithm does not necessarily minimize them, but performs better in systems with tight interaction of the processes.

Finally, partial order reduction techniques can also be considered as a technique for prioritizing transitions to overcome the state explosion problem [17–19]. Partial order reduction exploits the fact that independent transitions need not be considered in every ordering. It reduces the branching factor of the system by computing a subset of the applicable transitions that suffices to preserve completeness. Partial order reduction is orthogonal to our approach of interference contexts, and it will be interesting to investigate the combination of these techniques in the future.

5 Evaluation

We have implemented our algorithm based on interference contexts and empirically evaluate its potential on a range of problems, including academic benchmarks as well as large and practical relevant systems from industrial case studies. We evaluate our algorithm on a number of different search methods, including uninformed search as well as various heuristic search methods as proposed in the literature and implemented in our model checker MCTA [20].

5.1 Benchmark Set

All our benchmarks, including real-world problems from industrial case studies, stem from the AVACS¹ benchmark suite. Our benchmark systems consist of parallel automata with bounded integer variables, interleavings and binary synchronization. Some of them also feature clock variables and actually represent timed automata [21]. Currently, clock variables are ignored by our implementation. Note that all these formalisms are instantiations of our system definition.

The M and N examples come from a case study called “Mutual Exclusion”. It models a real-time protocol to ensure mutual exclusion of a state in a distributed system via asynchronous communication. The protocol is described in full detail by Dierks [22]. The S examples (“Single-tracked Line Segment”) stem from a case study from an industrial project partner of the UniForM-project [23] where the problem is to design a distributed real-time controller for a segment of tracks where trams share a piece of track. For the evaluation of our approach, we chose the property that both directions are never given simultaneous permission to enter the shared segment. In both case studies, a subtle error has been inserted by manipulating a delay so that the asynchronous

¹ <http://www.avacs.org/>

communication between these automata is faulty. The F_i examples are versions of the Fischer protocol for mutual exclusion (cf. [24]). The index i gives the number of parallel automata. An error state is reached if two predefined automata are simultaneously in a certain location. We made error states reachable by weakening one of the temporal conditions in the automata. As a final set of benchmarks, the H examples model the well-known Towers of Hanoi for a varying number of disks. The index of the examples gives the number of involved disks. Initially, all n disks are on the first peg; the goal is to move them all to the second peg, moving only one disk at a time and such that never a larger disk is on top of a smaller one.

5.2 Experimental Setting

We implemented our context-enhanced search algorithm (denoted with CE in the following) into our model checker MCTA [20]. All experiments have been performed on an AMD Opteron 2.3 GHz system with 4 GByte of memory. We set a timeout to 30 minutes. We apply CE to uninformed search as well as to directed model checking with various distance heuristics.

We compare to the (rather coarse) distance heuristics d^L and d^U [6] as well as to the (more informed) distance heuristics h^L and h^U [2]. All of them are implemented in MCTA. The d^L and d^U heuristics are based on the *graph distance* of automata; synchronization behavior and integer variables are ignored completely. The h^L and h^U heuristics are based on the *monotonicity abstraction*. Under this abstraction, variables can have multiple values simultaneously. The h^L heuristic performs a fixpoint iteration under this abstraction starting in the current state until an error state is reached, and returns the number of iterations as heuristic value. Based on this fixpoint iteration, h^U additionally extracts an abstract error trace starting from the abstract error state, and returns the number of abstract transitions as the estimate. Furthermore, we compare CE to various related search algorithms, including iterative context bounding (ICB) and the useless transitions approach (UT). As threads correspond to processes in our setting, we have implemented ICB with process contexts as proposed by Musuvathi and Qadeer [14], denoted with ICB_P . This means that a context switch occurs if two consecutive transitions belong to different processes. Moreover, we implemented ICB with our definition of interference contexts, denoted with ICB_I . Here, a context switch occurs if a transition t' does not belong to $C_n(t)$, where t is the preceding transition. After some limited experiments, we set the bound n to 2 because we achieved the best results in terms of explored states for this value. For larger values, no context switches occurred, and hence ICB_I behaves like greedy search. Finally, we also compare CE with a multi-queue version of the useless transitions approach [12]. In the tables, we denote this approach with UT .

5.3 Experimental Results

We give detailed results for a coarse distance heuristic based on the graph distance (d^U), for a more informed heuristic based on the monotonicity abstraction (h^L), as well as for breadth-first search as uninformed search method. Moreover, we additionally provide

average performance results for depth-first search as well as for the distance heuristics d^L and h^U .

Table 1. Experimental results for greedy search with the d^U heuristic. Abbreviations: plain: greedy search, CE : greedy search + interference contexts, ICB_P : iterative context bound algorithm with process contexts, ICB_I : iterative context bound algorithm with interference contexts, UT : useless transitions. Dashes indicate out of memory (> 4 GByte) or out of time (> 30 min). Uniquely best results are given in bold fonts.

Exp.	explored states					runtime in s					trace length				
	plain	CE	ICB_P	ICB_I	UT	plain	CE	ICB_P	ICB_I	UT	plain	CE	ICB_P	ICB_I	UT
F_5	9	21	112	8	9	0.0	0.0	0.0	0.0	0.0	6	6	6	6	6
F_{10}	9	36	447	8	9	0.0	0.0	0.0	0.0	0.0	6	6	6	6	6
F_{15}	9	51	1007	8	9	0.0	0.0	0.0	0.0	0.0	6	6	6	6	6
S_1	11449	10854	44208	14587	9796	0.0	0.1	0.2	0.2	0.1	823	192	59	1022	842
S_2	33859	31986	163020	93047	31807	0.1	0.3	0.8	0.5	0.5	1229	223	59	134	1105
S_3	51526	43846	199234	80730	52107	0.2	0.3	1.1	0.5	0.8	1032	184	59	1584	1144
S_4	465542	402048	3.0e+6	1.8e+6	504749	2.0	2.1	18.0	8.6	7.5	3132	1508	60	783	5364
S_5	4.6e+6	2.8e+6	3.8e+7	2.4e+7	4.6e+6	22.6	18.8	256.9	136.8	70.6	14034	886	65	1014	14000
S_6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
M_1	185416	5360	100300	56712	7557	156.4	0.1	2.5	1.1	0.1	106224	246	65	94	923
M_2	56240	15593	608115	82318	294877	1.1	0.2	22.5	1.7	67.1	13952	520	63	3378	51541
M_3	869159	15519	623655	281213	26308	1729.5	0.2	23.8	7.2	0.5	337857	562	82	3863	1280
M_4	726691	21970	3.4e+6	–	100073	428.0	0.4	221.3	–	1.8	290937	486	80	–	4436
N_1	10215	5688	92416	68582	19698	0.3	0.1	5.6	3.3	0.7	2669	162	79	109	1855
N_2	–	11939	645665	161800	96307	–	0.3	71.7	8.2	5.1	–	182	75	277	8986
N_3	–	14496	616306	417181	29254	–	0.4	62.4	25.8	1.2	–	623	84	1586	784
N_4	330753	25476	3.7e+6	889438	239877	23.0	0.8	760.8	91.6	17.8	51642	949	89	361	1969
H_3	222	279	1000	479	244	0.0	0.0	0.0	0.0	0.1	44	32	52	82	60
H_4	2276	2133	11726	7631	545	0.0	0.1	0.0	0.1	0.5	184	132	116	190	92
H_5	20858	5056	142359	116200	15435	0.1	0.2	0.3	0.4	13.9	708	238	266	422	400
H_6	184707	122473	1.5e+6	1.1e+6	130213	0.6	0.8	4.1	3.2	128.8	2734	1670	522	918	1242
H_7	1.6e+6	876847	1.5e+7	1.4e+7	1.2e+6	5.7	5.4	52.0	42.0	1267.7	11202	4456	1136	2100	3922
H_8	1.4e+7	2.3e+6	–	–	–	53.8	17.1	–	–	–	45280	5666	–	–	–
H_9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Let us start to discuss the results for CE with the d^U heuristic from Table 1. First, we observe that for the hard problems, the number of explored states with CE is significantly better than with plain directed model checking as well as the other, related approaches. In the smaller instances (e. g., the F examples and the small S examples), the performance is comparable. Compared to plain directed model checking, we could also solve more problems when CE is applied. This mostly also pays off in better search time. Finally, the length of the found error trace is comparable in the F and the small H instances, but mostly much shorter than with plain directed model checking and the UT approach. Overall, when using d^U , we could significantly improve directed model checking with CE , and also outperformed related approaches in terms of scalability and error trace length.

The results for CE and the h^L heuristic are given in Table 2. First, we observe that UT performs much better with h^L than with d^U and often leads to the best configura-

Table 2. Experimental results for greedy search with the h^L heuristic. Abbreviations as in Table 1.

Exp.	explored states					runtime in s					trace length				
	plain	CE	ICB _P	ICB _I	UT	plain	CE	ICB _P	ICB _I	UT	plain	CE	ICB _P	ICB _I	UT
F_5	179	21	112	216	7	0.0	0.0	0.0	0.0	0.0	12	6	6	12	6
F_{10}	86378	36	447	95972	7	3.3	0.0	0.0	3.8	0.0	22	6	6	22	6
F_{15}	–	51	1007	–	7	–	0.0	0.0	–	0.0	–	6	6	–	6
S_1	1704	4903	24590	1971	1537	0.0	0.2	0.4	0.1	0.1	84	68	62	77	76
S_2	3526	11594	81562	4646	1229	0.1	0.4	1.2	0.2	0.1	172	70	62	166	76
S_3	4182	5508	89879	6118	1061	0.1	0.3	1.4	0.2	0.1	162	73	62	109	76
S_4	29167	7728	1.2e+6	108394	879	0.7	0.5	14.1	1.2	0.2	378	198	60	653	77
S_5	215525	2715	1.4e+7	870603	1116	5.2	0.4	142.7	7.4	0.3	1424	85	65	2815	78
S_6	1.7e+6	9812	–	1.3e+7	1116	37.4	1.1	–	85.5	0.4	5041	130	–	8778	78
S_7	1.6e+7	15464	–	–	1114	332.5	2.0	–	–	0.6	15085	130	–	–	78
S_8	7.1e+6	2695	–	–	595	129.3	0.5	–	–	0.3	5435	81	–	–	76
S_9	9.6e+6	4.0e+6	–	–	2771	201.1	192.9	–	–	1.3	5187	1818	–	–	94
M_1	4581	1098	102739	3230	4256	0.1	0.0	3.1	0.1	0.2	457	89	66	433	97
M_2	15832	1904	605001	53206	7497	0.3	0.0	27.8	1.2	0.5	1124	123	61	113	104
M_3	7655	8257	622426	141247	10733	0.1	0.2	26.1	2.8	0.7	748	180	86	100	91
M_4	71033	18282	3.3e+6	525160	16287	1.6	0.7	239.8	14.5	1.7	3381	334	81	118	98
N_1	50869	1512	93750	74307	5689	39.0	0.0	7.2	59.9	0.3	26053	93	79	26029	108
N_2	30476	2604	634003	82158	22763	1.2	0.1	77.3	3.8	1.5	1679	127	75	97	259
N_3	11576	10009	607137	177899	35468	0.4	0.4	77.1	9.6	2.7	799	224	86	106	204
N_4	100336	20248	3.7e+6	971927	142946	5.3	1.1	755.6	103.2	14.9	2455	396	85	134	792
H_3	127	256	1017	164	190	0.0	0.0	0.0	0.0	0.0	48	48	48	86	62
H_4	2302	764	11830	7488	620	0.0	0.1	0.0	0.1	0.0	300	94	124	438	114
H_5	20186	15999	144668	121027	31553	0.2	0.4	0.5	0.6	1.2	1458	478	252	878	890
H_6	230878	85947	1.5e+6	1.5e+6	281014	2.2	2.0	5.8	5.8	13.4	7284	1350	558	2070	2766
H_7	2.0e+6	622425	1.5e+7	1.6e+7	2.7e+6	21.4	17.9	69.0	67.1	155.3	18500	14314	1086	5164	7176
H_8	1.8e+7	2.1e+6	–	–	–	206.8	78.3	–	–	–	70334	74594	–	–	–
H_9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

tion. As already outlined, UT uses the distance heuristic itself to estimate the quality of a transition. Hence, the performance of UT strongly depends on the quality of the applied distance heuristic. This also shows up in our experiments: UT performs best in the F and S examples. On the other hand, CE performs best in M , N and H , and only marginally worse than UT in most of the F and S problems. Moreover, CE scales significantly better than the iterative context bounding algorithm for both context definitions. Overall, we observe that CE performs similarly to directed model checking with h^L and useless transitions (also in terms of error trace length), and scales (often significantly) better than the other approaches.

We also applied our context enhanced search algorithm to uninformed search. The results for breadth-first are given in Table 3. First, we want to stress that breadth-first search is *optimal* in the sense that it returns shortest possible counterexamples. This is not the case for CE . However, the results in Table 3 show that the length of the found error traces by CE is mostly only marginally longer than the optimal one (around a factor of 2 in the worst case in S_1 and S_2 , but mostly much better). Contrarily, the scaling behavior of CE is much better than that of breadth-first search. This allows us to solve two more problems on the one hand, and also allows us to solve almost all of the harder problems much faster. In particular, this shows up in the M and N examples, that

Table 3. Experimental results for breadth-first search. Abbreviations as in Table 1.

Exp.	explored states				runtime in s				trace length			
	plain	<i>CE</i>	<i>ICB_P</i>	<i>ICB_I</i>	plain	<i>CE</i>	<i>ICB_P</i>	<i>ICB_I</i>	plain	<i>CE</i>	<i>ICB_P</i>	<i>ICB_I</i>
<i>F</i> ₅	333	29	129	273	0.0	0.0	0.0	0.0	6	6	6	6
<i>F</i> ₁₀	5313	44	484	3868	0.0	0.0	0.0	0.0	6	6	6	6
<i>F</i> ₁₅	34068	59	1064	20538	0.4	0.0	0.0	0.3	6	6	6	6
<i>S</i> ₁	41517	19167	50361	58379	0.1	0.1	0.2	0.3	54	109	59	54
<i>S</i> ₂	118075	52433	181941	217712	0.4	0.3	0.8	0.9	54	109	59	54
<i>S</i> ₃	149478	33297	230173	264951	0.5	0.2	1.1	1.1	54	71	59	54
<i>S</i> ₄	1.3e+6	146094	3.5e+6	3.7e+6	5.4	0.7	18.7	16.2	55	67	60	55
<i>S</i> ₅	1.1e+7	569003	4.6e+7	4.4e+7	49.8	3.2	258.5	227.3	56	73	56	56
<i>S</i> ₆	–	2.5e+6	–	–	–	16.2	–	–	–	74	–	–
<i>S</i> ₇	–	–	–	–	–	–	–	–	–	–	–	–
<i>M</i> ₁	192773	32593	273074	116682	4.6	0.4	7.4	1.5	47	51	50	50
<i>M</i> ₂	680288	75159	1.7e+6	1.6e+6	19.1	1.1	95.2	56.6	50	52	53	50
<i>M</i> ₃	740278	94992	1.7e+6	1.7e+6	22.0	1.6	80.9	76.9	50	54	63	50
<i>M</i> ₄	2.6e+6	189888	9.7e+6	8.0e+6	87.6	3.5	893.7	506.7	53	55	66	53
<i>N</i> ₁	361564	34787	310157	157610	20.0	0.7	15.6	4.3	49	56	58	55
<i>N</i> ₂	2.2e+6	81859	2.4e+6	3.9e+6	399.0	1.8	357.6	622.2	52	57	72	52
<i>N</i> ₃	2.4e+6	92729	2.3e+6	4.2e+6	442.8	2.5	275.8	677.9	52	54	71	52
<i>N</i> ₄	–	201226	–	–	–	5.9	–	–	–	57	–	–
<i>H</i> ₃	448	315	1062	954	0.0	0.0	0.0	0.0	22	22	30	24
<i>H</i> ₄	4040	2689	11702	7228	0.0	0.1	0.0	0.1	52	54	64	54
<i>H</i> ₅	42340	19158	146955	121807	0.1	0.2	0.3	0.4	114	116	148	148
<i>H</i> ₆	377394	161747	1.5e+6	1.2e+6	0.7	0.8	3.0	2.8	240	300	294	278
<i>H</i> ₇	3.4e+6	1.2e+6	1.5e+7	1.3e+7	7.2	5.3	36.8	33.7	494	520	634	636
<i>H</i> ₈	3.0e+7	9.2e+6	–	–	67.2	44.0	–	–	1004	1302	–	–
<i>H</i> ₉	–	–	–	–	–	–	–	–	–	–	–	–

could be solved within seconds with *CE* (compared to sometimes hundreds of seconds otherwise). Finally, we remark that *UT* is not applicable with breadth-first search, as no distance heuristic is applied. Overall, we conclude that if *short*, but not necessarily *shortest* error traces are desired, breadth-first search should be applied with *CE* because of the better scaling behavior of *CE* on the one hand, and the still reasonable short error traces on the other hand.

To get a deeper insight into the performance of our context enhanced search algorithm in practice, we report average results for our approach applied to various previously proposed heuristics in Table 4. For each heuristic as well as breadth-first and depth-first search, we average the data on the instances that could be solved by all configurations. Furthermore, we give the total number of solved instances.

First, we observe that also on average, *CE* compares favorably to plain directed model checking. The average number of explored states could (sometimes by an order of magnitude) be reduced, which mostly also pays off in overall runtime. Furthermore, except for breadth-first search which returns shortest possible counterexamples, the length of the error traces could be reduced. Compared to the related approaches *ICB_P*, *ICB_I* and *UT*, we still observe that the average number of explored states is mostly lower with *CE* (except for the median with *UT* for h^L and h^U). In almost all cases, the average runtime is still much shorter. Apart from breadth-first search, the

Table 4. Summary of experimental results. Abbreviations: *DFS*: depth-first search, *BFS*: breadth-first search, average: arithmetic means, solved: number of solved instances (out of 27). Uniquely best results in bold fonts for each configuration. Other abbreviations as in Table 1.

	explored states		runtime in s		trace length		solved
	average	median	average	median	average	median	
<i>BFS</i>							
plain	1266489.9	277168.5	53.0	2.7	78.3	52.0	21
<i>CE</i>	140644.4	43610.0	1.1	0.6	91.9	55.5	23
<i>ICB_P</i>	4248527.1	291615.5	102.3	5.2	95.7	59.0	20
<i>ICB_I</i>	4133416.8	241331.5	111.5	2.2	89.7	53.5	20
<i>DFS</i>							
plain	445704.1	63712.5	3.6	0.5	37794.3	9624.5	21
<i>CE</i>	197295.0	15688.0	1.1	0.2	2308.8	795.5	22
<i>ICB_P</i>	3397688.6	409480.0	34.8	2.4	148.7	71.0	21
<i>ICB_I</i>	2371171.5	155694.0	12.9	1.5	16477.9	1819.5	21
d^U							
plain	469310.8	42692.5	107.9	0.3	30415.8	1949.0	20
<i>CE</i>	245070.9	13186.5	1.6	0.2	665.4	230.5	22
<i>ICB_P</i>	3503904.9	152689.5	63.8	1.8	155.0	64.0	21
<i>ICB_I</i>	2380880.4	81524.0	16.5	0.8	892.9	391.5	20
<i>UT</i>	394999.4	23003.0	87.6	0.6	4764.3	1124.5	21
d^L							
plain	701962.7	54742.0	14.9	0.5	16414.1	7725.0	21
<i>CE</i>	189144.3	12181.0	1.3	0.2	1391.6	562.0	22
<i>ICB_P</i>	2783118.2	199548.0	72.7	3.7	100.6	67.0	21
<i>ICB_I</i>	1918955.4	105246.0	17.5	0.9	2642.1	798.0	19
<i>UT</i>	344885.2	46509.0	23.8	1.4	13664.1	2521.0	20
h^L							
plain	143536.3	18009.0	4.1	0.4	3327.0	773.5	25
<i>CE</i>	41090.5	5205.5	1.2	0.3	917.8	108.5	26
<i>ICB_P</i>	2075198.1	374834.5	72.5	6.5	150.5	70.5	21
<i>ICB_I</i>	1052988.1	89065.0	14.1	2.0	1981.0	126.0	21
<i>UT</i>	164821.9	4972.5	9.7	0.3	657.8	97.5	25
h^U							
plain	138325.2	12938.0	3.1	0.3	419.6	112.0	26
<i>CE</i>	22956.1	6523.0	1.0	0.3	213.0	92.0	27
<i>ICB_P</i>	1953710.5	145351.0	73.7	7.3	147.7	66.0	21
<i>ICB_I</i>	965865.2	72437.0	12.7	1.6	302.1	116.0	21
<i>UT</i>	106837.0	2537.0	9.6	0.2	436.3	80.0	26

shortest error traces are obtained with iterative context bounding. However, *CE* scales much better on average, and could solve the most problem instances in *every* configuration. Furthermore, the only configuration that could solve *all* instances is *CE* with the h^U heuristic.

Finally, we also compared *CE* with the exact pruning criterion given by Prop. 1. However, it turned out that this exact criterion is very strict in practice, and did not fire in any case. This effectively means that the results for this criterion are the same as the results for plain search, except for a slightly longer runtime due to the preprocessing.

Overall, our evaluation impressively demonstrated the power of directed model checking with interference contexts and multiple open queues. We have observed that the overall model checking performance could (often significantly) be increased for various search methods and distance heuristics on a range of heuristics and real world problems. The question remains on which parameters the performance of the different search algorithms depend, and in which cases they perform best. We will discuss these points in the next section.

5.4 Discussion

Our context-enhanced search algorithm uses several open queues to handle the different levels of interference. In Table 5, we provide additional results about the number of queues and queue accesses for *CE* and *UT*, as well as the number of context switches for iterative context bounding. The data is averaged over all instances of the specific case study.

For our context enhanced search algorithm, we report the number of queues that are needed for the different problems, as well as the number of pushed and popped states of these queues. First, we observe that in the *F* examples, only one deferred queue (q_1) is needed, which is accessed very rarely. However, the situation changes in the *S*, *M* and *N* examples, where three deferred queues are needed. Deferred states are explored from up to two deferred queues (for d^U and breadth-first search), whereas the last (non-empty) deferred queue is never accessed. Most strikingly, in the *H* examples, we need six deferred queues, from which most of them are never accessed. Overall, the performance of *CE* also depends on the number of explored deferred states. If there are deferred queues that never have to be accessed at all, the corresponding states do not have to be explored, and the branching factor of the system is reduced.

We applied the useless transitions approach in a multi-queue setting, where *useless* states, i. e., states reached by a useless transition, are maintained in a separate queue. Table 5 shows the number of explored non-useless states (q_0) as well as the number of explored useless states (q_1). We observe that for the well-informed h^L heuristic, q_1 is never accessed except for the *H* examples, which explains the good performance of *UT* in this case. However, for the coarser d^U heuristic, q_1 is accessed very often, which explains the favorable performance of *CE* over *UT* with d^U .

Finally, let us give some explanations of the performance of the iterative context bound algorithm *ICB* compared to *CE*. The *ICB* approach is guaranteed to minimize the number of context switches, and obviously performs best in systems where not many context switches are needed. Contrarily, if the context has to be switched n times, the whole state space for all context switches smaller than n has to be traversed until an error state is found, which could be problematic if n is high. Table 5 shows the average number n of context switches needed to find an error state in our examples.² We observe that, except for the *F* examples, n is pretty high for *ICB_P*. Contrarily, in the *F* examples where the context has to be switched only two times, iterative context bounding performs very well. Overall, we conclude from our experiments that the method proposed by

² As a side remark, the different number of context switches for h^L , d^U and *BFS* with *ICB_P* and *ICB_I* are due to the different number of solved instances of these configurations.

Table 5. Average number of queue accesses (for queues q_0, \dots, q_6) for *CE* and *UT*, and average number of context switches for *ICB* per case study. Number of pushed states at the top, number of popped states at the bottom. Abbreviations as in Table 1.

	Accesses with <i>CE</i>							Accesses with <i>UT</i>		Context Switches	
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_0	q_1	<i>ICB_P</i>	<i>ICB_I</i>
<i>h^L</i> heuristic											
F	38 35	114 1	0 0	0 0	0 0	0 0	0 0	13 7	24 0	2	0
S	169161 169144	742370 277922	87766 0	136105 0	0 0	0 0	0 0	1702 1268	2440 0	31.6	0
M	5585 5584	5687 1801	8598 0	3455 0	0 0	0 0	0 0	12308 9693	9137 0	21	3
N	6654 6652	6547 1941	9384 0	4124 0	0 0	0 0	0 0	69534 51716	4332 0	22.33	3
H	377962 377935	294737 93051	230682 0	297175 0	345419	445245	612451	404205 404186	308105 204806	134.8	15.6
<i>d^U</i> heuristic											
F	38 35	114 1	0 0	0 0	0 0	0 0	0 0	14 9	24 0	2	0
S	287161 287149	395163 379679	65564 520	273596 0	0 0	0 0	0 0	12746 12746	1042207 1030510	31.6	3
M	9712 9708	7734 4902	11949 0	5043 0	0 0	0 0	0 0	23355 23354	115058 83849	21	4
N	10006 10000	7943 4399	11920 0	6614 0	0 0	0 0	0 0	16204 16204	109769 80079	21.5	4.75
H	379149 379100	289671 164472	222272 0	289380 0	323717	385445	498216	117141 117141	220791 143929	134.8	15.2
<i>BFS</i>											
F	56 43	114 1	0 0	0 0	0 0	0 0	0 0	n/a n/a	n/a n/a	2	0
S	323745 323704	607706 229784	189050 1111	447035 0	0 0	0 0	0 0	n/a n/a	n/a n/a	31.6	0
M	50251 50249	56932 47908	86116 0	30730 0	0 0	0 0	0 0	n/a n/a	n/a n/a	21	1.25
N	55105 55103	55314 47546	92173 0	39908 0	0 0	0 0	0 0	n/a n/a	n/a n/a	20.67	1.67
H	894952 894927	925034 874677	649694 0	843886 0	861079	980240	1476125	n/a n/a	n/a n/a	134.8	20.8

Musuvathi and Qadeer works well for programs with rather loose interaction where not many context switches are required. However, in protocols with tight interaction and many required changes of the active threads, directed model checking with interference contexts performs better.

6 Conclusion

In this paper, we have introduced context-enhanced directed model checking. This multi-queue search algorithm makes use of interference contexts to determine the de-

gree of relevance of transitions. Our approach is orthogonal to the directed model checking process and can hence be combined with arbitrary heuristics and blind search. Our empirical evaluation impressively shows the potential for various heuristics on large and realistic case studies. We obtain considerable performance improvements compared to plain directed model checking as well as compared to related search algorithms like iterative context bounding or useless transitions.

For the future, it will be interesting to extend and refine our concept of interference contexts. This includes, for example, to take into account the structure of our automaton model more explicitly. In particular, we plan to better adapt our approach to timed automata. Although we are able to handle such systems, our technique is not yet optimized for them as clocks are currently ignored. We expect that taking them into account will further improve our method for that class of systems.

References

1. Edelkamp, S., Schuppan, V., Bosnacki, D., Wijs, A., Fehnker, A., Aljazzar, H.: Survey on directed model checking. In Peled, D., Wooldridge, M., eds.: Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence (MOCHART 2008). Volume 5348 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2009) 65–89
2. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In Valmari, A., ed.: Proceedings of the 13th International SPIN Workshop (SPIN 2006). Volume 3925 of Lecture Notes in Computer Science., Springer-Verlag (2006) 35–52
3. Dräger, K., Finkbeiner, B., Podelski, A.: Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer* **11**(1) (2009) 27–37
4. Hoffmann, J., Smaus, J.G., Rybalchenko, A., Kupferschmid, S., Podelski, A.: Using predicate abstraction to generate heuristic functions in Uppaal. In Edelkamp, S., Lomuscio, A., eds.: Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MOCHART 2006). Volume 4428 of Lecture Notes in Computer Science., Springer-Verlag (2007) 51–66
5. Smaus, J.G., Hoffmann, J.: Relaxation refinement: A new method to generate heuristic functions. In Peled, D., Wooldridge, M., eds.: Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence (MOCHART 2008). Volume 5348 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2009) 146–164
6. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer* **5**(2) (2004) 247–267
7. Qian, K., Nymeyer, A.: Guided invariant model checking based on abstraction and symbolic pattern databases. In Jensen, K., Podelski, A., eds.: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Volume 2988 of Lecture Notes in Computer Science., Springer-Verlag (2004) 497–511
8. Kupferschmid, S., Hoffmann, J., Larsen, K.G.: Fast directed model checking via russian doll abstraction. In Ramakrishnan, C.R., Rehof, J., eds.: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Volume 4963 of Lecture Notes in Computer Science., Springer-Verlag (2008)

9. Wehrle, M., Helmert, M.: The causal graph revisited for directed model checking. In Palsberg, J., Su, Z., eds.: *Proceedings of the 16th International Symposium on Static Analysis (SAS 2009)*. Volume 5673 of *Lecture Notes in Computer Science.*, Springer-Verlag (2009) 86–101
10. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* **14** (2001) 253–302
11. Helmert, M.: The Fast Downward planning system. *Journal of Artificial Intelligence Research* **26** (2006) 191–246
12. Wehrle, M., Kupferschmid, S., Podelski, A.: Useless actions are useful. In Rintanen, J., Nebel, B., Beck, J.C., Hansen, E., eds.: *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*, AAAI Press (2008) 388–395
13. Wehrle, M., Kupferschmid, S., Podelski, A.: Transition-based directed model checking. In Kowalewski, S., Philippou, A., eds.: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*. Volume 5505 of *Lecture Notes in Computer Science.*, Springer-Verlag (2009) 186–200
14. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In Ferrante, J., McKinley, K.S., eds.: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, ACM Press (2007) 446–455
15. Pearl, J.: *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley (1984)
16. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* **4**(2) (1968) 100–107
17. Godefroid, P.: *Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem*. Volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag (1996)
18. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. The MIT Press (2000)
19. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Partial-order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology Transfer* **6**(4) (2004) 277–301
20. Kupferschmid, S., Wehrle, M., Nebel, B., Podelski, A.: Faster than Uppaal? In Gupta, A., Malik, S., eds.: *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*. Volume 5123 of *Lecture Notes in Computer Science.*, Springer-Verlag (2008)
21. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* **126**(2) (1994) 183–235
22. Dierks, H.: Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing* **16**(2) (2004) 104–120
23. 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.: *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*. Volume 1709 of *Lecture Notes in Computer Science.*, Springer-Verlag (1999) 1186–1205
24. Lamport, L.: A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems* **5**(1) (1987) 1–11