Directed Model Checking with Distance-Preserving Abstractions

Klaus Dräger[1], Bernd Finkbeiner[1,2], Andreas Podelski[2]

[1] Universität des Saarlandes, Saarbrücken, Germany
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. In directed model checking, the traversal of the state space is guided by an estimate of the distance from the current state to the nearest error state. This paper presents a *distance-preserving abstraction* for concurrent systems that allows one to compute an interesting estimate of the error distance without hitting the state explosion problem. Our experiments show a dramatic reduction both in the number of states explored by the model checker and in the total runtime.

1 Introduction

The number of states of a concurrent system is exponential in the number of its components. This fundamental *state explosion* problem raises a complexity-theoretic barrier for all algorithmic methods based on state space traversal. As a consequence, it will always be interesting to investigate new approaches to circumvent the problem at least in particular situations. *Directed model checking* is one such approach that has received a lot of attention recently [4, 6, 12, 1, 2, 9, 19, 16]. The idea is to automatically compute an estimate of the *error distance*, which is the minimal number of steps between a given state and some error state. The state space traversal is then guided ("directed") by the estimate. In some situations, the benefit obtained from the guidance drastically outweighs the cost of the computation of the estimate; for success stories, we refer to [4, 6, 12, 1, 2, 9, 19, 16].

When we apply directed model checking to concurrent systems, the basic research question is: how can one compute an interesting estimate of the error distance without hitting the state explosion problem?

A natural idea is to compute an appropriate abstraction of the concurrent system and to base the estimate of the error distance between *concrete* states on the error distance between corresponding *abstract* states. We must make clear, however, what appropriate here means. We are not in a setting where the state space traversal is performed over abstract states and where the abstraction of a state aims at preserving the reachability vs. non-reachability of an error state. Instead, the state space traversal is performed over concrete states and the abstraction of a state aims at preserving the distance to an error state (we call it a "distance-preserving abstraction").

The contribution of this paper is a distance-preserving abstraction for concurrent systems that allows one to compute an interesting estimate of the error distance without hitting the state explosion problem. The definition of the abstraction originates from insights into the interplay between the impact of an action-based synchronization mechanism on the error distance in concurrent systems on the one hand and the use of estimated error distances during the state space traversal on the other hand.

We have implemented the directed model checking method with the distance-preserving abstraction. Our experiments indicate the usefulness of the estimate for a number of concurrent systems. We obtain a significant reduction both in the number of states explored and in the total running time, compared to directed model checking with an already existing estimate function that does not take into account synchronization.

2 Preliminaries

2.1 Notation

We verify safety properties over concurrent finite-state systems that are given as a finite set of processes $\mathcal{P}$. A *process* is a tuple $(\Sigma, Q, Q^0, Q^e, \rightarrow)$ where Σ is a finite *alphabet* of *observable actions*, Q is a finite set of states including the initial states $Q^0 \subseteq Q$ and error states $Q^e \subseteq Q$, and $\rightarrow \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ is a transition relation, where τ represents an unobservable internal action not in Σ. A transition $(p, a, p') \in \rightarrow$ is denoted by $p \xrightarrow{a} p'$.

An error occurs if all processes are in one of their error states Q^e. Often, one of the processes acts as the monitor for the safety property, in which case all other processes have the trivial error condition $Q^e = Q$.

The *error distance* $d_P(q) \in \mathbb{N} \cup \{\infty\}$ of a state q in a process P is the length of a shortest path from q to an error state (or ∞ if no such path exists).

We use a simple model of process synchronization where each observable action is shared by exactly two processes in $\mathcal{P}$. Consider a pair of processes $P_i = (\Sigma_i, Q_i, Q_i^0, Q_i^e, \rightarrow_i),\ i = 1, 2$. The *parallel composition*

$$P_1 \| P_2 = (\Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, Q_1^0 \times Q_2^0, Q_1^e \times Q_2^e, \rightarrow)$$

synchronizes the two processes on their common action symbols $(\Sigma_1 \cap \Sigma_2)$:

$$(p, q) \xrightarrow{a} (p', q') \text{ iff } \begin{cases} p \xrightarrow{a}_1 p', q = q', \text{ and } a \in (\Sigma_1 \setminus \Sigma_2) \cup \{\tau\} \\ p = p', q \xrightarrow{a}_2 q', \text{ and } a \in (\Sigma_2 \setminus \Sigma_1) \cup \{\tau\} \\ p \xrightarrow{c}_1 p', q \xrightarrow{c}_2 q' \text{ for some } c \in \Sigma_1 \cap \Sigma_2, \text{ and } a = \tau. \end{cases}$$

Since parallel composition is associative and commutative, we do not distinguish systems that are composed from the same set of processes by parallel composition in different orders. We denote the parallel composition of a set of processes $\mathcal{P} = \{P_1, \ldots, P_k\}$ by $\|_{P \in \mathcal{P}} P = P_1 \| \ldots \| P_k$.

```
Algorithm: ExpandingSearch
Input   : Initial Node $q_0$ of directed graph $G$
Output: true if a goal node is reachable from $q_0$, false otherwise
/* Initialization */
Open := $(s)$;
Closed := $()$;
while $Open \neq ()$ do
    $q := Open.pop()$;
    if $goal(q)$ then return true;
    $Closed.insert(q)$;
    foreach successor $q'$ of $q$ do
        if $q'$ not in Open or Closed then
            $Open.insert(q')$;
    end
end
return false;
```

Fig. 1. Algorithm ExpandingSearch decides reachability of a goal node from the initial node of a directed graph, using lists Open and Closed.

2.2 Directed Model Checking

Model checking can be implemented as an instance of the expanding search algorithm for directed graphs, shown in Figure 1. The algorithm maintains an *open list* of visited but not yet expanded states and a *closed list* of states that have been expanded. In each step, a state is chosen from the open list, expanded (i.e. all its successors that were not yet visited get added to the open list), and moved to the closed list. Organizing the open list as a FIFO queue results in a breadth-first traversal of the state space, while a LIFO stack results in a depth-first traversal.

In *directed* model checking [4], the open list is organized as a priority queue ordered by a function $h(q)$, which indicates the desirability of exploring a state q, usually based on an estimate $f(q)$ of $d_P(q)$. The best-known directed traversal algorithms are best-first traversal, where $h(q) = f(q)$, and A*, where $h(q)$ is the sum of $f(q)$ and the length of the shortest (currently known) path from an initial state to q. The advantage of A* is that it finds *shortest* error traces if the estimate function is *admissible*, which means it never overestimates $d_P(q)$. Typically, best-first traversal is faster than A*.

An even stronger property than admissibility is *consistency.* An estimate function f is *consistent* if, for every state q and every successor q' of q, $f(q) \leq f(q') + 1$. Consistent estimate functions improve the performance of the A* algorithm, because it is never necessary to *reopen* states. In general, a state q has to be put back on the open list if it is encountered again on a shorter path from the initial state. If the estimate function is consistent, we always find the shortest path first. Every consistent estimate is also admissible [15].

Our estimate function is based on an abstraction of the system. We define the abstraction of a process as the quotient with respect to an equivalence relation on the states. The *quotient* of a process $P = (\Sigma, Q, Q^0, Q^e, \rightarrow)$ *with respect to* an equivalence relation $\sim\ \subseteq Q \times Q$ is the process $P/{\sim} = (\Sigma, Q/{\sim}, Q^0/{\sim}, \{[q^e]_\sim \mid q^e \in Q^e\}, \Rightarrow)$, with

$$[p]_\sim \overset{a}{\Rightarrow} [q]_\sim \text{ iff } p' \xrightarrow{a} q' \text{ for some } p' \sim p, q' \sim q,$$

where $[q]_\sim$ denotes the equivalence class of a state $q \in Q$ with respect to $\sim$, and $Q/{\sim} = \{[q]_\sim \mid q \in Q\}$ denotes the quotient set. Every abstraction $P/{\sim}$ induces a consistent estimate function $f(q) = d_{P/\sim}([q]_\sim)$ of $d_P(q)$ [15].

3 Computing the Abstract System

Our estimate function is based on an abstraction of the system, which we compute in a preprocessing step before the model checking begins. To avoid constructing the full state space of the parallel product of all processes, we compute the abstraction incrementally: each composition of two processes is directly followed by an abstraction step.

Algorithm AbstractSystem, shown in Figure 2, describes this "compose-and-abstract" loop. For now, we ignore the question how the abstraction of a process is computed (we discuss algorithm AbstractProcess in Section 4) as well as the question in which order the processes are composed: algorithm AbstractSystem is parameterized by the *composition strategy*, a function S that selects a pair of two different processes from a set of processes. We discuss the composition strategy in Section 5.

Algorithm AbstractSystem maintains a set of processes $\mathcal{P}'$, which is initially equal to the given set of processes $\mathcal{P}$ and is eventually reduced to the singleton set $\{A\}$, where the process A represents the abstract system. Associated with each process P' in $\mathcal{P}'$ is the function $\alpha_{P'} : \prod_{P \in \mathcal{P}} Q_P \rightarrow (Q_{P'} \cup \{\perp\})$, which maps each concrete state q either to its abstraction in process P' or to $\perp$. The result $\alpha_{P'}(q) = \perp$ indicates that q is *irrelevant*, i.e., either q is not reachable from the initial states or the error states are not reachable from q. For the processes in $\mathcal{P}$, α_P is initialized with the projection to the respective component of the product states.

In each iteration of the "compose-and-abstract" loop, two processes P and P' are selected from the current set $\mathcal{P}'$ by the composition strategy S. Their parallel composition $P \| P'$ is first computed explicitly and then immediately abstracted by AbstractProcess to process C. In the new process set $\mathcal{P}'$, process C replaces P and P'. Associated with C is the new mapping α_C, which combines the mapping from the states of $P \| P'$ to the states of C (which is provided by AbstractProcess) with the mappings associated with P and P'.

The results of AbstractSystem are the abstract process A and the function α, which maps concrete states to abstract states or $\perp$. From these we derive the

```
Algorithm: AbstractSystem
Input   : concrete system, given as a finite set of processes $\mathcal{P} = \{P_1, \dots, P_n\}$
Output: • abstract system, given as process $A$
        • mapping from concrete to abstract states:
          $\alpha : \prod_{P \in \mathcal{P}} Q_P \to (Q_A \cup \{\perp\})$

/* Initialization */
$\mathcal{P}' := \mathcal{P}$;
for $i = 1, \dots, n$ do $\alpha_{P_i}(q_1, \dots, q_n) = q_i$;

/* "Compose-and-abstract" loop */
while $|\mathcal{P}'| > 1$ do
    $(P, P') := S(\mathcal{P}')$;
    $(C, \gamma) :=$ AbstractProcess$(P \| P')$;
    $\mathcal{P}' := \mathcal{P}' \cup \{C\} \setminus \{P, P'\}$;
    $\alpha_C(q) := \begin{cases} \perp & \text{if } \alpha_P(q) = \perp \text{ or } \alpha_{P'}(q) = \perp \\ \gamma(\alpha_P(q), \alpha_{P'}(q)) & \text{otherwise;} \end{cases}$
end

$A :=$ the remaining member of $\mathcal{P}'$;
return $A$, $\alpha_A$;
```

Fig. 2. Algorithm AbstractSystem computes an abstract system for a given concrete system.

estimate function

$$f(q) = \begin{cases} \infty & \text{if } \alpha(q) = \perp \\ d_A(\alpha(q)) & \text{otherwise.} \end{cases}$$

Since the mapping α induces an equivalence on the states of the concrete system ($p \sim q \Leftrightarrow \alpha(p) = \alpha(q)$), this estimate function is consistent for any choice of a process abstraction and composition strategy.

4 Computing Abstract Processes

How can we ensure that the error distance of the *abstract* state provides a good estimate for the error distance of the *concrete* state? A natural idea is to use one state in the abstraction as a representative for each set of concrete states with the same error distance. While this preserves the error distance in the immediate abstraction, it changes the *synchronization behavior* of the process. This, in turn, changes the error distance in the next iteration of the "compose-and-abstract" loop, when the abstracted process is composed with some other process. The straightforward solution of this problem, to identify only bisimilar states and thus preserve the synchronization behavior of the process, generally does not sufficiently reduce the state space.

Our approach draws from both ideas. We fix a bound N on the maximal number of states in the abstraction. Within this bound, our first priority is

```
Algorithm: AbstractProcess
Input  : concrete process $P = (\Sigma, Q, Q^0, Q^e, \rightarrow)$
Output: • abstract process $A$,
        • mapping from concrete to abstract states:
          $\alpha : Q \rightarrow (Q_A \cup \{\perp\})$

/* Initialization */
$Q' := \{q \in Q \mid d_P(q) < \infty$ and $q$ reachable from $Q^0\}$;
$P' := (\Sigma, Q', Q^0 \cap Q', Q^e \cap Q', \rightarrow \cap (Q' \times (\Sigma \cup \{\tau\}) \times Q'))$;
$\sim := \{(q, q') \in Q' \times Q' \mid \min(d_P(q), N-1) = \min(d_P(q'), N-1)\}$;
$K := |Q'/{\sim}|$;
for $i = 0, \ldots, K-1$ do
    $B_i := \{q \in Q' \mid \min(d_P(q), N-1) = i\}$;
    $R_i := \sim \cap (B_i \times B_i)$ ;
end

/* Refinement loop */
repeat
    $\sim' := \sim$;
    for $i = 0, \ldots, K-1$ do
        $R_i^* := \{(q, q') \in R_i \mid \forall a \; \{[r]_\sim \mid q \xrightarrow{a} r\} = \{[r']_\sim \mid q' \xrightarrow{a} r'\}\}$;
        if $|Q'/(R_1 \cup \cdots \cup R_i^* \cup \cdots \cup R_K)| \leq N$ then
            $R_i := R_i^*$;
            $\sim := \bigcup_{i=0}^{K-1} R_i$;
    end
until $\sim = \sim'$ ;

$A := P'/{\sim}$;
$\alpha(q) := \begin{cases} [q]_\sim & \text{if } q \in Q' \\ \perp & \text{otherwise;} \end{cases}$
return $A, \alpha$;
```

Fig. 3. Algorithm AbstractProcess computes an abstract process for a given concrete process.

to ensure that only states with the same error distance are identified, and our second priority is to preserve the synchronization behavior.

Algorithm AbstractProcess is shown in Figure 3. As part of the initialization, AbstractProcess prunes irrelevant states. Process P' contains only states that are both reachable and have paths to some error state. The computation of the equivalence relation $\sim$ starts with the equivalence that identifies two states iff they have the same error distance. During the entire run of the algorithm, we only consider refinements of this equivalence. We therefore partition the states into buckets $B_0, \ldots, B_{N-1}$ according to their error distance and consider a separate equivalence relation $R_i = \sim \cap (B_i \times B_i), i = 1, \ldots, N-1$, on each bucket.

The subsequent loop refines $\sim$ until a fixpoint is reached. For each relation R_i, we tentatively split the equivalence classes in R_i according to the equivalence classes of their successors in $\sim$. If the refined equivalence R_i^* does not increase the total number of equivalence classes beyond the bound N, we refine $\sim$ according to R_i^*. The buckets are considered in the order of increasing error distance, starting with B_0. This choice is based on the intuition that paths from states with high error distance traverse states with lower error distance on their way to the error state. Inaccuracies introduced for states with high error distance are therefore likely to affect fewer states than inaccuracies introduced for states with low error distance.

When the fixpoint is reached (after at most N iterations of the refinement loop), the abstraction is computed as the quotient $P'/\sim$. The function α maps each relevant concrete state q to its equivalence class $[q]_\sim$.

Experiments. To evaluate this approach experimentally, we compare AbstractProcess to an alternative solution that considers buckets with high error distance first. The advantage of AbstractProcess is especially clear in systems with long error paths, such as the Towers of Hanoi example described in Section 6. Figure 4 is based on data from the Towers of Hanoi benchmark with three disks. The graph shows the average difference between estimated and actual error distance over all states with the same actual error distance in percent of the actual error distance. The estimate obtained with AbstractProcess is significantly more accurate than the estimate obtained by considering buckets with high error distance first. Both estimate functions have an area around the error states with perfect precision, but the area of the estimate obtained with AbstractProcess is twice as large, resulting in a perfectly informed estimate at error distance 9, where the alternative solution already reaches its peak imprecision of 57%.

5 The Composition Strategy

Algorithm AbstractProcess is guaranteed to preserve the error distance in the immediate abstraction, but may cause changes to the error distance once the abstract process is composed with further processes. The goal of the composition strategy is to minimize the resulting inaccuracy by choosing a pair of processes such that the error distance in their parallel composition provides a good estimate of the error distance in the completely composed system.

A first observation is that in processes with trivial error condition $Q^e = Q$, the local error distance is 0 for all states. We therefore only consider pairs of processes where at least one process has a non-trivial error condition. Among these, we choose a pair such that their joint actions occur close to error states. The result of this strategy is that we build an area close to the error states where no synchronization is necessary to reach the error. Within this area, the local error distance accurately reflects the error distance in the completely composed system.

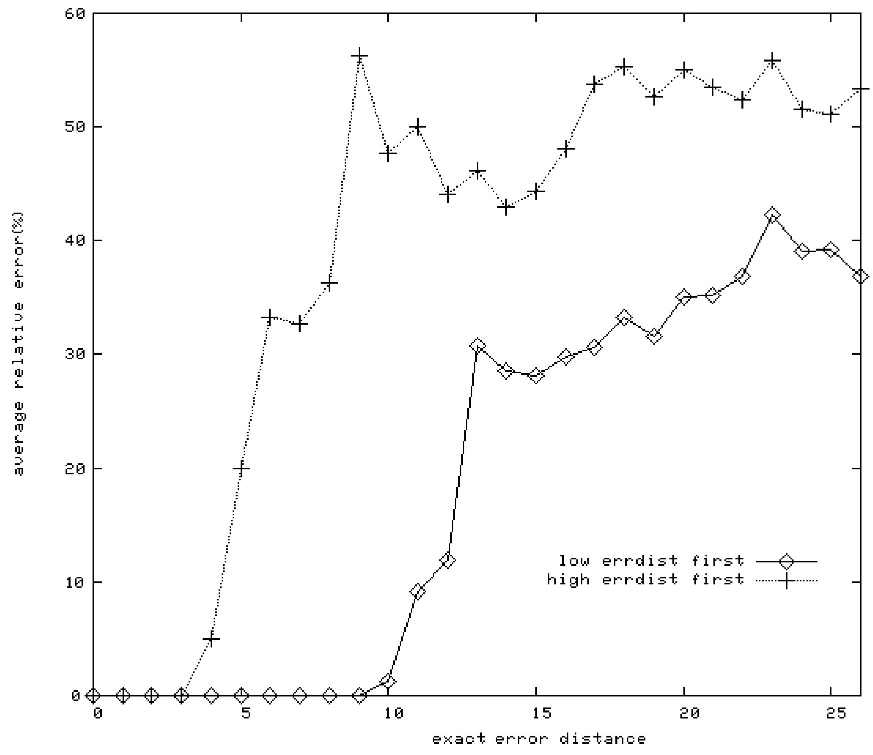

Fig. 4. Comparison of algorithm AbstractProcess with an alternative solution that considers buckets with high error distance first. The graph shows the average difference between estimated and actual error distance over all states with the same actual error distance in percent of the actual error distance. (Data from the Towers of Hanoi benchmark with three disks and a bound of 40 states.)

To implement this strategy, we introduce a ranking on the actions

$$r(P, a) = \min\{d_P(q) \mid q \in Q, \exists q' \in Q : q' \xrightarrow{a} q\}.$$

A low ranking indicates that the action may be taken in close proximity of the error. We associate with each pair (P_1, P_2) of two different processes the weight

$$\min\{\max\{r(P_1, a), r(P_2, a)\} \mid a \in \Sigma_1 \cap \Sigma_2\}$$

and choose a pair of processes that minimizes this weight.

Experiments. We compare the described ranking-based strategy with the default strategy that composes processes in the order in which they are defined. The advantage of the ranking-based strategy is especially clear in systems where only few processes have a non-trivial error condition. Figure 5 is based on data from the Arbiter Tree benchmark (see Section 6) with eight processes, where

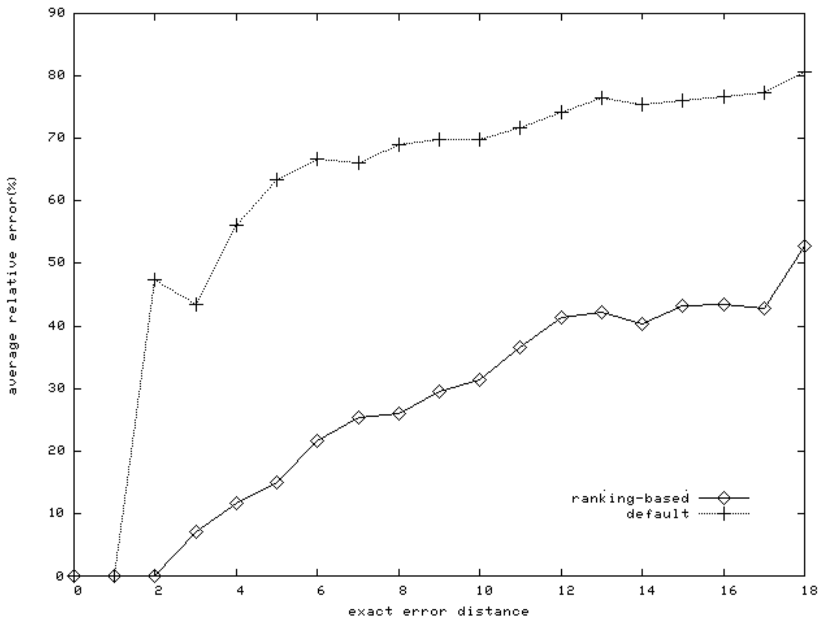

Fig. 5. Comparison of the ranking-based composition strategy with the default strategy, which composes processes in the order in which they are defined. The graph shows the average difference between estimated and actual error distance over all states with the same actual error distance in percent of the actual error distance. (Data from the Arbiter Tree benchmark with eight processes and a bound of 20 states.)

only two out of the eight processes have non-trivial error conditions. The graph shows the average difference between estimated and actual error distance over all states with the same actual error distance in percent of the actual error distance. The ranking-based strategy results in an estimate function that is roughly twice as accurate as the estimate function resulting from the default strategy.

6 Experiments

Our collection of benchmarks contains standard examples for distributed systems (Arbiter Tree, Towers of Hanoi), randomly generated systems, and industrial case studies. We have implemented our algorithms in an experimental version of the model checker UPPAAL [13].

We evaluate our estimate function both for best-first traversal (Table 1) and for A* (Table 2). For each benchmark, the tables show the running time, the number of explored states, and the length of the discovered error trace. We compare our estimate function with two different bounds ($N50$ and $N100$) to randomized depth-first traversal (rDF) and directed model checking with the FSM estimate function [6] (FSM).

Table 1. Experimental Results: Comparison of best-first traversal using our estimate function for two different bounds ($N50$ and $N100$) to best-first traversal using the FSM estimate function (FSM) and to randomized depth-first traversal (rDF).

	explored states				seconds				trace length			
Exp	rDF	FSM	$N50$	$N100$	rDF	FSM	$N50$	$N100$	rDF	FSM	$N50$	$N100$
A2	85	54	53	46	0.01	0.01	0.06	0.10	46	45	37	25
A3	6878	420	174	187	0.05	0.05	0.24	0.56	323	183	79	43
A4	1994	1.3e5	1.5e5	10633	0.06	1.01	3.16	2.78	429	1003	509	157
A5	***	9.9e5	7619	10673	1198	12.48	5.66	26.73	***	5213	3869	1151
A6	*	**	4.3e5	5.2e5	**	**	62.30	196.9	*	**	2.0e5	55535
H4	3027	4996	1283	711	0.03	0.05	0.07	0.09	573	761	181	125
H5	52417	57600	6497	6368	0.24	0.24	0.13	0.18	5528	3705	381	405
H6	3.1e5	5.0e5	1.1e5	63403	1.39	1.92	0.65	0.53	31225	26605	1445	1317
H7	1.5e6	4.5e6	7.4e5	7.5e5	7.50	20.37	4.09	4.32	2.3e5	2.0e5	3377	3177
H8	2.9e7	1.6e7	8.6e6	4.5e6	336.2	132.3	60.61	29.34	1.8e6	1.5e6	12073	6705
R5	5840	4177	697	443	0.04	0.04	0.05	0.06	936	154	62	64
R6	71098	19903	395	363	0.32	0.11	0.07	0.10	858	97	43	41
R7	3.1e5	83582	6656	8199	1.42	0.32	0.12	0.17	1040	81	56	50
R8	1.5e6	2.7e5	2.2e5	1.2e5	9.13	1.01	1.32	0.87	1453	138	58	59
R9	***	***	2.9e5	4.9e5	336.3	80.43	2.05	3.64	***	***	77	80
R10	***	***	***	2.6e5	496.3	71.83	38.87	2.20	***	***	***	122
M1	23894	31927	19063	12780	0.54	0.45	0.35	0.23	926	1349	129	74
M2	1.6e5	2.0e5	46545	46337	2.19	2.92	0.74	0.86	3717	7695	131	190
M3	68313	1.7e5	64522	42414	0.92	2.34	0.99	0.80	3589	5690	119	92
M4	2.0e5	5.8e5	1.7e5	1.3e5	2.71	7.34	2.49	1.86	14415	25819	146	105
N1	43655	42931	27275	1660	1.56	1.62	1.02	0.15	985	1803	187	194
N2	1.7e5	2.6e5	1.0e5	67168	5.61	9.43	3.55	2.16	4611	9279	218	138
N3	1.7e5	1.3e5	1.4e5	81804	5.85	4.96	4.99	2.69	3794	11656	178	130
N4	1.0e6	1.5e6	4.8e5	3.8e5	34.71	51.10	17.91	11.07	17851	41986	234	169
C1	25122	19263	871	810	0.24	0.24	0.30	0.49	1087	1442	188	191
C2	65275	68070	1600	2620	0.56	0.59	0.40	1.03	886	2032	203	206
C3	86439	97733	2481	2760	0.74	0.82	0.47	1.14	786	1663	204	198
C4	8.5e5	9.8e5	22223	25206	6.52	6.90	0.91	1.83	1680	5419	247	297
C5	8.3e6	8.8e6	1.6e5	1.6e5	66.41	66.85	2.90	3.97	1900	14163	322	350
C6	***	**	1.7e6	1.2e6	1181	**	18.32	14.87	***	**	480	404
C7	*	**	1.3e7	1.3e7	*	**	156.1	162.4	*	**	913	672
C8	*	**	1.4e7	1.2e7	*	**	163.0	155.3	*	**	1305	2210
C9	*	**	**	3.6e7	*	**	**	1046	*	**	**	1020

* timeout; ** out of memory; *** timeout on some instances

Our experiments were carried out on an Intel Xeon 3.06 Ghz system with 4 GByte of RAM. For all experiments, we set a time limit of 30 minutes. In the case of rDF, the table shows the average runtime over three runs. For some benchmarks, some but not all of these runs hit our time limit. These runs were added into the runtime average with the 30-minute timeout as their runtime.

Table 2. Experimental results: Comparison of A* traversal using our estimate function for two different bounds ($N50$ and $N100$) to A* traversal using the FSM estimate function (FSM).

	explored states			seconds			trace
Exp	FSM	$N50$	$N100$	FSM	$N50$	$N100$	length
A2	498	215	46	0.02	0.06	0.10	25
A3	81883	32106	20658	0.41	0.48	0.73	35
H4	6289	3876	3348	0.06	0.08	0.10	105
H5	67202	52348	48361	0.29	0.32	0.36	229
H6	627669	540286	516242	2.46	2.80	2.82	481
H7	5.8e6	5.4e6	5.3e6	27.08	32.29	31.48	989
R5	35784	4642	2392	0.15	0.06	0.08	27
R6	174589	6047	4295	0.69	0.07	0.12	22
R7	764727	14037	12083	3.30	0.16	0.20	27
R8	2.1e6	98420	60322	12.94	0.67	0.52	23
R9	**	93806	70578	125.95	0.71	0.69	25
R10	**	271935	279693	88.46	2.22	2.47	25
M1	50147	25103	23917	0.79	0.52	0.48	50
M2	223034	100513	94426	3.30	1.82	1.82	51
M3	231357	130747	129269	3.42	2.43	2.51	53
M4	971736	561599	516178	13.99	10.57	9.54	54
N1	99840	56550	52564	5.59	3.44	3.03	50
N2	446465	238369	218351	25.30	14.86	13.21	53
N3	473117	286506	257530	27.04	17.86	15.23	53
N4	2.0e6	1.2e6	1.1e6	117.43	74.83	70.88	56
C1	35768	13863	13455	0.37	0.42	0.62	55
C2	110593	38483	36888	0.99	0.76	1.37	55
C3	144199	44730	42366	1.27	0.91	1.54	55
C4	1.4e6	368813	354091	11.23	4.30	5.05	56
C5	1.3e7	2.8e6	2.7e6	116.28	29.60	29.97	57
C6	*	2.8e7	2.7e7	*	377.77	364.15	57

* (**) out of memory (on some instances)

Arbiter Tree. The Arbiter Tree [18] establishes mutual exclusion between 2^k client processes. The processes are arranged in a binary tree of height k, where each leaf node is a client and each internal node is an arbiter that ensures mutual exclusion between its two children, passes requests and releases upward, and passes grants downward. One additional process handles the requests of the root node by immediately sending a grant upon receiving a request and then waiting for the release. The benchmarks A2 – A6 contain arbiter trees of height 2 – 6, with an exponentially growing number of processes (A2 has 8 processes, A6 has 128). We specified mutual exclusion for one particular pair of client processes and introduced a fault in the form of an incorrect client that erroneously sends *several* release signals when done.

The error in a tree with 128 processes is found in approx. 1 minute using a bound of 50 states. Because not all processes contribute to reaching an error state, this low bound already produces a well-informed heuristic. Using the higher bound of 100 states is expensive: since in this benchmark the length of the shortest error path is only linear in the height of the tree, computing the estimate involves composing a large number of processes with few and therefore large buckets. The more accurate estimate produced by $N100$ does, however, lead to shorter error traces.

The Towers of Hanoi. Benchmarks H4 – H8 model the standard problem of moving a stack of differently sized disks from one of three columns to another, with the constraints that the disks may only be moved one at a time and a disk may never be stacked on top of a smaller disk. We modeled the problem with one process for each disk. A disk can at any time send a request upwards in the hierarchy of smaller disks to check whether itself and a target column is clear of smaller disks. If it gets an "ok" signal, it moves from its current column to the target column. To find a trace that leads to the target configuration we specify the target configuration as the error condition. In this benchmark, the length of the shortest error path grows exponentially with the number of processes. This explains why the bound $N100$ performs significantly better than the bound $N50$ in the largest benchmark H8.

Randomly generated systems. We obtained a further suite of benchmarks by randomly generating systems of processes. The parameters of the construction are the number of processes, the minimum and maximum number of states of the processes, and the seed for the random number generator (the Mersenne Twister [14]). Excluded from the benchmarks are systems with no error paths and systems that contain independent subsystems, i.e., systems where the process graph, with edges between processes that have shared actions, is not connected.

Benchmarks R5 – R10 each consist of 15 different randomly generated systems, with the size ranging from 5 (R5) to 10 (R10) processes. We set the number of actions to twice the number of processes, the minimum/maximum size to 3/10, and averaged the results over the 15 systems for each size. The only method besides our estimate function that also finds the error in all systems with 10 processes is rDF, which, however, takes significantly more time.

A* is usually much more expensive than best-first traversal. In this benchmark, however, A* results in a much more focused traversal, as the number of visited states shows. As a result, A* even becomes faster than best-first traversal.

Industrial Examples. Henning Dierks provided us with a collection of UPPAAL benchmarks from two industrial case studies: A real-time mutual exclusion protocol in a distributed system with asynchronous communication [3] (benchmarks M1 – M4 and N1 – N4) and a tramway controller from the UniForM project [10] (C1 – C9). The two case studies add real-time constraints and integer variables to the discrete setting of the other benchmarks: the faults in both case studies are introduced as erroneous time bounds. Even though our implementation is not yet optimized for this type of system (in the computation of the estimate, we simply ignore the clocks and use a flat representation of the integer values as discrete states), the directed model checker performs remarkably well, solving several benchmarks that were previously out of UPPAAL's reach.

7 Related Work

Several researchers have investigated techniques to guide the model checker. Typically, the guidance is application-specific and must be provided by the user. For example, Behrmann et al [1] describe UPPAAL case studies in which a dramatic reduction of the state space was achieved by a user-provided estimate of the error distance. Bloem et al [2] use *hints* in the form of assertions on the primary inputs and state variables of the model: the transition relation can then be underapproximated (by ignoring transitions out of states that violate the hint) or overapproximated (by allowing any transition from a state that violates the hint). Similarly, Kaltenbach and Misra [9] use hints in the form of *regular expressions* over the actions of the program.

Directed model checking with an automatically computed estimate of the error distance has been pioneered by Edelkamp, Leue, and Lluch-Lafuente with the tool HSF-SPIN [**?**]. In addition to several simpler heuristics for safety and liveness properties (including deadlock-detection), HSF-SPIN implements the FSM heuristic [6]. The FSM heuristic approximates the error distance by the maximum (or, alternatively, the sum) of the error distances in individual processes and is a significant improvement over program-independent estimates like the Hamming-distance [19]. The drawback of the FSM heuristic is that it ignores the synchronization between the processes. It is therefore less useful when searching for errors that require a complex interaction between multiple processes.

Similar to our approach, the *pattern databases* of Qian and Nymeyer [16] and the *abstraction databases* by Edelkamp and Lluch-Lafuente [5] also make use of an abstraction of the system. The error distances in the abstract state space are stored in a table, from which they are read off during the traversal of the concrete state space. Our abstraction technique extends these methods: while both pattern databases and abstraction databases assume that a particular abstraction function is chosen beforehand, we automatically compute an abstraction function that aims at preserving the error distance.

Related to our incremental abstraction technique is the *Incremental Composition and Reduction (ICR) Method* [17], which reduces the partially composed system after each composition of two processes to an observationally equivalent process. Since ICR maintains an accurate representation of the behavior of the partially composed system (which often requires more states than the completely composed system), ICR is only feasible if the user provides additional constraints on the process interaction [7]. By contrast, our method, which only maintains an approximate representation of the behavior, is fully automatic.

In very recent work, Kupferschmid et al [11] investigate using an estimate function from AI planning for directed model checking. The estimate is based on a relaxation of the system in which every state variable, once it has obtained a value, keeps that value forever. Because Kupferschmid et al's estimate function is computed on-the-fly, it can be used in systems with infinite data types (such as unbounded integers), which are currently out of our scope. On the other hand, our precomputed abstraction reflects the process synchronization more accurately, which leads to much better performance in systems with complex process interaction, such as the Towers of Hanoi benchmark (see Section 6). There is obvious potential in a combination of the two approaches, which we plan to explore in future work.

An important complement to directed model checking with estimates of the error distance are structural heuristics as implemented in the Java PathFinder [8]. These heuristics exploit the program structure for example by maximizing thread interleavings and code coverage.

8 Conclusion

Abstraction has always been considered a key in fighting the state explosion problem. Here, we have given a new twist to abstraction. We traverse abstract states in order to compute an estimate of the error distance, and then traverse concrete states in order to find an error path. The quality of an abstraction is not determined by a Boolean value ("does the abstraction preserve the reachability of an error state by the initial state?"). It is rather determined by the ratio between the estimated and the actual error distance.

While we are still in the beginning of the systematic design of such abstractions, this paper has made an initial contribution. It presents a distance-preserving abstraction for concurrent systems that allows one to compute an interesting estimate of the error distance without hitting the state explosion problem. As detailed in the paper, the definition of the abstraction originates from insights into the interplay between the impact of an action-based synchronization mechanism on the error distance in concurrent systems on the one hand and the use of estimated error distances during the state space traversal on the other hand.

We have implemented the resulting directed model checking method, and we have led a series of experiments that indicate the usefulness of an estimate that takes into account synchronization.

With abstraction, one always encounters a tradeoff between cost and precision. A potential advantage of our abstraction method is that it is parameterized (by the size of the abstract state space), and that one can fine-tune the parameter (and thus the accuracy of the abstraction). To demonstrate the tradeoff on an example, we took a randomly generated system with eight processes and changed the parameter gradually. Figure 6 shows the corresponding running times. Initially, the runtime decreases with a increasing parameter. After the sweet spot in the tradeoff is reached (in the region between 60 and 80), the runtime increases with increasing parameter. More experience is needed in order to provide systematic ways to choose the parameter.

Acknowledgements We thank Gerd Behrmann and Henning Dierks for fruitful discussions on heuristics in UPPAAL. Gerd Behrmann helped us with the UP-

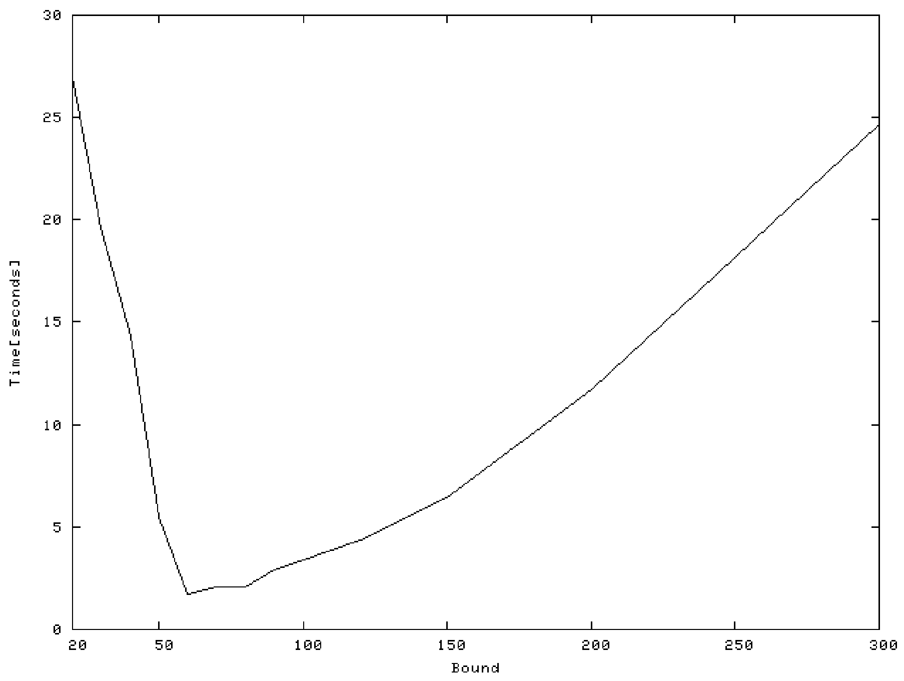

Fig. 6. Running time of the directed model checker for different bounds on the abstract state space. (Data from a randomly generated system with eight processes.)

PAAL source code and Henning Dierks provided us with interesting UPPAAL benchmarks from industrial case studies.

References

1. G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient guiding towards cost-optimality in uppaal. In T. Margaria and W. Yi, editors, *Proceedings of TACAS'01*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer-Verlag, 2001.
2. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Design Automation Conference*, pages 29–34, 2000.
3. H. Dierks. Comparing model checking and logical reasoning for real-time systems. *Formal Aspects of Computing*, 16(2):104–120, 2004.
4. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Software Tools for Technology Transfer*, 2003.
5. S. Edelkamp and A. Lluch-Lafuente. Abstraction databases in theory and model checking. In *Proc. ICAPS Workshop on Connecting Planning Theory with Practice, June 2004.*
6. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Trail-directed model checking. In *Proc. of the Workshop on Software Model Checking*, Electrical Notes in Theoretical Computer Science. Elsevier, July 2001.

7. S. Graf, B. Steffen, and G. Lüttgen. Compositional minimization of finite state systems using interface specifications. *Formal Aspects of Computation*, 8, September 1996.
8. A. Groce and W. Visser. Heuristics for model checking Java programs. In *SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science 2318, pages 242–245. Springer, 2002.
9. M. Kaltenbach and J. Misra. A theory of hints in model checking. In B. K. Aichernig and T. Maibaum, editors, *Formal Methods at the Crossroads: From Panacea to Foundational Support*, number 2757 in Lecture Notes in Computer Science, pages 423–438. Springer-Verlag, 2003.
10. B. Krieg-Brückner, J. Peleska, E.-R. Olderog, and A. Baer. The UniForM workbench, a universal development environment for formal methods. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*, number 1709 in Lecture Notes in Computer Science, 1999.
11. S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Boosting UPPAAL by an AI planning heuristic. Submitted to SPIN 2006.
12. A. L. Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, Institute of Computer Science, University of Freiburg, June 2003.
13. K. Larsen, P. Petterson, and Wang Yi. Uppaal in a nutshell. *STTT – International Journal on Software Tools for Technology Transfer*, 1(1+2):134–152, Dec. 1997.
14. M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.
15. J. Pearl. *Heuristics*. Morgan Kaufmann, San Francisco, CA, 1983.
16. K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In K. Jensen and A. Podelski, editors, *Proceedings of TACAS'04*, number 2988 in Lecture Notes in Computer Science, pages 497–511, 2004.
17. K. K. Sabnani, A. M. Lapone, and M. Ü. Uyar. An algorithmic procedure for checking safety properties of protocols. *IEEE Trans. Commun.*, 37(9):940–948, September 1989.
18. C. Seitz. Ideas about arbiters. *Lambda*, pages 10–14, 1980.
19. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Design Automation Conference*, pages 599–604, 1998.