Multi-core Reachability for Timed Automata

Andreas E. Dalsgaard[2], Alfons Laarman[1], Kim G. Larsen[2],
Mads Chr. Olesen[2], and Jaco van de Pol[1]

[1] Formal Methods and Tools, University of Twente
{a.w.laarman,vdpol}@cs.utwente.nl
[2] Department of Computer Science, Aalborg University
{andrease,kgl,mchro}@cs.aau.dk

Abstract. Model checking of timed automata is a widely used technique. But in order to take advantage of modern hardware, the algorithms need to be parallelized. We present a multi-core reachability algorithm for the more general class of well-structured transition systems, and an implementation for timed automata.

Our implementation extends the opaal tool to generate a timed automaton successor generator in C++, that is efficient enough to compete with the UPPAAL model checker, and can be used by the discrete model checker LTSMIN, whose parallel reachability algorithms are now extended to handle subsumption of semi-symbolic states. The reuse of efficient lockless data structures guarantees high scalability and efficient memory use.

With experiments we show that opaal+LTSMIN can outperform the current state-of-the-art, UPPAAL. The added parallelism is shown to reduce verification times from minutes to mere seconds with speedups of up to 40 on a 48-core machine. Finally, strict BFS and (surprisingly) parallel DFS search order are shown to reduce the state count, and improve speedups.

1 Introduction

In industries developing safety-critical real-time systems, a number of safety requirements must be fulfilled. Model checking is a well-known method to achieve this and is critical for ensuring correct behaviour along all paths of execution of a system. One popular formalism for real-time systems is timed automata [3], where the time is modelled as a number of resettable clocks. Good tool support for timed automata exists [9].

However, as the desire to model check ever larger and more complex models arises, there is a need for more effective techniques. One option for handling large models has always been to buy a bigger machine. This provided great improvements; while early model checkers handled thousands of states, now we can handle billions. However, in recent years processor speed has stopped increasing, and instead more cores are added. These cores cannot be taken advantage of by the normal sequential algorithms for model checking.

M. Jurdziński and D. Ničković (Eds.): FORMATS 2012, LNCS 7595, pp. 91–106, 2012.

The goal of this work is to develop scaling multi-core reachability for timed automata [3] as a first step towards full multi-core LTL model checking. A review of the history of discrete model checkers shows that indeed multi-core reachability is a crucial ingredient for efficient parallel LTL model checking (see Sec. 2). To attain our goal, we extended and combined several existing software tools:

LTSmin is a language-independent model checking framework, comprising, inter alia, an explicit-state multi-core backend [23,13].

opaal is a model checker designed for rapid prototype implementation of new model checking concepts. It supports a generalised form of timed automata [17], and uses the UPPAAL input format.

The UPPAAL DBM library is an efficient library for representing timed automata zones and operations thereon, used in the UPPAAL model checker [9].

Contributions: We describe a multi-core reachability algorithm for timed automata, which is generalizable to all models where a well-quasi-ordering on the behaviour of states exist [19]. The algorithm has been implemented for timed automata, and we report on the structure and performance of this prototype.

Before we move on to a description of our solution and its evaluation, we first review related work, and then briefly introduce the modelling formalism.

2 Related Work

One efficient model checker for timed automata is the UPPAAL tool [9,7]. Our work is closely related to UPPAAL in that we share the same input format and reuse its editor to create input models. In addition, we reused the open source UPPAAL DBM library for the internal symbolic representation of time zones.

Distributed model checking algorithms for timed automata were introduced in [11,6]. These algorithms exhibited almost linear scalability (50–90% efficiency) on a 14-node cluster of that time. However, analysis also shows that static partitioning used for distribution has some inherent limitations [15]. Furthermore, in the field of explicit-state model checking, the DiVinE tool showed that static partitioning can be reused in a shared-memory setting [5]. While the problem of parallelisation is considerably simpler in this setting, this tool nonetheless featured suboptimal performance with less than 40% efficiency on 16-core machines [22]. It was soon demonstrated that shared-memory systems are exploited better by combining local search stacks with a lockless hash table as shared passed set and an off-the-shelf load balancing algorithm for workload distribution [22]. Especially in recent experiments on newer 48-core machines [18, Sec. 5], the latter solution was clearly shown to have the edge with 50–90% efficiency.

Linear-time, on-the-fly liveness verification algorithms are based on depth-first search (DFS) order [20]. Next to the additional scalability, the shared hash table solution also provides more freedom for the search algorithm, which can be pseudo DFS and pseudo breadth-first search (BFS) order [22], but also strict BFS (see Sec. 6.2). This freedom has already been exploited by parallel NDFS algorithms for LTL model checking [20,18] that are linear in the size of the

input graph (unlike their BFS-based counterparts). While these algorithms are heuristic in nature, their scalability has been shown to be superior to their BFS-based counterparts.

3 Preliminaries

We will now define the general formalism of well-structured transition systems [19,1], and specifically networks of timed automata under the zone abstraction [16].

Definition 1 (Well-quasi-ordering). *A well-quasi-ordering $\sqsubseteq$ is a reflexive and transitive relation over a set X, s.t. for any infinite sequence $x_0, x_1, \ldots$ eventually for some $i < j$ it will hold that $x_i \sqsubseteq x_j$.*

In other words, in any infinite sequence eventually an element exists which is "larger" than some earlier element.

Definition 2 (Well-structured transition system). *A well-structured transition system is a 3-tuple $(S, \rightarrow, \sqsubseteq)$, where S is the set of states, $\rightarrow$: $S \times S$ is the (computable) transition relation and $\sqsubseteq$ is a well-quasi-ordering over S, s.t. if $s \rightarrow t$ then $\forall s'.s \sqsubseteq s'$ there $\exists t'.s' \rightarrow t' \wedge t \sqsubseteq t'$.*[1]

We thus require $\sqsubseteq$ to be a monotonic ordering on the behaviour of states, i.e., if $s \sqsubseteq t$ then t has at least the behaviour of s (and possibly more), and we say that t *subsumes* or *covers* s.

One instance of well-structured transition systems arise from the symbolic semantics of timed automata. Timed automata are finite state machines with a finite set of real-valued, resettable clocks. Transitions between states can be guarded by constraints on clocks, denoted $G(C)$.

Definition 3 (Timed automaton). *An extended timed automaton is a 7-tuple $\mathcal{A} = (L, C, Act, s_0, \rightarrow, I_C)$ where*

- *L is a finite set of locations, typically denoted by ℓ*
- *C is a finite set of clocks, typically denoted by c*
- *Act is a finite set of actions*
- *$s_0 \in L$ is the initial location*
- *$\rightarrow \subseteq L \times G(C) \times Act \times 2^C \times L$ is the (non-deterministic) transition relation. We normally write $\ell \xrightarrow{g,a,r} \ell'$ for a transition, where ℓ is the source location, g is the guard over the clocks, a is the action, and r is the set of clocks reset.*
- *$I_C : L \rightarrow G(C)$ is a function mapping locations to downwards closed clock invariants.*

Using the definition of extended timed automata we can now define networks of timed automata, as modelled by UPPAAL, see [9] for details. A network of timed automata is a parallel composition of extended timed automata that enables synchronisation over a finite set of channel names $Chan$. We let $ch!$ and $ch?$ denote the output and input action on a channel $ch \in Chan$.

[1] With strong compatibility, see [19].

Definition 4 (Network of timed automata). *Let* $Act = \{ch!, ch?|ch \in Chan\} \cup \{\tau\}$ *be a finite set of actions, and let* C *be a finite set of clocks. Then the parallel composition of extended timed automata* $\mathcal{A}_i = (L_i, C, Act, s_0^i, \rightarrow_i, I_C^i)$ *for all* $1 \leq i \leq n$*, where* $n \in \mathbb{N}$*, is a network of timed automata, denoted* $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2||\ldots||\mathcal{A}_n$*.*

The concrete semantics of timed automata [9] gives rise to a possibly uncountable state space. To model check it a finite abstraction of the state space is needed; the abstraction used by most model checkers is the zone abstraction [14]. Zones are sets of clock constraints that can be efficiently represented by Difference Bounded Matrices (DBMs) [12]. The fundamental operations of DBMs are:

- $D \uparrow$ modifying the constraints such that the DBM represents all the clock valuations that can result from delay from the current constraint set
- $D \cap D'$ adding additional constraints to the DBM, e.g. because a transition is taken that imposes a clock constraint (guard clock constraints can also be represented as a DBM, and we will do so) [2]. The additional constraints might also make the DBM empty, meaning that no clock valuations can satisfy the constraints.
- $D[r]$ where $r \subseteq C$ is a clock reset of the clocks in r.
- D/B doing maximal bounds extrapolation, where $B : C \rightarrow \mathbb{N}_0$ is the maximal bounds needed to be tracked for each clock. Extrapolation with respect to maximal bounds [8] is needed to make the number of DBMs finite. Basically, it is a mapping for each clock indicating the maximal possible constant the clock can be compared to in the future. It is used in such a way that if the value of a clock has passed its maximal constant, the clock's value is indistinguishable for the model.
- $D \subseteq D'$ for checking if the constraints of D' imply the constraints of D, i.e. D' is a more relaxed DBM. D' has the behaviour of D and possibly more.

Lemma 1. *Timed automata under the zone abstraction are well-structured transition systems:* $(S, \Rightarrow_{DBM}, Act, \sqsubseteq)$ *s.t.*

1. S *consists of pairs* (ℓ, D) *where* $\ell \in L$*, and* D *is a DBM.*
2. $\Rightarrow_{DBM}$ *is the symbolic transition function using DBMs, and Act is as before*
3. $\sqsubseteq$*:* $S \rightarrow S$ *is defined as* $(\ell, D) \sqsubseteq (\ell', D')$ *iff* $\ell = \ell'$*, and* $D \subseteq D'$*.*

Remark that part of the ordering $\sqsubseteq$ is compared using discrete equality (the location vector), while only a subpart is compared using a well-quasi-ordering. Without loss of generality, and as done in [17], we can split the state into an explicit part $\mathcal{S}$, and a symbolic part Σ, s.t. the well-structured transition system is defined over $\mathcal{S} \times \Sigma$. We denote the explicit part as $s, t, r \in \mathcal{S}$ and the symbolic part of states by $\sigma, \tau, \rho, \pi, \upsilon \in \Sigma$, and a state as a pair (s, σ).

Model checking of safety properties is done by proving or disproving the reachability of a certain concrete goal location s_g.

[2] The DBM might need to be put into normal form after more constraints have been added [14].

Definition 5 ((Safety) Model checking of a well-structured transition system). *Given a well-structured transition system* $(\mathcal{S} \times \Sigma, \rightarrow, \sqsubseteq)$*, an initial state* $(s_0, \sigma_0) \in \mathcal{S} \times \Sigma$*, and a goal location* s_g *does a path exist* $(s_0, \sigma_0) \rightarrow \cdots \rightarrow (s_g, \sigma'_g)$*.*

In practice, the transition system is constructed *on-the-fly* starting from (s_0, σ_0) and recursively applying $\rightarrow$ to discover new states. To facilitate this, we extend the next-state interface of PINS with subsumption:

Definition 6. *A next-state interface with subsumption has three functions:*
INITIAL-STATE$() = (s_0, \sigma_0)$*,*
NEXT-STATE$((s, \sigma)) = \{(s_1, \sigma_1), \ldots, (s_n, \sigma_n)\}$ *returning all successors of* (s, σ)*,* $(s, \sigma) \rightarrow (s_i, \sigma_i)$*, and*
COVERS$(\sigma', \sigma) = \sigma \sqsubseteq \sigma'$ *returning whether the symbolic part* σ' *subsumes* σ*.*

4 A Multi-core Timed Reachability Tool

For the construction of our real-time multi-core model checker, we made an effort to reuse and combine existing components, while extending their functionality where necessary. For the specification models, we use the UPPAAL XML format. This enables the use of its extensive real-time modelling language through an excellent user interface. To implement the model's semantics (in the form of a next-state interface) we rely on opaal and the UPPAAL DBM library.[3] Finally, LTSMIN is used as a model checking backend, because of its language-independent design.

Fig. 1 gives an overview of the new toolchain. It shows how the XML input file is read by opaal which generates C++ code. The C++ file implements the PINS interface with subsumption specifically for the input model. Hence, after compilation (C++ compiler), LTSMIN can load the object file to perform the model checking.

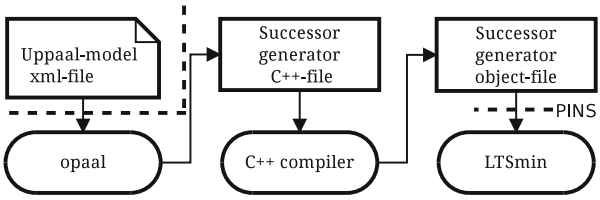

Fig. 1. Reachability with subsumption [17]

Previously, the opaal tool was used to generate Python code [17], but important parts of its infrastructure, e.g., analysing the model to find max clock constants [8], can be reused. In Sec. 5, we describe how opaal implements the semantics of timed automata, and the structure of the generated C++ code.

The PINS interface of the LTSMIN tool [13] has been shown to enable efficient, yet language-independent, model checking algorithms of different flavours, inter alia: distributed [13], symbolic [13] and multi-core reachability [22,24], and LTL model checking [20,21]. We extended the PINS interface to distinguish the new symbolic states of the opaal successor generator according to Def. 6. In Sec. 6, we describe our new multi-core reachability algorithms with subsumption.

[3] `http://people.cs.aau.dk/~adavid/UDBM/`

5 Successor Generation Using Opaal

The opaal tool was designed to rapidly prototype new model checking features and as such was designed to be extended with other successor generators. It already implements a substantial part of the UPPAAL features. For an explanation of the UPPAAL features see [9, p. 4-7]. The new C++ opaal successor generator supports the following features: templates, constants, bounded integer variables, arrays, selects, guards, updates, invariants on both variables and clocks, committed and urgent locations, binary synchronisation, broadcast channels, urgent synchronisation, selects, and much of the C-like language that UPPAAL uses to express guards and variable updates.

A state in the symbolic transition system using DBMs, is a location vector and a DBM. To represent a state in the C++ code we use a struct with a number of components: one integer for each location, and a pointer to a DBM object from the UPPAAL DBM library. Therefore a state is a tuple: $(\ell_1, \ldots, \ell_n, D)$.

The INITIAL-STATE function is rather straightforward: it returns a state struct initialised to the initial location vector, and a DBM representing the initial zone (delayed, and with invariants applied as necessary). The structure of the NEXT-STATE function is more involved, because it needs to consider the syntactic structure of the model, as can be seen in Alg. 1.

Alg. 1. Overall structure of the successor generator

1 **proc** NEXT-STATE($s_{in} = (\ell_1, \ldots, \ell_n, D)$)
2 out_states $:= \emptyset$
3 **for** $\ell_i \in \ell_1, \ldots, \ell_n$
4 **for all** $\ell_i \xrightarrow{g,a,r} \ell'_i$
5 $D' := D \cap g$
6 **if** $D' \neq \emptyset$ ▷*is the guard satisfied?*
7 **if** $a = \tau$ ▷*this is not a synchronising transition*
8 $D' := D'[r]\uparrow$ ▷*clock reset, delay*
9 $D' := D' \cap I^i_C(\ell'_i) \cap \bigcap_{k \neq i} I^k_C(\ell_k)$ ▷*apply clock invariants*
10 **if** $D' \neq \emptyset$
11 $D' := D'/B(\ell_1, \ldots, \ell'_i \ldots, \ell_n)$
12 out_states := out_states $\cup \{(\ell_1, \ldots, \ell'_i, \ldots, \ell_n, D')\}$
13 **else if** $a = ch!$ ▷*binary sync. sender*
14 **for** $\ell_j \in \ell_1, \ldots, \ell_n, j \neq i$
15 **for all** $\ell_j \xrightarrow{g_j, ch?, r_j} \ell'_j$ ▷*find receivers*
16 **if** $D'' := D' \cap g_j \neq \emptyset$ ▷*receiver guard satisfied?*
17 $D'' := D''[r][r_j]\uparrow$ ▷*clock resets, delay*
18 $D'' := D'' \cap I^i_C(\ell'_i) \cap I^j_C(\ell'_j) \cap \bigcap_{k \notin \{i,j\}} I^k_C(\ell_k)$ ▷*apply clock invariants*
19 **if** $D'' \neq \emptyset$
20 $D'' := D''/B(\ell_1, \ldots, \ell'_i, \ldots, \ell'_j \ldots, \ell_n)$
21 out_states := out_states $\cup \{(l_1, \ldots, l'_i, \ldots, l'_j, \ldots, l_n, D'')\}$
22 **return** out_states

At l. 4, we consider all outgoing transitions for the current location of each process (l. 3). If the transition is internal, we can evaluate it right away, and possibly generate a successor at l. 12. If it is a sending synchronisation (`ch!`), we need to find possible synchronisation partners (l. 15). So again we iterate over all processes and the transitions of their current locations (l. 14–21).

In the generated C++ code a few optimisations have been made, compared to Alg. 1: The loops on line l. 3 and l. 14 have been unrolled, since the number of processes they iterate over is known beforehand. In that manner the transitions to consider can be efficiently found. As an optimisation, before starting the code generation, we compute the set of all possible receivers for all channels, for the unrolling of l. 14. In practice there are usually many receivers but few senders for each channel, resulting in the unrolling being an acceptable trade-off.

When doing the max bounds extrapolation ($/$) in Alg. 1, we obtain the bounds from a location-dependent function $B : L_1 \times \cdots \times L_n \to (C \to \mathbb{N}_0)$. This function is pre-computed in opaal using the method described in [8].

Some features are not formalised in this work, but have been implemented for ease of modelling. We support integer variables, urgency that can be modelled using urgent/committed locations and urgent channels, but also channel arrays with dynamically computed senders, broadcast channels, and process priorities. These are all implemented as simple extensions of Alg. 1. Other features are supported in the form of a syntactic expansion, namely: selects, and templates.

To make the NEXT-STATE function thread-safe, we had to make the UPPAAL DBM library thread-safe. Therefore, we replaced its internal allocator with a concurrent memory allocator (see Sec. 7). We also replaced the internal hash table, used to filter duplicate DBM allocations, with a concurrent hash table.

6 Well-Structured Transition Systems in LTSmin

The current section presents the parallel reachability algorithm that was implemented in LTSMIN to handle well-structured transition systems. According to Def. 6, we can split up states into a discrete part, which is always compared using equality (for timed automata this consists of the locations and variables), and a part that is compared using a well-quasi-ordering (for timed automata this is the DBM).

Alg. 2. Reachability with subsumption [17]

```
1 proc reachability(s_g)
2   W := { INITIAL-STATE() }; P := ∅
3   while W ≠ ∅
4     W := W \ (s, σ) for some (s, σ) ∈ W
5     P := P ∪ {(s, σ)}
6     for (t, τ) ∈ NEXT-STATE((s, σ)) do
7       if t = s_g then report & exit
8       if ∄ρ: (t, ρ) ∈ W ∪ P ∧ COVERS(ρ, τ)
9         W := W \ {(t, ρ) | COVERS(τ, ρ)} ∪ (t, τ)
```

We recall the sequential algorithm from [17] (Alg. 2) and adapt it to use the next-state interface with subsumption. At its basis, this algorithm is a search with a waiting set (W), containing the states to be explored, and a passed set (P), containing the states that are already explored.

New successors (t, τ) are added to W (l. 9), but only if they are not subsumed by previous states (l. 8). Additionally, states in the waiting set W that are subsumed by the new state are discarded (l. 9), avoiding redundant explorations.

6.1 A Parallel Reachability Algorithm with Subsumption

In the parallel setting, we localize all work sets (Q_p, for each worker p) and create a shared data structure L storing both W and P. We attach a status flag passed or waiting to each state in L to create a global view of the passed and waiting set and avoid unnecessary reexplorations. L can be represented as a multimap, saving multiple symbolic state parts with each explicit state part $L : \mathcal{S} \to \Sigma^*$. To make L thread-safe, we protect its operations with a fine-grained locking mechanism that locks only the part of the map associated with an explicit state part s: **lock**$(L(s))$, similar to the spinlocks in [22]. An off-the-shelf load balancer takes care of distributing work at the startup and when some Q_p runs empty prematurely. This design corresponds to the shared hash table approach discussed in Sec. 2 and avoids a *static partitioning* of the state space.

Alg. 3 presents the discussed design. The algorithm is initialised by calling reachability with the desired number of threads P and a discrete goal location s_g. This method initialises the shared data structure L and gets the initial state using the INITIAL-STATE function from the next-state interface with subsumption. The initial state is then added to L and the worker threads are initialised at l. 6. Worker thread 1 explores the initial state; work load is propagated later.

The **while** loop on l. 20 corresponds closely to the sequential algorithm, in a quick overview: a state (s, σ) is taken from the work set at l. 21, its flag is set to passed by grab if it were not already, and then the successors (t, τ) of (s, σ) are checked against the passed and the waiting set by update. We now discuss the operations on L (update, grab) and the load balancing in more detail.

To implement the subsumption check (line l. 8–9 in Alg. 2) for successors (t, τ) and to update the waiting set concurrently, update is called. It first locks

Alg. 3. Reachability with cover update of the waiting set

```
1  global L : S → (Σ × {waiting, passed})*
2  proc reachability(P, s_g)
3      L := S → ∅
4      (s_0, σ_0) := s := INITIAL-STATE()
5      L(s_0) := (σ_0, waiting)
6      search(s, s_g, 1)|| ... ||search(s, s_g, P)
7  proc update(t, τ)
8      lock(L(t))
9      for (ρ, f) ∈ L(t) do
10         if COVERS(ρ, τ)
11             unlock(L(t))
12             return true
13         else if f = waiting ∧ COVERS(τ, ρ)
14             L(t) := L(t) \ (ρ, waiting)
15     L(t) := L(t) ∪ (τ, waiting)
16     unlock(L(t))
17     return false
18 proc search((s_0, σ_0), s_g, p)
19     Q_p := if p = 1 then {(s_0, σ_0)} else ∅
20     while Q_p ≠ ∅ ∨ balance(Q_p)
21         Q_p := Q_p \ (s, σ) for some (s, σ) ∈ Q_p
22         if ¬grab(s, σ) then continue
23         for (t, τ) ∈ NEXT-STATE((s, σ)) do
24             if t = s_g then report & exit
25             if ¬update(t, τ)
26                 Q_p := Q_p ∪ (t, τ)
27 proc grab(s, σ)
28     lock(L(s))
29     if σ ∉ L(s) ∨ passed = L(s, σ)
30         unlock(L(s))
31         return false
32     L(s, σ) := passed
33     unlock(L(s))
34     return true
```

L on t. Now, for all symbolic parts and status flag ρ, f associated with t, the method checks if τ is already covered by ρ. In that case (t, τ) will not be explored. Alternatively, all ρ with status flag waiting that are covered by τ are removed from $L(t)$ and τ is added. The update algorithm maintains the invariant that a state in the waiting set is never subsumed by any other state in L: $\forall s\, \forall (\rho, f), (\rho', f') \in L(s)\colon f = \mathsf{waiting} \wedge \rho \neq \rho' \Rightarrow \rho \not\sqsubseteq \rho'$ (**Inv. 1**). Hence, similar to Alg. 2 l. 8–9, it can never happen that (t, τ) first discards some (t, ρ) from $L(s)$ (l. 14) and is discarded itself in turn by some (t, ρ') in $L(s)$ (l. 10), since then we would have $\rho \sqsubseteq \tau \sqsubseteq \rho'$; by transitivity of $\sqsubseteq$ and the invariant, ρ and ρ' cannot be both in $L(t)$. Finally, notice that update unlocks $L(t)$ on all paths.

The task of the method grab is to check if a state (s, σ) still needs to be explored, as it might have been explored by another thread in the meantime. It first locks $L(s)$. If σ is no longer in $L(s)$ or it is no longer globally flagged waiting (l. 29), it is discarded (l. 22). Otherwise, it is "grabbed" by setting its status flag to passed. Notice again that on all paths through grab, $L(s)$ is unlocked.

Finally, the method balance handles termination detection and load balancing. It has the side-effect of adding work to Q_p. We use a standard solution [25].

6.2 Exploration Order

The shared hash table approach gives us the freedom to allow for a DFS or BFS exploration order depending on the implementation of Q_p. Note, however, that only pseudo-DFS/BFS is obtained, due to randomness introduced by parallelism.

It has been shown for timed automata that the number of generated states is quite sensitive to the exploration order and that in most cases strict BFS shows the best results [11]. Fortunately, we can obtain strict BFS by synchronising workers between the different BFS levels. To this end, we first split Q_p into two separate sets that hold the current BFS level (C_p) and the next BFS level (N_p) [2]. The order within these sets does not matter, as long as the current is explored before the next set. Load balancing will only be performed on C_p, hence a level terminates once $C_p = \emptyset$ for all p. At this point, if $N_p = \emptyset$ for all p, the algorithm can terminate because the next BFS level is empty. The synchronising reduce method counts $\sum_{i=1}^{P} |N_i|$ (similar to `mpi_reduce`).

Alg. 4. Strict parallel BFS

```
1  proc search(s_0, σ_0, p)
2    C_p := if p = 1 then {(s_0, σ_0)} else ∅
3    do
4      while C_p ≠ ∅ ∨ balance(C_p)
5        C_p := C_p \ (s, σ) for some (s, σ) ∈ C_p
6              ...
7                N_p := N_p ∪ (t, τ)
8      load := reduce(sum, |N_p|, P)
9      C_p, N_p := N_p, ∅
10   while load ≠ 0
```

Alg. 4 shows a parallel strict-BFS implementation. An extra outer loop iterates over the levels, while the inner loop (l. 4–7) is the same as in Alg. 3. Except for the lines that add and remove states to and from the work set, which now operate on N_p and C_p. The (pointers to) the work sets are swapped, after the reduce call at l. 8 calculates the load of the next level.

6.3 A Data Structure for Semi-symbolic States

In [22], we introduced a lockless hash table, which we reuse here to design a data structure for L that supports the operations used in Alg. 3. To allow for massive parallelism on modern multi-core machines with steep memory hierarchies, it is crucial to keep a low memory footprint [22, Sec. II]. To this end, lookups in the large table of state data are filtered through a separate smaller table of hashes. The table assigns a unique number (the hash location) to each explicit state stored in it: $D\colon \mathcal{S} \to \mathbb{N}$. In finite reality, we have: $D\colon \mathcal{S} \to \{1, \ldots, N\}$.

We now reuse the state numbering of D to create a multimap structure for L. The first component of the new data structure is an array $I[N]$ used for indexing on the explicit state parts. To associate a set of symbolic states (pointers to DBMs) with our explicit state stored in $D[x]$, we are going to attach a linked list structure to $I[x]$. Creating a standard linked list would cause a single *cache line* access per element, increasing the memory footprint, and would introduce costly synchronisations for each modification. Therefore, we allocate multi-buckets, i.e., an array of pointers as one linked list element. To save memory, we store lists of just one element directly in I and completely fill the last multi-bucket.

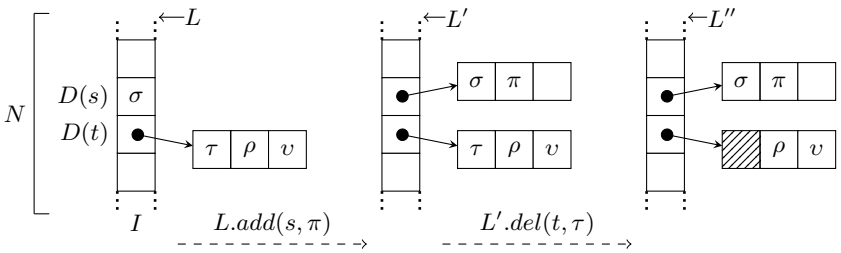

Fig. 2. Data structure for L, and operations

Fig. 2 shows three instances of the discussed data structure: L, L' and L''. Each multimap is a pointer (arrow) to an array I shown as a vertical bucket array. L contains $\{(s,\sigma),(t,\tau),(t,\rho),(t,\upsilon)\}$. We see how a multi-bucket with (fixed) length 3 is created for t, while the single symbolic state attached to s is kept directly in I. The figure shows how σ is moved when (s,π) is added by the add operation (dashed arrow), yielding L'. Adding π to t would have moved υ to a new linked multi-bucket together with π.

Removing elements from the waiting list is implemented by marking bucket entries as tombstone, so they can later be reused (see L''). This avoids memory fragmentation and expensive communication to reuse multi-buckets. For highest scalability, we allocate multi-buckets of size 8, equal to a cache line. Other values can reduce memory usage, but we found this sufficiently efficient (see Sec. 7).

We still need to deal with locking of explicit states, and storing of the various flags for symbolic states (waiting/passed). Internally, the algorithms also need to distinguish between the different buckets: empty, tomb stone, linked list pointers

and symbolic state pointers. To this end, we can bitcram additional bits into the pointers in the buckets, as is shown in Fig. 3. Now **lock**$(L(s))$ can be implemented as a spinlock using the atomic *compare-and-swap* (CAS) instruction on $I[s]$ [22]. Since all operations on $L(s)$ are done after **lock**$(L(s))$, the corresponding bits of the buckets can be updated and read with normal load and store instructions.

```
struct link_or_dbm {
  bit pointer[60]
  bit flag ∈ {waiting, passed}
  bit lock ∈ {locked, unlocked}
  bit status[2] ∈ {empty, tomb,
                   dbm_ptr, list_ptr}
}
```

Fig. 3. Bit layout of word-sized bucket

6.4 Improving Scalability through a Non-blocking Implementation

The size of the critical regions in Alg. 3 depends crucially on the $|\Sigma|/|\mathcal{S}|$ ratio; a higher ratio means that more states in $L(t)$ have to be considered in the method update(t, τ), affecting scalability negatively. A similar limitation is reported for distributed reachability [15]. Therefore, we implemented a *non-blocking* version: instead of first deleting all subsumed symbolic states with a waiting flag, we atomically replace them with the larger state using CAS. For a failed CAS, we retry the subsumption check after a reread. L can be atomically extended using the well-known *read-copy-update* technique. However, workers might miss updates by others, as Inv. 1 no longer holds. This could cause $|\Sigma|$ to increase again.

7 Experiments

To investigate the performance of the generated code, we compare full reachability in opaal+LTSmin with the current state-of-the-art (uppaal).[4] To investigate scalability, we benchmarked on a 48-core machine (a four-way AMD Opteron$^{\text{TM}}$ 6168) with a varying number of threads. Statistics on memory usage were gathered and compared against uppaal. Experiments were repeated 5 times.

We consider three models from the uppaal demos: `viking` (one discrete variable, but many synchronisations), `train-gate` (relatively large amount of code, several variables), and `fischer` (very small discrete part). Additionally, we experiment with a generated model, `train-crossing`, which has a different structure from most hand-made models. For some models, we created multiple numbered instances, the numbers represent the number of processes in the model.

For uppaal, we ran the experiments with BFS and disabled space optimisation. The `opaal_ltsmin` script in opaal was used to generate and compile models. In LTSmin we used a fixed hash table (`--state=table`) size of 2^{26} states (`-s26`), waiting set updates as in Alg. 3 (`-u1`) and multi-buckets of size 8 (`-l8`).

Performance & Scalability. Table 1 shows the reachability runtimes of the different models in uppaal and opaal+LTSmin with strict BFS (`--strategy=sbfs`). Except for `fischer6`, we see that both tools compete with each other on the

[4] opaal is available at `https://code.launchpad.net/~opaal-developers/opaal/opaal-ltsmin-succgen`, LTSmin at `http://fmt.cs.utwente.nl/tools/ltsmin/`

Table 1. $\mathcal{S}$, $|\Sigma|$ ($\frac{|\Sigma|}{|\mathcal{S}|}$) and runtimes (sec) in UPPAAL and opaal+LTSMIN (strict BFS)

	$\lvert\mathcal{S}\rvert$	UPPAAL		opaal+LTSMIN (cores)							
		T	$\lvert\Sigma\rvert$	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_2	T_8	T_{16}	T_{32}	T_{48}
train-gate-N10	7e+07	837.4	1.0	1.0	1.0	573.3	297.8	76.7	39.4	21.1	14.4
viking17	1e+07	207.8	1.0	1.5	1.5	331.5	172.5	44.2	22.7	11.9	8.6
train-gate-N9	7e+06	76.8	1.0	1.0	1.0	52.4	28.5	7.7	4.1	2.4	2.0
viking15	3e+06	38.0	1.0	1.5	1.5	67.0	34.8	9.7	5.1	3.0	2.3
train-crossing	3e+04	48.3	20.8	16.1	17.3	24.5	37.2	5.8	2.7	2.0	2.1
fischer6	1e+04	0.1	0.3	50.1	50.1	219.2	129.2	46.4	36.1	32.9	31.8

sequential runtimes, with 2 threads however opaal+LTSMIN is faster than UPPAAL. With the massive parallelism of 48 cores, we see how verification tasks of minutes are reduced to mere seconds. The outlier, `fischer6`, is likely due to the use of more efficient clock extrapolations in UPPAAL, and other optimisations, as witnessed by the evolution of the runtime of this model in [10,4].

We noticed that the 48-core runtimes of the smaller models were dominated by the small BFS levels at the beginning and the end of the exploration due to synchronisation in the load balancer and the reduce function. This overhead takes consistently 0.5–1 second, while it handles less than thousand states. Hence to obtain useful scalability measurements for small models, we excluded this time in the speedup calculations (Fig. 4–7). The runtimes in Table 1–2 still include this overhead. Fig. 4 plots the speedups of strict BFS with the standard deviation drawn as vertical lines (mostly negligible, hence invisible). Most models show almost linear scalability with a speedup of up to 40, e.g. `train-gate-N10`. As expected, we see that a high $|\Sigma|/|\mathcal{S}|$ ratio causes low scalability (see `fischer` and `train-crossing` and Table 1). Therefore, we tried the non-blocking variant (Sec. 6.3) of our algorithm (`-n`). As expected, the speedups in Fig. 5 improve and the runtimes even show a threefold improvement for `fischer.6` (Table 2). The efficiency on 48 cores remains closely dependent to the $|\Sigma|/|\mathcal{S}|$ ratio of the model (or the average length of the lists in the multimap), but the scalability is now at least sub-linear and not stagnant anymore.

We further investigated different search orders. Fig. 6 shows results with pseudo BFS order (`--strategy=bfs`). While speedups become higher due to the lacking level synchronisations, the loose search order tends to reach "large" states later and therefore generates more states for two of the models ($|\Sigma_1|$ vs $|\Sigma_{48}|$ in Table 2). This demonstrates that our strict BFS implementation indeed pays off.

Finally, we also experimented with randomized DFS search order (`-prr --strategy=dfs`). Table 2 shows that DFS causes again more states to be generated. But, surprisingly, the number of states actually reduces with the parallelism for the `fischer6` model, even below the state count of strict BFS from Table 1! This causes a super-linear speedup in Fig. 7 and threefold runtime improvement over strict BFS. We do not consider this behaviour as an exception (even though `train-crossing` does not show it), since it is compatible with our observation that parallel DFS finds shorter counter examples than parallel BFS [18, Sec. 4.3].

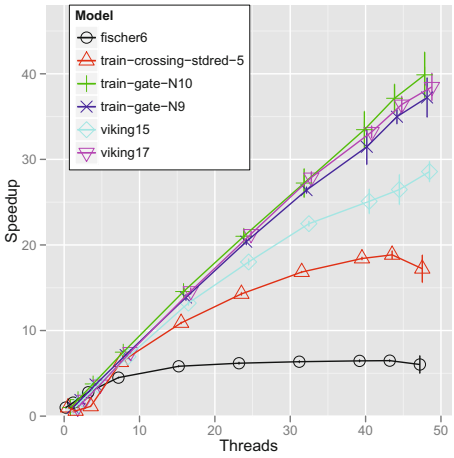

Fig. 4. Speedup strict BFS

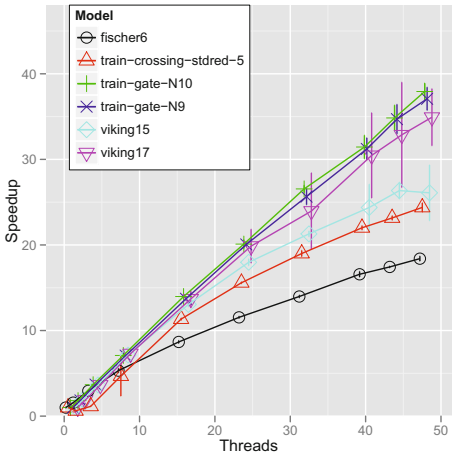

Fig. 5. Speedup non-blocking strict BFS

Design decisions. Some design decisions presented here were motivated by earlier work that has proven successful for multi-core model checking [22,18]. In particular, we reused the shared hash table and a *synchronous* load balancer [25]. Even though we observed load distributions close to ideal, a modern work stealing solution might still improve our results, since the work granularity for timed reachability is higher than for untimed reachability. The main bottlenecks, however, have proven to be the increase in state count by parallelism and the cost of the spinlocks due to a high $|\Sigma|/|\mathcal{S}|$ ratio. The latter we partly solved with a non-blocking algorithm. Strict BFS orders have proven to aid the former problem and randomized DFS orders could aid both problems.

Memory usage. Table 3 shows the memory consumption of UPPAAL (U-S0) and sequential opaal+LTSMIN (O+L_1) with strict BFS. From it, we conclude that

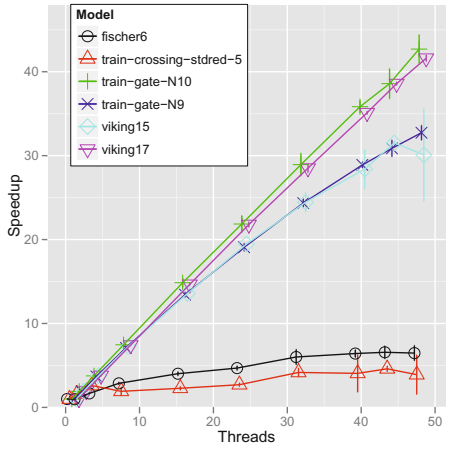

Fig. 6. Speedup pseudo BFS

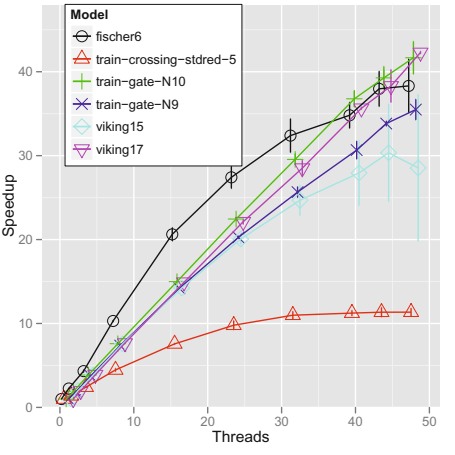

Fig. 7. Speedup randomized pseudo DFS

Table 2. $|\Sigma|$ ($\frac{|\Sigma|}{|\mathcal{S}|}$) and runtimes (sec) with non-blocking SBFS, DFS and BFS

	NB SBFS				DFS				BFS			
	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}
train-gate-N10	1.0	1.0	547.9	14.5	1.0	1.0	647.8	15.6	1.0	1.0	559.3	13.1
viking17	1.5	1.5	320.1	9.2	1.6	1.6	386.5	9.1	1.5	1.5	325.6	7.8
train-gate-N9	1.0	1.0	52.1	2.1	1.0	1.0	61.7	1.7	1.0	1.0	51.9	1.6
viking15	1.5	1.5	64.8	2.5	1.6	1.6	80.2	3.1	1.5	1.5	66.0	2.3
train-crossing	16.1	16.1	24.1	1.8	169.8	179.0	3371.0	297.4	16.1	37.1	24.5	157.5
fischer6	50.1	50.1	201.3	12.0	54.4	39.4	405.1	10.6	50.1	58.1	206.0	32.3

Table 3. Memory usage (MB) of both UPPAAL (U-S0 and U-S2) and opaal+LTSMIN

	T	D	L	L2	Q	dbm	O+L_1	O+L_{48}	O+L_1^T	O+L_{48}^T	U-S0	U-S2
train-gate-N10	777	5989	499	499	249	1363	8101	8241	2790	3028	6091	3348
viking17	156	1040	536	214	40	87	1704	1931	828	1047	1579	722
train-gate-N9	81	549	50	50	24	61	684	815	214	347	607	332
viking15	32	190	112	44	8	55	364	581	203	423	333	162
train-crossing	0	2	5	7	0	419	426	623	425	622	48	64
fischer6	0	0	5	9	1	176	429	512	290	429	0	4

our memory usage is within 25% of UPPAAL's for the larger models (where these measurements are precise enough). Furthermore, we extensively experimented with different concurrent allocators and found that TBB malloc (used in this paper) yields the best performance for our algorithms.[5] Its overhead (O+L_1 vs O+L_{48} in Table 3) appears to be limited to a moderate fixed amount of 250MB more than the sequential runs, for which we used the normal `glibc` allocator.

We also counted the memory usage inside the different data structures: the multimap L (including partly-filled multi-buckets), the hash table D, the combined local work sets (Q), and the DBM duplicate table (dbm). As we expected the overhead of the 8-sized multi-buckets is little compared to the size of D and the DBMs. We may however replace D with the compressed, parallel tree table (T) from [24]. The resulting total memory usage (O+L^T), can now be dominated by L, .i.e., for `viking17`. But if we reduce L to a linked list (`-l2`), its size shrinks by 60% to 214MB for this model (L2). Just a modest gain compared to the total.

For completeness, we included the results of UPPAAL's state space optimisation (U-S2). As expected, it also yields great reductions, which is the more interesting since the two techniques are orthogonal and could be combined.

8 Conclusions

We presented novel algorithms and data structures for multi-core reachability on well-structured transition systems and an efficient implementation for timed automata in particular. Experiments show good speedups, up to 40 times on a 48-core machine and also identify current bottlenecks. In particular, we see speedups

[5] cf. `http://fmt.cs.utwente.nl/tools/ltsmin/formats-2012/` for additional data.

of 58 times compared to UPPAAL. Memory usage is limited to an acceptable maximum of 25% more than UPPAAL.

Our experiments demonstrate the flexibility of the search order that our parallel approach allows for. BFS-like order is shown to be occasionally slightly faster than strict BFS but is substantially slower on other models, as previously observed in the distributed setting. A new surprising result is that parallel randomized (pseudo) DFS order sometimes reduces the state count below that of strict BFS, yielding a substantial speedup in those cases.

Previous work has shown that better parallel reachability [22,24] crucially enables new and better solutions to parallel model checking of liveness properties [20,18]. Therefore, our natural next step is to port multi-core nested depth-first search solutions to the timed automata setting.

Because of our use of generic toolsets, more possibilities are open to be explored. The opaal support for the UPPAAL language can be extended and support for optimisations like symmetry reduction and partial order reduction could be added, enabling easier modeling and better scalability. Additionally, lattice-based languages [17] can be included in the C++ code generator. On the backend side, the distributed [13] and symbolic [13] algorithms in LTSMIN can be extended to support subsumption, enabling other powerful means of verification. We also plan to add a join operator to the PINS interface, to enable abstraction/refinement-based approaches [17].

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General Decidability Theorems for Infinite-State Systems. In: Proceedings of Eleventh Annual IEEE Symposium on Logic in Computer Science, LICS 1996, pp. 313–321 (July 1996)
2. Agarwal, V., Petrini, F., Pasetto, D., Bader, D.A.: Scalable Graph Exploration on Multicore Processors. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Amnell, T., Behrmann, G., Bengtsson, J.E., D'Argenio, P.R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K.G., Möller, M.O., Pettersson, P., Weise, C., Yi, W.: UPPAAL - Now, Next, and Future. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 99–124. Springer, Heidelberg (2001)
5. Barnat, J., Ročkai, P.: Shared Hash Tables in Parallel Model Checking. Electronic Notes in Theoretical Computer Science 198(1), 79–91 (2007); Proceedings of PDMC 2007
6. Behrmann, G.: Distributed Reachability Analysis in Timed Automata. International Journal on Software Tools for Technology Transfer 7(1), 19–30 (2005)
7. Behrmann, G., Bengtsson, J.E., David, A., Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL Implementation Secrets. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 3–22. Springer, Heidelberg (2002)
8. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static Guard Analysis in Timed Automata Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003)

9. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
10. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing Uppaal over 15 years. Software: Practice and Experience 41(2), 133–142 (2011)
11. Behrmann, G., Hune, T., Vaandrager, F.: Distributing Timed Model Checking - How the Search Order Matters. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
12. Bengtsson, J.: Clocks, DBMs and states in timed systems. PhD thesis, Uppsala University (2002)
13. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
14. Bouyer, P.: Forward analysis of updatable timed automata. Formal Methods in System Design 24(3), 281–320 (2004)
15. Braberman, V., Olivero, A., Schapachnik, F.: Dealing with practical limitations of distributed timed model checking for timed automata. Formal Methods in System Design 29, 197–214 (2006), doi:10.1007/s10703-006-0012-3
16. Comon, H., Jurski, Y.: Timed Automata and the Theory of Real Numbers. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 242–257. Springer, Heidelberg (1999)
17. Dalsgaard, A.E., Hansen, R.R., Jørgensen, K.Y., Larsen, K.G., Olesen, M.C., Olsen, P., Srba, J.: `opaal`: A Lattice Model Checker. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 487–493. Springer, Heidelberg (2011)
18. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved Multi-Core Nested Depth-First Search. In: Mukund, M., Chakraborty, S. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 269–283. Springer, Heidelberg (2012)
19. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256(1-2), 63–92 (2001)
20. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core Nested Depth-First Search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
21. Laarman, A.W., van de Pol, J.: Variations on Multi-Core Nested Depth-First Search. In: Barnat, J., Heljanko, K. (eds.) PDMC. EPTCS, vol. 72, pp. 13–28 (2011)
22. Laarman, A.W., van de Pol, J., Weber, M.: Boosting Multi-Core Reachability Performance with Shared Hash Tables. In: Sharygina, N., Bloem, R. (eds.) Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Swiss. IEEE Computer Society (October 2010)
23. Laarman, A., van de Pol, J., Weber, M.: Multi-Core LTSmin: Marrying Modularity and Scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011)
24. Laarman, A., van de Pol, J., Weber, M.: Parallel Recursive State Compression for Free. In: Groce, A., Musuvathi, M. (eds.) SPIN Workshops 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011)
25. Sanders, P.: Lastverteilungsalgorithmen fur Parallele Tiefensuche. number 463. In: Fortschrittsberichte, Reihe 10. VDI. Verlag (1997)