Transition-based Directed Model Checking

Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski

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

Abstract. Directed model checking is a well-established technique that is tailored to fast detection of system states that violate a given safety property. This is achieved by influencing the order in which states are explored during the state space traversal. The order is typically determined by an abstract distance function that estimates a state's distance to a nearest error state. In this paper, we propose a general enhancement to directed model checking based on the evaluation of *state transitions*. We present a schema, parametrized by an abstract distance function, to evaluate transitions and propose a new method for the state space traversal. Our framework can be applied automatically to a wide range of abstract distance functions. The empirical evaluation impressively shows its practical potential. Apparently, the new method identifies a sweet spot in the trade-off between scalability (memory consumption) and short error traces.

1 Introduction

When model checking safety properties, the ultimate goal is to prove the absence of error states. This can be done by exploring the entire reachable state space. However, the state space of realistic applications is often too large to be enumerated exhaustively because of the state explosion problem. Directed model checking is a well-established technique to tackle this problem and has found its way in state-of-the-art tools such as SPIN, PATHFINDER or UPPAAL [5, 6, 11]. In directed model checking, the state space traversal is guided ("directed") based on specific criteria towards error states. Generally, these guidance criteria are automatically extracted from the model by taking an abstraction of the model and computing an abstract distance function $d^{\#}$. For a state s, the value $d^{\#}(s)$ approximates the distance of s to a nearest error state. These values are used during the state space traversal in order to determine which state is explored next.

Each different version of directed model checking thus arises through the choice of the abstraction that is used to compute the abstract distance function, and by the choice of the basic (non-deterministic) algorithm for traversing the state space. Earlier work on directed model checking was mainly focused on the first point, i. e., in defining abstractions that lead to distance estimation functions $d^{\#}$ to guide the state space traversal efficiently towards an error state [3, 4, 5, 9, 12, 13, 17, 18]. Considering the second point, there are two predominantly used algorithms of directed model checking, namely A* and *greedy search* (cf. [16]). A* is guaranteed to find shortest possible error traces

for certain kinds of distance estimation functions, but is often too memory consuming for large systems. Greedy search does not necessarily find shortest possible error traces, but mostly scales much better than A* in practice.

In this paper, we present a new version of directed model checking that seems to identify a sweet spot in the trade-off between scalability (memory consumption) and short computed error traces. It is based on the concept of *useless transitions* which is an adaptation of the *useless actions* approach that has been introduced in the context of AI Planning [20]. As indicated by its name, the concept of useless transitions extends directed model checking by additionally evaluating transitions, not just states. We will see that this is a general concept in the sense that useless transitions can be computed fully automatically with the given distance estimation function $d^{\#}$. That is, whatever the choice of the underlying abstraction for computing the abstract distance function has been, we can use the already computed abstract distance function in order to effectively recognize useless transitions. We will characterize a class of distance estimation functions for which our method is suited best. We define a new (non-deterministic) strategy for state space traversal that takes these useless transitions into account. The new strategy is an amalgam of the two strategies A* and greedy search. For the two extreme cases of abstraction, it becomes the former or the latter, respectively.

We have implemented our method and have applied it to a number of academic and industrial benchmarks. This allowed us to experimentally compare the new directed model checking method with the two existing predominant methods A* and greedy search. The empirical results impressively show the benefit of our approach: We obtain almost shortest error traces, whereas the number of expanded states reduces significantly compared to A* and also to greedy search in most cases.

We will next give an example that greatly oversimplifies the issues at hand but gives an intuition about useless transitions and their potential usefulness.

Example. Figure 1 depicts a system consisting of $n+1$ parallel components A_0 to A_n. The initial state of the system is $(s_0^0,\ s_0^1,\ \ldots, s_0^n)$ and the error state is $(s_1^0,\ s_1^1,\ \ldots, s_1^n)$. Suppose that we apply directed model checking to check if this error state is reachable. Further suppose that we therefore use the *maximum graph distance* as the abstract distance function [4, 5]. This function is based on the local distance of each single automaton A_i. More precisely, let $d(i)$ denote the graph distance from A_i's current location to A_i's error state, then the maximum graph distance is defined as $\max_{i=0,\ldots,n} d(i)$.

It turns out that, for this problem, the maximum graph distance is a rather uninformed distance function. It cannot distinguish states that are nearer to the error state from others. If there is at least one local state of the form s_0^i, then the abstract distance value is 1. We may characterize the state space topology induced by this abstract distance function as follows. There is one single plateau (for all of the 2^{n+1} reachable system states but for the error state the abstract distance is 1). This means that the guidance based on this abstract distance function is very poor. In fact, for every abstract distance function, the similar situation arises.

In the example, it is trivial to see that each state transition where a local state leaves a local error state (depicted by a double circle) should be avoided as much as possible during the state space traversal (it is a useless transition!). Without steps corresponding

to such transitions, the state space traversal stops after $n + 1$ steps and returns the shortest possible error path.

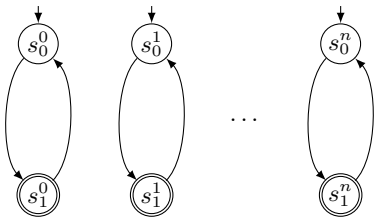

Fig. 1. An automata system with $n + 1$ components

The remainder of this paper is organized as follows. In the next section, we discuss related work. In Section 3, we give the preliminaries needed for this work, including a more detailed introduction to directed model checking. In Section 4, we introduce the notion of useless transitions and present our directed model checking algorithm based on this concept. In Section 5, we empirically evaluate our algorithm on a number of benchmarks. Section 6 concludes the paper.

2 Related Work

Research on directed model checking so far mainly focused on finding abstractions that lead to efficient distance estimation functions. It was pioneered by Edelkamp et al. [4, 5] who proposed to base the distance estimation on the graph distance (see also the example in the previous section). This is a rather coarse abstraction that leads to distance estimation functions that are easy to compute on the one hand, but that are not very informative on the other hand. Kupferschmid et al. [12] have introduced two abstract distance functions based on the *monotonicity abstraction*. This abstraction technique is an adaptation of the *ignoring-delete-lists* principle that has originally been introduced in the area of AI Planning [1]. The basic idea is that every state variable, once it obtained a value, keeps that value forever. Therefore, the value of a variable is no longer an element, but a subset of its domain. The distance estimation values are then computed by iteratively applying transitions under this abstraction until an error state is reached, and then returning the abstract error length as estimation.

Furthermore, Dräger et al. [3] iteratively "merge" a pair of automata, i. e., compute their product and then merge locations until there are at most n locations left, where n is an input parameter. The distance estimation function is read off the overall merged automaton. Moreover, several distance estimation functions based on *pattern databases* have been proposed [9, 13, 17, 18]. A pattern database heuristic function abstracts a problem by ignoring some of the relevant symbols, e. g., some of the state variables. The state space of the abstracted problem is built completely as a pre-process to search, and is used as a look-up table for the heuristic values during search.

The problem of evaluating state transitions has been studied mostly in the area of AI Planning. In this context, an approach to avoid *useless actions* has been proposed which has led to a significantly improved search behavior on a wide range of planning instances [20]. In Section 4, we adapt this technique to the context of directed model checking of concurrent systems with interleaving and binary synchronization. Complementary to useless actions, Hoffmann and Nebel [8] and Helmert [7] proposed what they call *helpful actions* and *preferred operators*, respectively. These methods are used to select a set of promising successors to a search state. The helpfulness of a transition is determined during the computation of the distance estimation values. These values are obtained by solving an abstract problem. Roughly speaking, a transition is considered as helpful if it is contained in that abstract solution. However, this approach is specific to the applied distance function.

3 Preliminaries

In this section, we give the basic notation as well as a formal definition of the considered models. Section 3.2 introduces directed model checking, A* and greedy search.

3.1 Notation

In our setting, an automaton is a tuple $A = (S,\, s^0,\, T,\, \Sigma)$, where S is a finite set of states, $s^0 \in S$ is the initial state, $T \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ is a set of labeled transitions, Σ a finite set of synchronization labels, and $\tau \notin \Sigma$ denotes a special internal label. A transition $(s,\, \alpha,\, s') \in T$ is also denoted by $s \xrightarrow{\alpha} s'$.

Let N be the set $\{1, \ldots, n\}$. For n automata $A_i = (S_i,\, s_i^0,\, T_i,\, \Sigma_i)$, $i \in N$, with pairwise disjoint sets of states, the parallel composition $A_1 \,||\, \ldots \,||\, A_n$ is defined by the product automaton $(S^\times,\, (s_1^0, \ldots, s_n^0),\, T^\times,\, \Sigma_1 \cup \cdots \cup \Sigma_n)$, where $S^\times = S_1 \times \cdots \times S_n$, and the set of transitions $T^\times \subseteq S^\times \times \{\tau\} \times S^\times$ is defined as follows. There is a transition $(s_1, \ldots, s_n) \xrightarrow{\tau} (s'_1, \ldots, s'_n) \in T^\times$ iff one of the following conditions holds.

1. There exists $i \in N$ such that $s_i \xrightarrow{\tau} s'_i \in T_i$, and $s_k = s'_k$ for all $k \in N \setminus \{i\}$.
2. There exist $i, j \in N$ with $i \neq j$, and there exists a label $\alpha \in (\Sigma_i \cap \Sigma_j)$ such that $s_i \xrightarrow{\alpha} s'_i \in T_i$ and $s_j \xrightarrow{\alpha} s'_j \in T_j$, and $s_k = s'_k$ for all $k \in N \setminus \{i, j\}$.

A system $\mathcal{S}$ of n automata $A_1, \ldots, A_n$ is the parallel composition $A_1 \,||\, \ldots \,||\, A_n$. Note that in a parallel system only τ transitions occur; two synchronized transitions in automata A_i and A_j also correspond to a τ transition in $\mathcal{S}$. We address the falsification of invariants; in CTL, these properties take the form $A\,G\,\varphi$. In this paper, φ is a formula of the form $\bigwedge_i \neg s_i$, where $s_i \in S_1 \times \cdots \times S_n$ are *error* states. We call a tuple $\mathcal{T} = \langle \mathcal{S},\, \varphi \rangle$ a model checking task. A *trace* $\pi = s_0,\, t_1,\, s_1,\, \ldots,\, t_n,\, s_n$ is an alternating sequence of states and transitions where t_i is an outgoing transition of s_{i-1}. We call a trace that leads to a state that satisfies $\neg\varphi$ an *error trace*. The length of a trace $|\pi|$ is defined as the number of transitions in π, i. e., $|\pi| = n$.

3.2 Directed Model Checking

Directed model checking describes the task of finding error states where the state space traversal is guided ("directed") by a distance estimation function $d^{\#}$. This function is computed fully automatically based on the declarative description of the system and an abstraction principle (e. g., the monotonicity abstraction [12]). In a nutshell, $d^{\#}$ is a function that maps states to integers, reflecting an estimate of the shortest error distance. Typically, this estimate is the length of a corresponding abstract error trace. States are evaluated with $d^{\#}$, states with lower values are preferred. Note that abstract distance functions influence the *order* in which the states are explored, thereby completeness is *not* affected. On the one hand, it is desirable to have distance functions that are as informative as possible. On the other hand, the computation must not be too expensive.

Figure 2 shows a basic directed model checking algorithm. Given a model checking task $\langle \mathcal{S}, \varphi \rangle$ and an abstract distance function $d^{\#}$, the algorithm returns False if there is a state that violates φ, otherwise it returns True. The initial state of $\mathcal{S}$ is s^0. 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 close. Every state encountered during search is first checked if it is an error state. 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. For A*, evaluate(s, $d^{\#}$) returns $d^{\#}(s) + c(s)$, 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 $d^{\#}(s)$. When every successor has been computed and prioritized, the process continues with the next state from open with lowest priority value. Every state stores information about how it has been reached, i. e., its immediate predecessor state and transition. Therefore, if an error state s is finally reached, the corresponding error trace is generated by back tracing from s.

For distance estimation functions that are *admissible*, i. e., that never overestimate the real error distance, A* is guaranteed to find shortest possible error traces [16].

4 Transition-based Directed Model Checking

Until now, directed model checking algorithms have roughly followed the scheme as outlined in the last section by evaluating *states*, thereby suffering from the fact that A* is often not practical and the error traces of greedy search are often of poor quality. In this section, we propose an extension based on transition evaluation. We will first define the theoretical concept of useless transitions and then its practical counterpart, the relatively useless transition. This notion can be directly used to combine A* and greedy search to a new *transition-based* directed model checking algorithm.

4.1 Useless and Relatively Useless Transitions

In this section, we give the definition of useless transitions. We will first give an exact notion that captures precisely our intuition on the one hand, but is computationally hard

```
1  function verify(S, φ, d#):
2      open = empty priority queue, closed = ∅
3      priority = evaluate(s^0, d#)
4      open.insert(s^0, priority)
5      while open is not empty:
6          s = open.getMinimum()
7          if s violates φ:
8              return False
9          if s ∉ closed:
10             closed = closed ∪ {s}
11             for each outgoing transition t of s:
12                 s' = successor(s, t)
13                 if s' ∉ closed ∪ open:
14                     priority = evaluate(s', d#)
15                     open.insert(s', priority)
16     return True
```

Fig. 2. A basic directed model checking algorithm

on the other hand. Therefore, we will investigate ways to approximate this definition, leading to the concept of relatively useless transitions.

Intuitively, a transition is useless if it is not needed to reach the nearest error state on a shortest path. This is formally stated in the next definition.

Definition 1 (Useless Transition). *Let $\langle \mathcal{S}, \varphi \rangle$ be a model checking task, where $\mathcal{S} = (S, s^0, T, \Sigma)$. A transition $t \in T$ leading from a state s to state s' is* useless *in s iff no shortest trace from s to a nearest error state starts with this transition.*

We use $d(s)$ to denote the distance of a state s to a nearest error state. More precisely, $d(s) = n$ if there is a trace π from s to an error state with $|\pi| = n$ and there is no trace π' from s to an error state with $|\pi'| < n$. When we want to stress that d is a function also on the system $\mathcal{S}$, we will write $d(\mathcal{S}, s)$.

By Definition 1, a transition t is useless in a state s if and only if the real error distance d does not decrease by one, i. e., a transition from s to s' is useless iff $d(s) \leq d(s')$. To see this, recall that $d(s) \leq d(s') + 1$ for every transition. If a shortest error trace starts from s with t, then $d(s) = d(s') + 1$. Otherwise the error distance increases, i. e., $d(s) < d(s') + 1$. Since the distance values are all integers, this is equivalent to $d(s) \leq d(s')$. We will use the inequality $d(s) \leq d(s')$ in connection with the idea of *removing* a useless transition. Therefore, we will define the notion of *reduced systems*. To do this, we first need some more terminology. For a system $\mathcal{S} = A_1 \mid\mid \ldots \mid\mid A_n$, say $\mathcal{S} = (S, s^0, T, \Sigma)$, we define a function $\mu_{\mathcal{S}}$ that maps transitions from $\mathcal{S}$ to the corresponding transitions in $A_i = (S_i, s_i^0, T_i, \Sigma_i)$. This function $\mu_{\mathcal{S}} : T \to 2^{T_1 \cup \cdots \cup T_n}$ is defined as follows.

$$\mu_{\mathcal{S}}((s_1, \ldots, s_n) \xrightarrow{\tau} (s'_1, \ldots, s'_n)) = \begin{cases} \{s_i \xrightarrow{\tau} s'_i\} & \exists i \in \{1, \ldots, n\} \\ \{s_i \xrightarrow{\alpha} s'_i,\ s_j \xrightarrow{\alpha} s'_j\} & \exists i, j \in \{1, \ldots, n\},\ i \neq j,\ \exists \alpha \in (\Sigma_i \cap \Sigma_j) \end{cases}$$

Based on this definition, we now define *reduced systems*.

Definition 2 (Reduced system). *Let $\mathcal{S} = A_1 \mid\mid \ldots \mid\mid A_n$ be a system, say $\mathcal{S} = (S, s^0, T, \Sigma)$, with $A_i = (S_i, s_i^0, T_i, \Sigma_i)$ for $i \in \{1, \ldots, n\}$. Let $t \in T$ be a transition. The* reduced system *with respect to t is defined as $\mathcal{S}_t = A'_1 \mid\mid \ldots \mid\mid A'_n$, where $A'_i = (S_i, s_i^0, T_i \setminus \mu_{\mathcal{S}}(t), \Sigma_i)$.*

Note that, according to the definition of $\mu_{\mathcal{S}}$, at most two automata A_i are affected by reducing the system (one in the case of interleaving, two in the case of binary synchronization). Roughly speaking, a transition t of the system $\mathcal{S}$ corresponds to one or two transitions of one or two automata of $\mathcal{S}$. The reduced system $\mathcal{S}_t$ is obtained by removing these transitions from the corresponding automata. Note that removing *one* transition from an automaton A removes *several* transitions from the system $\mathcal{S}$.

Based on the definition of reduced systems, we will give a proposition that leads to a testing criterion for useless transitions.

Proposition 1. *Let $\langle \mathcal{S}, \varphi \rangle$ be a model checking task with $\mathcal{S} = (S, s^0, T, \Sigma)$, $s, s' \in S$, $t \in T$ leading from s to s'. If $d(\mathcal{S}_t, s) \leq d(\mathcal{S}, s')$, then t is useless in s.*

Proof. If $d(\mathcal{S}_t, s) \leq d(\mathcal{S}, s')$, then $d(\mathcal{S}, s) \leq d(\mathcal{S}, s')$ because $d(\mathcal{S}, s) \leq d(\mathcal{S}_t, s)$ (the error distance cannot decrease in reduced systems). As $d(s) \leq d(s')$ iff t is useless in s, the claim follows directly.

This characterization can be interpreted as follows. A transition t is useless in s if the error state is still reachable from s on the same shortest trace when the corresponding transitions to t are *removed* from the system. However, it is not practical as computing exact distances is PSPACE-hard. A direct way to approximate this test is to use the given distance estimation function $d^\#$ instead of d. This is rational because $d^\#$ is designed for exactly the reason of approximating d. When we want to stress that $d^\#$ is a function also on the system S, we will write $d^\#(\mathcal{S}, s)$.

Definition 3 (Relatively Useless Transition). *Let $\langle \mathcal{S}, \varphi \rangle$ be a model checking task with $\mathcal{S} = (S, s^0, T, \Sigma)$, $s, s' \in S$, $t \in T$ leading from s to s'. Let $d^\#$ be a distance estimation function. Then t is* relatively useless *for $d^\#$ in s if $d^\#(\mathcal{S}_t, s) \leq d^\#(\mathcal{S}, s')$.*

Note that this is exactly the testing criterion from Proposition 1 where d has been replaced by $d^\#$. Obviously, the quality of this approximation strongly depends on $d^\#$'s precision. A very uninformed function, e. g. a function that constantly returns zero, recognizes every transition as relatively useless. However, the more sophisticated the distance estimation, the more precise is the approximation. We will come back to this point in the next section. Intuitively, taking a relatively useless transition t does not seem to guide the state space traversal towards an error state as the *stricter* distance estimate in $\mathcal{S}_t$ does not increase.

One would expect that transitions should not be relatively useless if they lead to states nearer to an error state. Indeed, under the rational assumption that distance functions $d^\#$ never decrease their estimate in reduced systems, i. e., $d^\#(\mathcal{S}, s) \leq d^\#(\mathcal{S}_t, s)$ for all systems $\mathcal{S}$, transitions t and states s, transitions leading to better estimates are *never* relatively useless in any system $\mathcal{S}$.

Proposition 2. *Let $\langle \mathcal{S}, \varphi \rangle$ be a model checking task with $\mathcal{S} = (S, s^0, T, \Sigma)$. Let $d^\#$ be a distance estimation function such that $d^\#(\mathcal{S}, s) \leq d^\#(\mathcal{S}_t, s)$ for all $s \in S$ and $t \in T$. Let $s, s' \in S$ be states and $t \in T$ be a transition that leads from s to s'. If $d^\#(s') < d^\#(s)$, then t is* not *relatively useless for $d^\#$ in s.*

Proof. Assume that t is relatively useless, i. e., $d^\#(\mathcal{S}_t, s) \leq d^\#(\mathcal{S}, s')$. As $d^\#(\mathcal{S}, s) \leq d^\#(\mathcal{S}_t, s)$, we have $d^\#(\mathcal{S}, s) \leq d^\#(\mathcal{S}, s')$, showing that the distance estimate does not decrease when the relatively useless transition t is applied.

4.2 Directed Model Checking with Relatively Useless Transitions

In this section, we put the pieces together. So far, we have presented a notion of useless transitions to identify transitions that should be less preferred during the state space traversal. A direct way to integrate this information is to "penalize" states that result from applying such a transition. This is rational because avoiding transitions that are not likely to appear in shortest error traces is likely to improve the detection of (short) error traces. States that are reached by applying such a useless transition should be less preferred when traversing the state space.

As argued in the introduction and Section 3.2, there are two choices to be made when the directed model checking approach is applied, namely choosing the underlying abstraction for the distance estimation functions, and choosing the algorithm that is essentially determined by the *evaluate* function that computes the priority values for the states. Here, we assume that a distance estimation function $d^\#$ is already given, and $d^\#$ is additionally used to determine relatively useless transitions. For the second point, we give a simple extension of the *evaluate* function in Fig. 3. Recall that s and t (lines 2 and 3) are stored in the successor state and can be accessed easily. As outlined above, states that result from applying a relatively useless transition are "penalized". As penalty value for s, we chose $c(s)$, the length of the trace on which s was reached for the first time. This leads to a combination of A* and greedy search as discussed in more detail below.

```
1 function evaluate(s', d#):
2     s = predecessor of s'
3     t = transition from s to s'
4     if t is relatively useless for d# in s:
5         priority = d#(s') + c(s')
6     else:
7         priority = d#(s')
8     return priority
```

Fig. 3. Evaluation function based on relatively useless transitions

Overall, this algorithm is an amalgam of the algorithms A* and greedy search based on transition evaluation. Its behavior depends on the accuracy of the underlying distance estimation function $d^\#$. As mentioned earlier, the more accurate $d^\#$, the more transitions are classified correctly, and therefore, the more it tends towards greedy search. At

the extreme ends of the spectrum, it becomes greedy search (for the perfect distance function that classifies every transition correctly) and breadth first search, respectively, which is a degenerated version of A* for the distance function that constantly returns zero. From this perspective, our algorithm can be considered as a combination of greedy search and A*.

4.3 Discussion

Although it is technically possible to apply our algorithm to every model checking task, there are distance functions that are probably best suited for this concept. Let us have a look at this class of functions. Roughly speaking, distance estimation functions can be divided into two classes, namely those that compute the values on-the-fly by solving an abstract problem in every search state, and those that do it in a preprocessing step, typically by computing a lookup table (e. g., a pattern database). The concept of useless transitions seems to be best suited for distance functions that are computed on-the-fly because the time overhead to compute this information is comparatively low. Contrarily, distance functions from the second class are less suited because for every modified system, an additional pattern database has to be computed (recall that for the computation of the useless-values, the system is modified and the distance value is recomputed on this modified system). However, as we will see, for distance functions computed on-the-fly, the overall performance can often be significantly improved.

The performance of our approach strongly depends on the quality of $d^\#$ that is used to guide the search and to determine useless transitions. The higher the precision of $d^\#$, the more transitions are evaluated correctly, and hence, the better the overall performance as many unnecessary states need not be considered. In small examples, distance functions like the graph distance could already lead to improvements. Pointing to our motivating example in the introduction, we recognize that all transitions corresponding to edges from down to up are relatively useless for the graph distance heuristic, whereas all other transitions are not. In this example, applying our algorithm leads to a shortest possible error trace with dramatically smaller explored state space than with greedy search or A*. For more complex examples, more sophisticated distance functions are needed to benefit from our approach, as we will empirically show in the next section.

5 Evaluation

We have implemented our algorithm in the model checker MCTA [14] as part of a tool development effort within the AVACS project[1]. The tool and its source code are freely available at http://mcta.informatik.uni-freiburg.de/. We compare our search method with A* and greedy search. In addition to the automaton model as considered in this paper, many of them feature integer and clock variables and represent timed automata. Transitions can additionally be guarded by integer and clock constraints. Moreover, a transition can change the value of integer variables and reset clock variables. Note that the concept of useless transitions is general and can be

[1] http://www.avacs.org/

adapted to that class of automata in a straightforward way. To get a conservative approximation of useless transitions, we have implemented our concept in a stronger way as described in the last section. When $\mu_{\mathcal{S}}(t)$ is computed for a system $\mathcal{S} = A_1||\ldots||A_n$ and a transition t in $\mathcal{S}$, we additionally remove transitions in the automata A_i that read variables that are set by some $t' \in \mu_{\mathcal{S}}(t)$, and transitions that lead to the same state as some $t' \in \mu_{\mathcal{S}}(t)$.

5.1 The Distance Estimation Functions

We evaluated our algorithm for a number of distance estimation functions. We give detailed results for the distance functions h^L and h^U introduced by Kupferschmid et al. [12] and for the distance function based on the maximum graph distance h^{gd} introduced by Edelkamp et al. [4, 5]. As outlined in Section 2, h^L and h^U are based on the monotonicity abstraction principle, where a state variable can have multiple values simultaneously. h^L 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 distance estimate. 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. Observe that computing h^U is more expensive than h^L. As we will see in Section 5.3, this pays off in better search behavior. The maximum graph distance function h^{gd} uses the graph distance as indicated by its name.

5.2 The Benchmark Set

Our benchmarks stem from the AVACS benchmark suite.

Industrial benchmarks The M and N examples come from a case study that 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 in [2]. The C examples stem from a case study from an industrial project partner of the UniForM-project [10] 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 communication between these automata is faulty.

Academic benchmarks The F^A and F^B examples are flawed versions of the *Fischer protocol* for mutual exclusion (cf. [15]). The variants differ in the way they encode the error condition. As a second set of benchmarks, we use *arbiter trees* to establish mutual exclusion between 2^k client processes [19]. The benchmarks A_2–A_6 contain arbiter trees of height 2–6, with an exponentially growing number of processes.

5.3 Experimental Results

The reported experimental results were obtained on a 2.66 GHz Intel Xeon computer with memory out at 4 GB and a Linux operating system. We compare our new state

space traversal technique, denoted *UT*, with A* and greedy search (*G*) in three different configurations. In the first configuration, h^L is used as the abstract distance function, the second uses h^U and the third configuration uses h^{gd}.

Table 1 shows the results of the first configuration. Here, the number of explored states significantly decreases compared to A* and we are able to solve much larger problems. Compared to greedy search, the length of the found error traces are significantly shorter. Moreover, due to better search guidance, we additionally often get significant improvements in terms of the number of explored states and traversal time.

Table 1. Experimental results for h^L with A*, greedy search (*G*), and our combined approach (*UT*). Abbreviations: #a: number of parallel automata, #v: number of variables, *memory*: peak memory used in MB, y e+x: $y \cdot 10^x$, dashes indicate out of memory ($>$ 4 GB).

			explored states			runtime in s			memory			trace length		
Inst.	#a	#v	A*	G	UT	A*	G	UT	A*	G	UT	A*	G	UT
C_1	5	15	22501	1928	1658	0.16	0.04	0.06	9	8	8	54	100	91
C_2	6	17	66791	4566	1333	0.48	0.09	0.08	14	8	8	54	132	91
C_3	6	18	76777	6002	1153	0.58	0.11	0.06	15	9	8	54	128	91
C_4	7	20	726516	81131	1001	5.34	1.00	0.10	70	19	8	55	344	121
C_5	8	22	6.00e+6	430494	833	44.64	5.32	0.12	484	63	8	56	1057	114
C_6	9	24	–	4.56e+6	833	–	48.00	0.17	–	521	9	–	3217	114
C_7	10	26	–	–	829	–	–	0.22	–	–	9	–	–	114
C_8	10	27	–	1.19e+7	816	–	110.81	0.18	–	1158	9	–	5644	95
C_9	10	28	–	2.77e+7	13423	–	252.66	2.27	–	2534	22	–	5803	90
M_1	3	15	34680	4581	4256	0.17	0.02	0.02	9	8	8	47	457	97
M_2	4	17	135073	15832	8186	0.75	0.08	0.06	15	10	9	50	1124	146
M_3	4	17	155164	7655	10650	0.88	0.04	0.07	15	9	10	50	748	91
M_4	5	19	584221	71033	22412	4.27	0.44	0.19	38	19	15	53	3381	136
N_1	3	18	80541	50869	5689	0.97	1.26	0.06	17	45	9	49	26053	108
N_2	4	20	332486	30476	22763	5.00	0.31	0.24	37	19	14	52	1679	259
N_3	4	20	406908	11576	35468	6.66	0.12	0.42	38	12	15	52	799	204
N_4	5	22	1.59e+6	100336	142946	33.78	1.14	1.86	117	40	39	55	2455	792
F_5^A	6	6	71	9	9	0.00	0.00	0.00	7	7	7	8	8	8
F_{10}^A	11	11	511	9	9	0.00	0.00	0.00	8	7	7	8	8	8
F_{15}^A	16	16	1701	9	9	0.05	0.00	0.00	16	7	7	8	8	8
F_5^B	5	6	54	179	7	0.00	0.00	0.00	7	7	7	6	12	6
F_{10}^B	10	11	429	86378	7	0.01	1.29	0.00	8	109	7	6	22	6
F_{15}^B	15	16	1504	–	7	0.04	–	0.00	17	–	7	6	–	6
A_2	8	0	73	36	15	0.00	0.00	0.00	7	7	7	12	21	12
A_3	16	0	5168	206	32	0.08	0.01	0.01	8	7	7	17	24	17
A_4	32	0	4.44e+6	76811	95	95.95	7.53	0.06	1060	65	9	22	42	22
A_5	64	0	–	263346	34	–	50.83	0.11	–	325	13	–	112	27
A_6	128	0	–	–	39	–	–	0.49	–	–	30	–	–	32

The results for the second configuration are depicted in Table 2. Note that h^U is more informative than h^L, and search behavior therefore is mostly better (in particu-

lar, the Fischer protocol examples are trivial for h^U). This fact directly influences the performance when applied with our algorithm: With *UT* and h^U, we obtain even better results than with *UT* and h^L. Note that h^U is not admissible, which means that there is no guarantee to obtain a shortest possible error trace in this setting in theory. However, in practice, we obtained shortest possible traces in our examples with A*. The trace length of *UT* is still mostly shorter than with greedy search. In particular, note that for both h^L and h^U configurations without *UT*, the large C examples C_7–C_9 could only be solved with an error trace of very poor quality compared to *UT*. Moreover, the largest arbiter example A_6 could not be solved at all without *UT* within 4 GB of memory.

Table 2. Experimental results for h^U. Abbreviations as in Table 1

			explored states			runtime in s			memory			trace length		
Inst.	#a	#v	A*	G	UT	A*	G	UT	A*	G	UT	A*	G	UT
C_1	5	15	12480	715	277	0.22	0.02	0.02	9	7	7	54	73	59
C_2	6	17	35047	1612	242	0.56	0.05	0.03	12	7	7	54	99	59
C_3	6	18	39755	734	228	0.68	0.03	0.03	12	7	7	54	86	59
C_4	7	20	359376	9120	566	5.46	0.15	0.12	50	9	8	55	139	55
C_5	8	22	2.88e+6	83911	190	42.01	1.08	0.06	325	18	8	56	300	56
C_6	9	24	2.89e+7	718015	190	374.72	6.39	0.08	3122	79	8	56	864	56
C_7	10	26	–	2.55e+6	184	–	21.74	0.10	–	236	8	–	2412	56
C_8	10	27	–	1.11e+7	570	–	145.24	0.28	–	1237	9	–	3733	94
C_9	10	28	–	–	1225	–	–	0.67	–	–	10	–	–	153
M_1	3	15	33999	7668	4366	0.16	0.04	0.03	9	8	8	47	71	73
M_2	4	17	124237	18847	2036	0.71	0.11	0.02	14	10	8	50	119	81
M_3	4	17	157173	19597	12829	0.93	0.11	0.11	16	10	11	50	124	89
M_4	5	19	562527	46170	9873	4.30	0.28	0.11	40	16	12	53	160	97
N_1	3	18	78798	9117	5191	0.96	0.08	0.05	17	9	9	49	99	80
N_2	4	20	279853	23462	3260	4.17	0.24	0.04	35	15	9	52	154	136
N_3	4	20	378963	43767	19271	6.16	0.47	0.22	39	20	14	52	147	149
N_4	5	22	1.32e+6	152163	15102	26.91	1.97	0.20	110	54	18	55	314	377
F_5^A	6	6	9	9	9	0.00	0.00	0.00	7	7	7	8	8	8
F_{10}^A	11	11	9	9	9	0.00	0.00	0.00	7	7	7	8	8	8
F_{15}^A	16	16	9	9	9	0.00	0.00	0.00	7	7	7	8	8	8
F_5^B	5	6	7	7	7	0.00	0.00	0.00	7	7	7	6	6	6
F_{10}^B	10	11	7	7	7	0.00	0.00	0.00	7	7	7	6	6	6
F_{15}^B	15	16	7	7	7	0.00	0.00	0.00	7	7	7	6	6	6
A_2	8	0	20	25	20	0.00	0.01	0.00	7	7	7	12	21	18
A_3	16	0	25	82	27	0.00	0.01	0.01	7	7	7	17	18	17
A_4	32	0	213	39	34	0.06	0.02	0.05	9	8	9	22	28	22
A_5	64	0	187148	4027	42	42.68	1.22	0.23	414	17	13	27	47	27
A_6	128	0	–	–	50	–	–	1.10	–	–	31	–	–	32

Table 3 gives the results for the third configuration (h^{gd}). Here, we observe that the results with *UT* are less significant than with the first two configurations. This is because, having a closer look at the distance estimation values in many of the instances,

the estimation values are often constant. This is due to the very coarse abstraction (i. e., the graph distance) used by h^{gd}. Therefore, too many transitions are relatively useless for this distance function, causing the search process to degenerate towards A*. However, *UT* mostly still explores less states than A*, thereby producing *significant* shorter error traces than greedy search.

Table 3. Experimental results for h^{gd}. Abbreviations as in Table 1

			explored states			runtime in s			memory			trace length		
Inst.	#a	#v	A*	G	UT	A*	G	UT	A*	G	UT	A*	G	UT
C_1	5	15	56496	18796	32583	0.12	0.06	0.12	11	9	10	54	1167	61
C_2	6	17	185109	66389	107175	0.49	0.22	0.42	20	14	17	54	1847	69
C_3	6	18	240090	94536	133529	0.68	0.33	0.55	24	17	20	54	2153	68
C_4	7	20	2.45e+6	1.11e+6	1.27e+6	8.00	3.80	5.84	160	100	124	55	6805	71
C_5	8	22	2.28e+7	1.27e+7	1.07e+7	82.79	43.50	56.38	1319	877	976	56	35067	67
C_6	9	24	–	–	–	–	–	–	–	–	–	–	–	–
M_1	3	15	44611	12277	19333	0.21	0.07	0.10	9	9	8	47	2779	95
M_2	4	17	176429	43784	67184	0.96	0.28	0.33	16	14	11	50	11739	105
M_3	4	17	188472	54742	84020	1.05	0.37	0.46	15	15	12	50	12701	113
M_4	5	19	706127	202924	319485	5.02	1.69	1.98	41	43	26	53	51402	218
N_1	3	18	94908	15732	29276	1.10	0.18	0.28	17	11	11	49	3565	113
N_2	4	20	391813	102909	149431	5.49	1.42	1.71	36	27	22	52	18180	130
N_3	4	20	428812	131202	166041	6.34	2.04	1.94	35	30	21	52	20021	160
N_4	5	22	1.76e+6	551091	734171	34.13	11.73	10.89	111	115	74	55	90467	147
F_5^A	6	6	658	271	658	0.00	0.00	0.00	7	7	7	8	218	8
F_{10}^A	11	11	13623	271	13623	0.12	0.00	0.13	24	8	24	8	218	8
F_{15}^A	16	16	109773	271	109773	1.61	0.00	1.78	309	9	309	8	218	8
F_5^B	5	6	78	496	9	0.00	0.00	0.00	7	7	7	6	79	6
F_{10}^B	10	11	523	6.73e+6	9	0.00	94.81	0.00	8	3254	7	6	27107	6
F_{15}^B	15	16	1718	–	9	0.03	–	0.00	17	–	7	6	–	6
A_2	8	0	359	27	334	0.00	0.01	0.00	7	7	7	12	22	12
A_3	16	0	61633	344	49652	0.13	0.01	0.18	13	7	14	17	169	17
A_4	32	0	–	38209	–	–	0.30	–	–	18	–	–	867	–
A_5	64	0	–	–	–	–	–	–	–	–	–	–	–	–

Overall, the concept of useless transitions has shown its potential in an impressive way. The results show a significant improvement of the error traces in comparison to greedy search as well as a significant reduction of the explored state space compared to A*. On many problems, the size of the explored state space is even lower than with greedy search. Our experimental evaluation has shown this effect on a large number of benchmarks, ranging from academic to industrial examples with instances of different difficulties, ranging from very easy to very hard. Some problems represent timed systems. We have seen that the overall performance of *UT* depends on the precision of the underlying distance estimation function. With *UT*, a sophisticated distance function like h^L already often leads to significant better guidance of the state space traversal than with greedy search and A*. More informative distance functions (like h^U) also lead to

better search guidance when applied with *UT*, and hence, the number of explored states further decreases. With less informative distance functions (like h^{gd}), the impact of *UT* decreases and the whole search process degenerates towards A*.

6 Conclusion

We have introduced the concept of useless transitions to directed model checking as an adaptation of the useless actions approach that has successfully been proposed in the area of AI Planning. Based on useless transitions, we have proposed a hybrid algorithm between A* and greedy search that seems to identify the sweet spot of the trade-off between scalability and short computed error traces. We have implemented this algorithm and evaluated it empirically on a number of benchmarks for a number of distance estimation functions. Our empirical evaluation shows a substantial performance gain in terms of explored states when compared with A*, and a significant solution quality improvement when compared with greedy search. Due to better guidance abilities, we often even explore less states than greedy search, being able to solve much larger problems than A* and greedy search.

As outlined in the discussion section, our approach seems to be currently best suited for distance estimation functions that are computed on-the-fly, and less suited for distance functions based on pattern databases. This is because the time overhead seems to be too large when adapting it for such functions in a straight forward way. To investigate how to adapt our concept *efficiently* to distance functions based on pattern databases will be an important topic for future research. Furthermore, it will be interesting to refine our concept to more than two degrees of uselessness. We expect that algorithms exploiting that knowledge further improve the state space traversal.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. 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] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

[2] Henning Dierks. Comparing model-checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.

[3] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *LNCS*, pages 19–34. Springer-Verlag, 2006.

[4] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2):247–267, 2004.

[5] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *LNCS*, pages 57–79. Springer-Verlag, 2001.

[6] Alex Groce and Willem Visser. Heuristics for model checking Java programs. *International Journal on Software Tools for Technology Transfer*, 6(4):260–276, 2004.

[7] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[8] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[9] Jörg Hoffmann, Jan-Georg Smaus, Andrey Rybalchenko, Sebastian Kupferschmid, and Andreas Podelski. Using predicate abstraction to generate heuristic functions in Uppaal. In *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, volume 4428 of *LNCS*, pages 51–66. Springer-Verlag, 2007.

[10] Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, and Alexander Baer. The UniForM workbench, a universal development environment for formal methods. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*, volume 1709 of *LNCS*, pages 1186–1205. Springer-Verlag, 1999.

[11] Sebastian Kupferschmid, Klaus Dräger, Jörg Hoffmann, Bernd Finkbeiner, Henning Dierks, Andreas Podelski, and Gerd Behrmann. UPPAAL/DMC – Abstraction-based heuristics for directed model checking. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *LNCS*, pages 679–682. Springer-Verlag, 2007.

[12] Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *LNCS*, pages 35–52. Springer-Verlag, 2006.

[13] Sebastian Kupferschmid, Jörg Hoffmann, and Kim G. Larsen. Fast directed model checking via russian doll abstraction. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 203–217. Springer-Verlag, 2008.

[14] Sebastian Kupferschmid, Martin Wehrle, Bernhard Nebel, and Andreas Podelski. Faster than UPPAAL? In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *LNCS*, pages 552–555. Springer-Verlag, 2008.

[15] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[16] Judea Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley, 1984.

[17] Kairong Qian and Albert Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 497–511. Springer-Verlag, 2004.

[18] Kairong Qian, Albert Nymeyer, and Steven Susanto. Abstraction-guided model checking using symbolic IDA* and heuristic synthesis. In *Proceedings of the 25th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, volume 3731 of *LNCS*, pages 275–289. Springer-Verlag, 2005.

[19] Charles L. Seitz. Ideas about arbiters. *Lambda*, 1:10–14, 1980.

[20] Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Useless actions are useful. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 388–395. AAAI Press, 2008.