Multi-Core Emptiness Checking of Timed Büchi Automata using Inclusion Abstraction

Alfons Laarman[1], Mads Chr. Olesen[2], Andreas Dalsgaard[2], Kim G. Larsen[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. This paper contributes to the multi-core model checking of timed automata (TA) with respect to liveness properties, by investigating checking of TA Büchi emptiness under the very coarse inclusion abstraction or zone subsumption, an open problem in this field.
We show that in general Büchi emptiness is not preserved under this abstraction. However, we prove that some other structural properties are preserved. Solving the problem partly, we propose a variation of the classic nested depth-first search (NDFS) algorithm that takes advantage of these properties thereby exploiting inclusion abstraction. In addition, we extend the multi-core CNDFS algorithm with subsumption, providing the first parallel LTL model checking algorithm for timed automata. We prove both algorithms correct and show that other, more aggressive variations are incorrect.
The algorithms are implemented in LTSMIN, and experimental evaluations show the effectiveness and scalability of both contributions: subsumption halves the number of states in the real-world FDDI case study, and the multi-core algorithm yields speedups of up to 40 using 48 cores.

1 Introduction

Model checking safety properties can be done with reachability, but only guarantees that the system does not enter a dangerous state, not that the system actually serves some useful purpose. To model check such liveness properties is more involved since they state conditions over infinite executions, e.g. that a request must infinitely often produce a result. One of the most well-known logics for describing liveness properties is Linear Temporal Logic (LTL) [2].

The automata-theoretic approach for LTL model checking [30] solves the problem efficiently by translating it to the Büchi emptiness problem, which has been shown to be decidable for real-time systems as well [1]. However, its complexity is still exponential in the size of the model and the property. In the current paper, therefore, we consider two possible ways of alleviating this so-called state space explosion problem: (1) by utilising the many cores in modern processors, and (2) by employing coarser abstractions to the state space.

Related work. The verification of timed automata was made possible by the *region construction* of Alur and Dill [1], which represents clock valuations using constraints, called *regions*. A max-clock constant abstraction, or k-extrapolation, bounded the number of regions. Since the region construction is exponential in the number of clocks and constraints in the automaton, coarser abstractions such as the symbolic *zone abstraction* have been studied [16], and also implemented in, among others, the state-of-the-art model checker UPPAAL [25]. Later, the k-extrapolation for zones was refined to include lower clock constraints in the so-called lower/upper-bound (LU) abstraction proposed in [5]. Finally, the *inclusion abstraction*, or simply *subsumption*, prunes reachability according to the partial order of the symbolic states [15]. All these abstractions preserve reachability properties [15,5].

Model checking LTL properties on timed automata, or equivalently checking timed Büchi automata (TBA) emptiness, was proven decidable in [1], by using the region construction. Bouajjani et al. [10] showed that the region-closed simulation graph preserve TBA emptiness. Tripakis [28] proved that the k-extrapolated zone simulation graph also preserves TBA emptiness, while posing the question whether other abstractions such as the LU abstraction and subsumption also preserve this property. Li [26] showed that the LU abstraction does in fact preserve TBA emptiness.

One way of establishing TBA emptiness on a finite simulation graph is the nested depth-first (NDFS) algorithm [12]. Recently, some multi-core version of these algorithms were introduced by Laarman et al [20,18,17]. These algorithms have the following properties: their runtime is linear in the number of states in the worst case while typically yielding good scalability; they are on-the-fly [21] and yield short counter examples [17, Sec. 4.3]. The latest version, called CNDFS, combines all these qualities and decreases memory usage [17].

In previous work, we parallelised reachability for timed automata using the mentioned abstractions [14]. It resulted in almost linear scalability, and speedups of up to 60 on a 48 core machine, compared to UPPAAL. The current work extends this previous work to the setting of liveness properties for timed automata. It also shares the UPPAAL input format, and re-uses the UPPAAL DBM library.

Problem statement. Parallel model checking of liveness properties for timed systems has been a challenge for several years. While advances were made with distributed versions of e.g. UPPAAL [3], these were limited to safety properties. Furthermore, it is unknown how subsumption, the coarsest abstraction, can be used for checking TBA emptiness.

Contributions. (1) For the first time, we realize parallel LTL model checking of timed systems using the CNDFS algorithm. (2) We investigate subsumption and prove that it preserves several structural state space properties (Sec. 3), and we show how these properties can be exploited by NDFS and CNDFS (Sec. 4 and Sec. 5). (3) We implement NDFS and CNDFS with subsumption in the LTSMIN toolset [23] and opaal [13]. Finally, (4) we report on experiments that show that subsumption can reduce state spaces considerably and good parallel scalability of CNDFS with a speedup of 40 using 48 cores in one case.

2 Preliminaries: Timed Büchi Automata and Abstractions

In the current section, we first recall the formalism of timed Büchi automata (TBA), that allows modelling of both a real-time system and its liveness requirements. Subsequently, we introduce finite symbolic semantics using zone abstraction with extrapolation and subsumption. Finally, we show which properties are known to be preserved under said abstractions.

Timed Automata and Transition Systems. Def. 2 provides a basic definition of a TBA. It can be extended with features such as finitely valued variables, and parallel composition to model networks of timed automata, as done in UPPAAL [6].

Definition 1 (Guards). *Let $\mathcal{G}(\mathcal{C})$ be a conjunction of clock constraints over the set of clocks $c \in \mathcal{C}$, generalized by:*

$$g ::= c \bowtie n \mid g \wedge g \mid true$$

where $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, =, >, \geq\}$ is a comparison operator. We call a guard downwards closed if all $\bowtie \in \{<, \leq, =\}$. $\mathcal{G}(\mathcal{C})$ is diagonal-free [11].

Definition 2 (Timed Büchi Automaton). *A timed Büchi automaton (TBA) is a 6-tuple $\mathbb{B} = (L, \mathcal{C}, \mathcal{F}, l_0, \rightarrow, I_{\mathcal{C}})$, where*

- *L is a finite set of locations, typically denoted by ℓ, where $\ell_0 \in L$ is the initial location, and $\mathcal{F} \subseteq L$, is the set of accepting locations,*
- *$\mathcal{C}$ is a finite set of clocks, typically denoted by c,*
- *$\rightarrow \subseteq L \times \mathcal{G}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$ is the (non-deterministic) transition relation. We write $\ell \xrightarrow{g,R} \ell'$ for a transition, where ℓ is the source and ℓ' the target location, $g \in \mathcal{G}(\mathcal{C})$ is a transition guard, $R \subseteq \mathcal{C}$ is the set of clocks to reset, and*
- *$I_{\mathcal{C}} \colon L \rightarrow \mathcal{G}(\mathcal{C})$ is an invariant function, mapping locations to a set of guards. To simplify the semantics, we require invariants to be downwards-closed.*

The states of a TBA involve the notion of clock valuations. A clock valuation is a function $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$. We denote all clock valuations over $\mathcal{C}$ with $\mathcal{V}_{\mathcal{C}}$. We need two operations on clock valuations: $v' = v + \delta$ for delay of $\delta \in \mathbb{R}_{\geq 0}$ time units s.t. $\forall c \in \mathcal{C} \colon v'(c) = v(c) + \delta$, and reset $v' = v[R]$ of a set of clocks $R \subseteq \mathcal{C}$ s.t. $v'(c) = 0$ if $c \in R$, and $v'(c) = v(c)$ otherwise. We write $v \models g$ to mean that the clock valuation v satisfies the clock constraint g.

Definition 3 (Transition system semantics of a TBA). *The semantics of a TBA $\mathbb{B}$ is defined over the transition system $\mathcal{TS}_v^{\mathbb{B}} = (\mathcal{S}_v, s_0, \mathcal{T}_v)$ s.t.:*

1. *A state $s \in \mathcal{S}_v$ is a pair: (ℓ, v) with a location $\ell \in L$, and a clock valuation v.*
2. *An initial state $s_0 \in \mathcal{S}_v$, s.t. $s_0 = (\ell_0, v_0)$, where $\forall c \in \mathcal{C} \colon v_0(c) = 0$.*
3. *$\mathcal{T}_v \colon \mathcal{S}_v \times (\{\epsilon\} \cup \mathbb{R}_{\geq 0}) \times \mathcal{S}_v$ is a transition relation with $(s, a, s') \in \mathcal{T}_v$, denoted $s \xrightarrow{a} s'$ s.t. there are two types of transitions:*
   (a) *A discrete (instantaneous) transition: $(\ell, v) \xrightarrow{\epsilon} (\ell', v')$ if an edge $\ell \xrightarrow{g,R} \ell'$ exists, $v \models g$ and $v' = v[R]$, and $v' \models I_{\mathcal{C}}(\ell')$.*
   (b) *A delay by δ time units: $(\ell, v) \xrightarrow{\delta} (\ell, v + \delta)$ for $\delta \in \mathbb{R}_{\geq 0}$ if $v + \delta \models I_{\mathcal{C}}(\ell)$.*

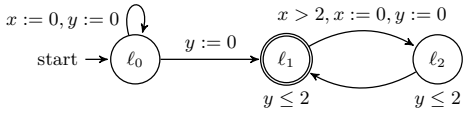


Fig. 1. A timed Büchi automaton.

We say a state $s \in \mathcal{S}$ is accepting, or $s \in \mathcal{F}$, when $s = (\ell, \ldots)$ and $\ell \in \mathcal{F}$. We write $s \xrightarrow{\delta}\xrightarrow{\epsilon} s'$ if there exists a state s'' such that $s \xrightarrow{\delta} s''$ and $s'' \xrightarrow{\epsilon} s'$. We denote an infinite run in $\mathcal{TS}_v^{\mathbb{B}} = (\mathcal{S}_v, s_0, \mathcal{T}_v)$ as an infinite path $\pi = s_1 \xrightarrow{\delta_1}\xrightarrow{\epsilon} s_2 \xrightarrow{\delta_2}\xrightarrow{\epsilon} s_3 \ldots$. The run is accepting if there exist an infinite number of indices i s.t. $s_i \in \mathcal{F}$. A(n infinite) run's time lapse is $Time(\pi) = \sum_{i \geq 1} \delta_i$. An infinite path π in $\mathcal{TS}_v^{\mathbb{B}}$ is *time convergent*, or zeno, if $Time(\pi) < \infty$, otherwise it is divergent. For example, the TBA in Fig. 1 has an infinite run: $(\ell_0, v_0) \xrightarrow{1} (\ell_0, v_0) \xrightarrow{1} \cdots$, that is not accepting, but is non-zeno. We claim that there is no accepting non-zeno run, exemplified by the finite run:
$(\ell_0, v_0) \xrightarrow{2}\xrightarrow{\epsilon} (\ell_1, v_1) \xrightarrow{0}\xrightarrow{\epsilon} (\ell_2, v_0) \xrightarrow{0}\xrightarrow{\epsilon} (\ell_1, v_0) \xrightarrow{1.9}\not\xrightarrow{\epsilon}$.

Definition 4 (A TBA's language and the emptiness problem). *The language accepted by $\mathbb{B}$, denoted $\mathcal{L}(\mathbb{B})$, is defined as the set of non-zeno accepting runs. The language emptiness problem for $\mathbb{B}$ is to check whether $\mathcal{L}(\mathbb{B}) = \emptyset$.*

Remark 1 (Zenoness). Zenoness is considered a modelling artifact as the behaviour it models cannot occur in any real system, which after all has finite processing speeds. Therefore, zeno runs should be excluded from analysis. Zenoness can either be detected during verification [19], or the TBA $\mathbb{B}$ can be syntactically transformed to a *strongly non-zeno* $\mathbb{B}$' [29], s.t. $\mathcal{L}(\mathbb{B}) = \emptyset$ iff $\mathcal{L}(\mathbb{B}') = \emptyset$. Therefore, in the following, w.l.o.g., we assume that all TBAs are strongly non-zeno.

Definition 5 (Time-abstracting simulation relation). *A time-abstracting simulation relation R is a binary relation on $\mathcal{S}_v$ s.t. if $s_1 R s_2$ then:*
- *If $s_1 \xrightarrow{\epsilon} s_1'$, then there exists s_2' s.t. $s_2 \xrightarrow{\epsilon} s_2'$ and $s_1' R s_2'$.*
- *If $s_1 \xrightarrow{\delta} s_1'$, then there exists s_2' and δ' s.t. $s_2 \xrightarrow{\delta'} s_2'$ and $s_1' R s_2'$.*

If both R and R^{-1} are time-abstracting simulation relations, we call R a time-abstracting bisimulation relation.

Symbolic Abstractions using Zones. A *zone* is a symbolic representation of an infinite set of clock valuations by means of a clock constraint. These constraints are conjuncts (Def. 6) of simple linear inequalities on clock values, and thus describe (unbounded) convex polytopes in a $|\mathcal{C}|$-dimensional plane (e.g. Fig. 2). Therefore, zones can be efficiently represented by Difference Bounded Matrices (DBMs) [7].

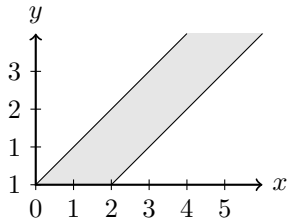


Fig. 2. A graphical representation of a zone over 2 clocks: $0 \leq y - x \leq 2$.

Definition 6 (Zones). *Similar to the guard definition, let $\mathcal{Z}(\mathcal{C})$ be the set of clock constraints over the set of clocks $c, c_1, c_2 \in \mathcal{C}$ generalized by:*

$$Z ::= c \bowtie n \mid c_1 - c_2 \bowtie n \mid Z \wedge Z \mid \mathit{true} \mid \mathit{false}$$

where $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, >, \geq\}$ is a comparison operator. We also use $=$ for equalities, short for the conjunction of $\leq$ and $\geq$.

We write $v \models Z$ if the clock valuation v is included in Z, for the set of clock valuations in a zone $[\![Z]\!] = \{v \mid v \models Z\}$, and for *zone inclusion* $Z \subseteq Z'$ iff $[\![Z]\!] \subseteq [\![Z']\!]$. Notice that $[\![\mathit{false}]\!] = \emptyset$. Using the fundamental operations below, which are detailed in [7], we define the *zone semantics* over *simulation graphs* in Def. 7. Most importantly, these operations are implementable in $O(n^3)$ or $O(n^2)$ and closed w.r.t. $\mathcal{Z}$.
delay: $[\![Z \uparrow]\!] = \{v + \delta | \delta \in \mathbb{R}_{\geq 0}, v \in [\![Z]\!]\}$,
clock reset: $[\![Z[R]]\!] = \{v[R] | v \in [\![Z]\!]\}$, and
constraining: $[\![Z \cap Z']\!] = [\![Z]\!] \cap [\![Z']\!]$.

Definition 7 (Zone semantics). *The semantics of a TBA $\mathbb{B} = (L, \mathcal{C}, \mathcal{F}, \ell_0, \rightarrow, I_{\mathcal{C}})$ under the zone abstraction is a simulation graph: $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ s.t.:*

1. *$\mathcal{S}_{\mathcal{Z}}$ consists of pairs (ℓ, Z) where $\ell \in L$, and $Z \in \mathcal{Z}$ is a zone.*
2. *$s_0 \in \mathcal{S}_{\mathcal{Z}}$ is an initial state $(\ell_0, Z_0 \uparrow \cap\, I_{\mathcal{C}}(\ell_0))$ with $Z_0 = \bigwedge_{c \in \mathcal{C}} c = 0$.*
3. *$\mathcal{T}_{\mathcal{Z}}$ is the symbolic transition function using zones, s.t. $(s, s') \in \mathcal{T}_{\mathcal{Z}}$, denoted $s \Rightarrow s'$ with $s = (\ell, Z)$ and $s' = (\ell', Z')$, if an edge $\ell \xrightarrow{g,R} \ell'$ exists, and $Z \cap g \neq \mathit{false}$, $Z' = (((Z \cap g)[R]) \uparrow) \cap\, I_{\mathcal{C}}(\ell')$ and $Z' \neq \mathit{false}$.*

Any simulation graph is a discrete graph, hence cycles and lassos are defined in the standard way. We use the recursive application of the transition relation $\mathcal{T}_{\mathcal{Z}}$: $s \Rightarrow^+ s'$ iff there is a path in $SG(\mathbb{B})$ from s to s', or $s \Rightarrow^* s'$ if possibly $s = s'$. An infinite run in $SG(\mathbb{B})$ is an infinite sequence of states $\pi = s_1 s_2 \ldots$, s.t. $s_i \Rightarrow s_{i+1}$ for all $i \geq 1$, it is accepting if it contains infinitely many accepting states. If $SG(\mathbb{B})$ is finite, any infinite path from s_0 defines a lasso: $s_0 \Rightarrow^* s \Rightarrow^+ s$. For example, the infinite run of the TBA in Fig. 1 is: $s_0 \Rightarrow^* s_0 \Rightarrow^+ s_0$.

Definition 8 (A TBA's language under Zone Semantics). *The language accepted by a TBA $\mathbb{B}$ under the zone semantics, denoted $\mathcal{L}(SG(\mathbb{B}))$, is the set of infinite runs $\pi = s_0 s_1 s_2 \ldots$ s.t. there exists an infinite set of indices s.t. $s_i \in \mathcal{F}$.*

Because there are infinitely many zones, the state space of $SG(\mathbb{B})$ may also be infinite. To bound the number of zones, *extrapolation* methods combine all zones which a given TBA $\mathbb{B}$ cannot distinguish. For example, k-extrapolation [4] finds the largest upper bound k in the guards and invariants of $\mathbb{B}$, and extrapolates all bounds in the zones $\mathcal{Z}$ that exceed this value, while LU-extrapolation uses both the maximal lower bound l and the maximal upper bound u [5]. Extrapolation can be refined on a per-clock basis [5], and on a per-location basis [4].

Definition 9. *An x-abstraction over a simulation graph $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ is a mapping $\alpha_x : \mathcal{S}_{\mathcal{Z}} \rightarrow \mathcal{S}_{\mathcal{Z}}$ s.t. if $\alpha((\ell, Z)) = (\ell, Z')$ then $\ell = \ell'$ and $Z \subseteq Z'$. If the image of an abstraction α_x is finite, we call it a finite abstraction.*

Definition 10. *An abstraction α_x over a zone transition system $SG(\mathbb{B}) = (\mathcal{S}_\mathcal{Z}, s_0, \mathcal{T}_\mathcal{Z})$ induces a zone transition system $SG_x(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}_x}, \alpha_x(s_0), \mathcal{T}_{\mathcal{Z}_x})$ where:*

- *$\mathcal{S}_{\mathcal{Z}_x} = \{\alpha_x(s) | s \in \mathcal{S}_\mathcal{Z}\}$ is the set of states, s.t. $\mathcal{S}_{\mathcal{Z}_x} \subseteq \mathcal{S}_\mathcal{Z}$,*
- *$\alpha_x(s_0)$ is the initial state, and*
- *$(s, s') \in \mathcal{T}_{\mathcal{Z}_x}$ if $(s, s'') \in \mathcal{T}_\mathcal{Z}$ and $s' = \alpha(s'')$, is the transition relation.*

We call an abstraction α_x an *extrapolation* if there exists a simulation relation R s.t. if $\alpha_x((\ell, Z)) = (\ell, Z')$ then for all $v' \in Z'$ there exist a $v \in Z$ s.t. $v' R v$ [26]. This means extrapolations do not introduce behaviour that the un-extrapolated system cannot simulate. The abstraction defined by k-extrapolation is given by α_k, while the abstraction defined by LU-extrapolation is called α_{lu}. Hence, α_k and α_{lu} induce the finite simulation graphs $SG_k(\mathbb{B})$ and $SG_{lu}(\mathbb{B})$

Subsumption abstraction. While $SG_k(\mathbb{B})$ and $SG_{lu}(\mathbb{B})$ are finite, their size is still exponential in the number of clocks. Therefore, we turn to the coarser inclusion/ subsumption abstraction of [15], hereafter denoted *subsumption abstraction*. We extend the notion of subsumption to states: a state $s = (\ell, Z) \in \mathcal{S}_\mathcal{Z}$ is *subsumed* by another $s' = (\ell', Z')$, denoted $s \sqsubseteq s'$, when $\ell = \ell'$ and $Z \subseteq Z'$. Let $\mathcal{R}(SG(\mathbb{B})) = \{s | s_0 \Rightarrow^* s\}$ denote the set of *reachable states* in $SG(\mathbb{B})$.

Proposition 1 ($\sqsubseteq$ is a simulation relation). *If $(\ell, Z_1) \sqsubseteq (\ell, Z_2)$ and $(\ell, Z_1) \Rightarrow (\ell', Z_1')$ then there exists Z_2' s.t. $(\ell, Z_2) \Rightarrow (\ell', Z_2')$ and $(\ell', Z_1') \sqsubseteq (\ell', Z_2')$.*

Proof. By the definition of $\sqsubseteq$, and the fact that $\Rightarrow$ is monotone w.r.t $\subseteq$ of zones.

Definition 11 (Subsumption abstraction [15]). *A subsumption abstraction $\alpha_\sqsubseteq$ over a zone transition system $SG(\mathbb{B}) = (\mathcal{S}_\mathcal{Z}, s_0, \mathcal{T}_\mathcal{Z})$ is a total function $\alpha_\sqsubseteq : \mathcal{R}(SG(\mathbb{B})) \to \mathcal{R}(SG(\mathbb{B}))$ s.t. $s \sqsubseteq \alpha(s)$*

Note the subsumption abstraction is defined only over the reachable state space, and is *not* an extrapolation, because it might introduce extra transitions that the unabstracted system cannot simulate. Typically α is constructed on-the-fly during analysis, only abstracting to states that are already found to be reachable. This makes its performance depend heavily on the search order, as finding "large" states quickly can make the abstraction coarser [14].

Property preservation under abstractions. We now consider the preservation by the abstractions above of the property of *location reachability* (a location ℓ is reachable if a state $(\ell, \ldots)$ is reachable in a transition system) and that of Büchi emptiness.

Proposition 2. *For any of the abstractions α_x: α_k [15,11], α_{lu} [5], $\alpha_k \circ \alpha_\sqsubseteq$ [15], and $\alpha_{lu} \circ \alpha_\sqsubseteq$ [5], it holds that ℓ is reachable in $\mathcal{TS}_v^\mathbb{B} \iff \ell$ is reachable in $SG_x(\mathbb{B})$*

Proposition 3. *For any finite extrapolation [26] α_x, e.g. the abstractions α_k [28] and α_{lu} [26] it holds that $\mathcal{L}(\mathbb{B}) = \emptyset \iff \mathcal{L}(SG_x(\mathbb{B})) = \emptyset$*

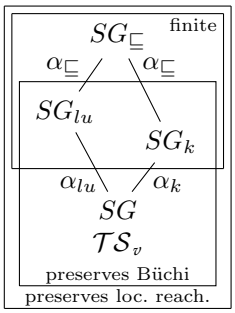


Fig. 3. Abstractions.

From hereon we will denote any finite extrapolation as α_{fin}, and the associated simulation graph $SG_{fin}(\mathbb{B})$. To denote that this graph can be generated *on-the-fly* [30,2,15], we use a NEXT-STATE(s) function which returns the set of successor states for s: $\{s' \in \mathcal{S}_{\mathcal{Z}_{fin}} \mid s \Rightarrow s'\}$.

As a result of Prop. 3 we can focus on finding accepting runs in $SG_{fin}(\mathbb{B})$. Because it is finite, any such run is represented by lassos: $s_0 \Rightarrow s \Rightarrow^+ s$. Tripakis [28] poses the question of whether $\alpha_{\sqsubseteq}$ can be used to check Büchi emptiness. We will investigate this further in the next section.

3 Preservation of Büchi Emptiness under Subsumption

The current section, investigates what properties are preserved by a subsumption abstraction $\alpha_{\sqsubseteq}$, when applied on a finite simulation graph obtained by an extrapolation, α_{fin}, in the following, denoted as $SG_{\sqsubseteq}(\mathbb{B}) = (SG_{fin \circ \sqsubseteq}(\mathbb{B}))$.

Prop. 5 shows that subsumption abstraction preserves Büchi emptiness in one direction. Unfortunately, an accepting cycle in $SG_{\sqsubseteq}(\mathbb{B})$ is not always reflected in $SG_{fin}(\mathbb{B})$, as Fig. 4 illustrates. The figure visualises $SG_{\sqsubseteq}(\mathbb{B})$ by drawing subsumed states inside subsuming states (e.g. $s_3 \sqsubseteq s_1$). However, subsumption introduces strong properties on paths and cycles to which we devote the rest of the current section. In the subsequent section, we exploit these properties to improve an algorithm that implements the TBA emptiness check.

Proposition 4. *For all abstractions α_x, $s \in \mathcal{F} \Leftrightarrow \alpha_x(s) \in \mathcal{F}$ (by Def. 9).*

Proposition 5. *An $\alpha_{\sqsubseteq}$ abstraction is safe w.r.t. Büchi emptiness:*

$$\mathcal{L}(\mathbb{B}) \neq \emptyset \implies \mathcal{L}(SG_{\sqsubseteq}(\mathbb{B})) \neq \emptyset$$

Proof. If $\mathcal{L}(\mathbb{B}) \neq \emptyset$, there must be an infinite accepting path π. This path is inscribed [28] in $SG_{fin}(\mathbb{B})$, and because $\sqsubseteq$ is a simulation relation a similar path exists in $SG_{\sqsubseteq}(\mathbb{B})$. □

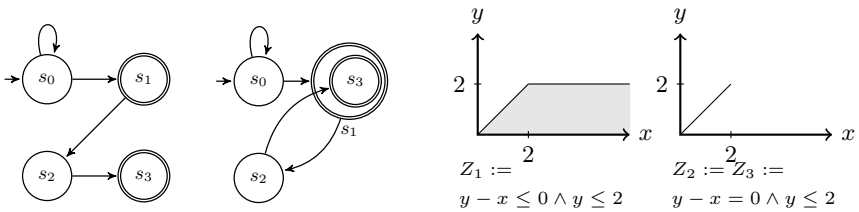


Fig. 4. The state space $SG_{\sqsubseteq}(\mathbb{B})$ of the model in Fig. 1 with $\ell_1 \in \mathcal{F}$ contains 4 states (shown on the left): s_0, $s_1 = (\ell_1, Z_1)$, $s_2 = (\ell_2, Z_2)$ and $s_3 = (\ell_1, Z_3)$. The graphical representation of the zones Z_1–Z_3 (right) reveals that $Z_3 \subseteq Z_1$ and hence $s_3 \sqsubseteq s_1$. As $s_3 \sqsubseteq s_1$ and both are reachable, a subsumption abstraction is allowed to map $\alpha_{\sqsubseteq}(s_3) = s_1$, introducing a cycle $s_1 \Rightarrow s_2 \Rightarrow s_1$ in $SG_{\sqsubseteq}(\mathbb{B})$.

Lemma 1 (Accepting cycles under $\sqsubseteq$). *If $SG_{fin}(\mathbb{B})$ contains states s, s' s.t. s leads to an accepting cycle and $s \sqsubseteq s'$, then s' leads to an accepting cycle.*

Proof. We have that $s \Rightarrow^* t \Rightarrow^+ t$, and because $\sqsubseteq$ is a simulation relation we have the existence of a state x s.t. $t \sqsubseteq x$:

$$\begin{array}{ccccccc} s' & \Rightarrow^* & t' & \Rightarrow & \cdots & \Rightarrow & x \\ \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| \\ s & \Rightarrow^* & t & \Rightarrow & \cdots & \Rightarrow & t \end{array}$$

From x, we again have a similar path, to some x'. This sequence will eventually repeat some x'', because $SG_{fin}(\mathbb{B})$ is finite. It follows that all states in $x'' \Rightarrow^+ x''$ subsume states in $t \Rightarrow^+ t$, hence the former cycle is also accepting (Prop. 4). □

Lemma 2 (Paths under $\sqsubseteq$). *If $SG_{fin}(\mathbb{B})$ contains a path $s \Rightarrow^+ s'$ containing an accepting state and $s \sqsubseteq s'$, then s leads to an accepting cycle.*

Proof. Because $\sqsubseteq$ is a simulation relation we have that $s \Rightarrow^+ s'$ and $s \sqsubseteq s'$ implies the existence of some t such that $s' \Rightarrow^+ t$ and $s' \sqsubseteq t$. From t, we again obtain a similar path to some t', s.t. $t \sqsubseteq t'$. Because $SG_{fin}(\mathbb{B})$ is finite, the sequence of t's will eventually repeat some element x, s.t. $x \Rightarrow^+ \cdots \Rightarrow^+ x$.

$$\begin{array}{ccccccccccc} s' & \Rightarrow^+ & t & \Rightarrow^+ & t' & \Rightarrow^+ & \cdots & \Rightarrow^+ & t'' & \Rightarrow^+ & x \\ \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \| \\ s & \Rightarrow^+ & s' & \Rightarrow^+ & t & \Rightarrow^+ & \cdots & \Rightarrow^+ & x & \Rightarrow^+ & x \end{array}$$

This gives us the lasso $s \Rightarrow^* x \Rightarrow^+ x$. It also follows that all states in $x \Rightarrow^+ x$ subsume states in $s \Rightarrow^+ s'$, hence the former cycle is accepting (Prop. 4). □

4 Timed Nested Depth-First Search with Subsumption

In the current section, we extend the classic linear-time NDFS [12,27] algorithm to exploit subsumption. The algorithm detects accepting cycles, the absence of which implies Büchi emptiness. It is correct for the graph $SG_{fin}(\mathbb{B})$ according to Prop. 3. In the following, with *soundness*, we mean that when NDFS reports a cycle, indeed an accepting cycle exists in the graph, while completeness indicates that NDFS always reports an accepting cycle if the graph contains one.

The NDFS algorithm in Alg. 1 consists of an outer DFS (*dfsBlue*) that sorts accepting states s in DFS *postorder*. And an inner DFS (*dfsRed*) that searches for cycles over each s, called the *seed*. States are maintained in 3 colour sets:

1. *Blue*, states *explored* by *dfsBlue*,
2. *Cyan*, states on the stack of *dfsBlue* (*visited* but not yet explored), which are used by *dfsRed* to close cycles over s early at l.8 [27], and
3. *Red*, visited by *dfsRed*.

Alg. 1 maintains a few strong invariants, which are mentioned informally by [12,27], but for which we include a formal proof in Sec. A:

Alg. 1 NDFS

```
1: procedure ndfs( )
2:     Cyan := Blue := Red := ∅
3:     dfsBlue(s0)
4:     report no cycle
5: procedure dfsRed(s)
6:     Red := Red ∪ {s}
7:     for all t in NEXT-STATE(s) do
8:         if (t ∈ Cyan) then report cycle
9:         if (t ∉ Red) then dfsRed(t)
10: procedure dfsBlue(s)
11:     Cyan := Cyan ∪ {s}
12:     for all t in NEXT-STATE(s) do
13:         if t ∉ Blue ∧ t ∉ Cyan then
14:             dfsBlue(t)
15:     if s ∈ F then
16:         dfsRed(s)
17:     Blue := Blue ∪ {s}
18:     Cyan := Cyan \ {s}
```

I0 At l.13 all red states are blue (Corollary 2).
I1 The only accepting state visited by *dfsRed* is the seed (Corollary 3).
I2 Outside of *dfsRed*, red states do not lead to accepting cycles (Corollary 4).
I3 A sufficient postcondition for *dfsRed* is that it concludes non-reachability of cyan states in the non-red graph, adding all reachable states to *Red* (Corollary 5).

We now try to employ subsumption on the different colours to prune the searches, even though we cannot use it on all colours as $SG_{\sqsubseteq}(\mathbb{B})$ introduces additional cycles as Fig. 4 showed. To express subsumption checks on sets we write $s \sqsubseteq S$, meaning $\exists s' \in S\colon s \sqsubseteq s'$. And $S \sqsubseteq s$, meaning $\exists s' \in S\colon s' \sqsubseteq s$. At several places in Alg. 1 we might apply subsumption:

1. On cyan for cycle detection:
   (a) $t \sqsubseteq \mathit{Cyan}$ at l.8, or
   (b) $\mathit{Cyan} \sqsubseteq t$ at l.8.
2. On *dfsBlue*, by replacing $t \notin \mathit{Blue} \wedge t \notin \mathit{Cyan}$ at l.13 with $t \not\sqsubseteq \mathit{Blue} \cup \mathit{Cyan}$.
3. On the blue set (explored states), by replacing $t \notin \mathit{Blue}$ at l.13 with $t \not\sqsubseteq \mathit{Blue}$.
4. On *dfsRed*, by replacing $t \notin \mathit{Red}$ at l.9 with $t \not\sqsubseteq \mathit{Red}$.

Applying subsumption on cyan for cycle detection as in item 1a makes the algorithm unsound as cycles in $SG_{\sqsubseteq}(\mathbb{B})$ are not always reflected in $SG_{\mathit{fin}}(\mathbb{B})$ (Fig. 4). There is also no hope of "rewinding" the algorithm upon detecting an accepting cycle that does not exist in the underlying $SG_{\mathit{fin}}(\mathbb{B})$ without losing its linear-time complexity, as the number of cycles can be exponential in the size of $SG_{\sqsubseteq}(\mathbb{B})$.

If, on the other hand, we prune the blue search as in item 2, the algorithm becomes incomplete. Fig. 5 shows a run of the modified NDFS on an $SG_{\mathit{fin}}(\mathbb{B})$ with cycle $s_3 \Rightarrow s_2 \Rightarrow s_3$. The *dfsBlue* backtracked over s_2 as $s_3 \sqsubseteq s_1$ and $s_1 \in \mathit{Cyan}$. The *dfsRed* now launched from s_1, will however continue to visit s_3, while missing the cycle as s_2 is not cyan. We also observe that I1 is violated, indicating that the postorder on accepting states (s_3 before s_1) is lost.

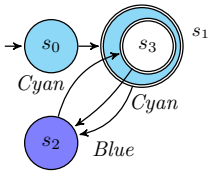
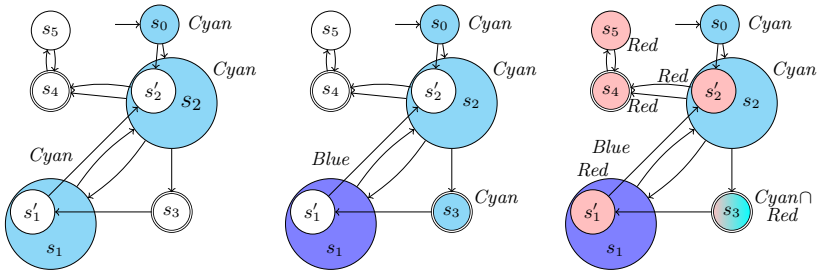


Fig. 5. Counter example to subsumption on *Blue* and *Cyan* (item 2).

It is tempting therefore to use subsumption only on blue as in item 3. However, Fig. 6 shows an "animation" of a run with the modified NDFS which is incomplete. In the example, the state s_1 is first backtracked in the blue search



(a) *dfsBlue*(s_1) (b) *dfsBlue*(s_3) (c) *dfsRed* from s_3

Fig. 6. Counter example to subsumption on *Blue*

as all successors are cyan (left). The state s_1 is then marked as blue (middle). The subsequent red search from s_3 subsumes the cyan stack (s_2) and visits accepting state s_4, violating I1 and missing the accepting cycle $s_4 \Rightarrow s_5 \Rightarrow s_4$.

A viable option however is to use inverse subsumption on cyan as described by item 1b. According to Lemma 1, a state that subsumes a state on the cyan stack leads to a cycle. And as the only goal of the red search is to find a cyan state (to close an accepting cycle over the seed), it does not rely on DFS (I3). Thus we may as well use subsumption in the red search as in item 4. By definition (Def. 11), $SG_{\sqsubseteq}(\mathbb{B})$ contains a "larger" state for all reachable states in $SG_{fin}(\mathbb{B})$. So in combination with item 1b that therefore suffices to find all accepting cycles.

Due to a strong invariant on red states during the blue search (I2), which states that they do not lead to accepting cycles, we can use red states to prune the blue search. We do this by strengthening the condition on l.13 to $t \notin \mathit{Blue} \cup \mathit{Cyan} \cup \mathit{Red}$. However, I0 tells use that this is no use since $\mathit{Red} \subseteq \mathit{Blue}$. Luckily, also states subsumed by red do not lead to accepting cycles (by contraposition of Lemma 1), so we can use subsumption again: $t \notin \mathit{Blue} \cup \mathit{Cyan} \wedge s \not\sqsubseteq \mathit{Red}$. The benefit of this can be illustrated using Fig. 4. Once *dfsBlue* backtracks over s_1, we have $s_1, s_2, s_3 \in \mathit{Red}$ by *dfsRed* at l.16. Now any hypothetical other path from s_0 to a state subsumed by these red states can be ignored.

Alg. 2 shows a version of NDFS with all correct improvements.Notice that I2 and I3 are sufficient to conclude correctness of these modifications.

Alg. 2 NDFS with subsumption on red, cycle detection, and red prune of *dfsBlue*.

```
1: procedure ndfs( )
2:     Cyan := Blue := Red := ∅
3:     dfsBlue(s_0)
4:     report no cycle
5: procedure dfsRed(s)
6:     Red := Red ∪ {s}
7:     for all t in NEXT-STATE(s) do
8:         if (Cyan ⊑ t) then report cycle
9:         if (t ⋢ Red) then dfsRed(t)
10: procedure dfsBlue(s)
11:     Cyan := Cyan ∪ {s}
12:     for all t in NEXT-STATE(s) do
13:         if t ∉ Blue ∧ t ∉ Cyan∧t ⋢ Red then
14:             dfsBlue(t)
15:     if s ∈ F then
16:         dfsRed(s)
17:     Blue := Blue ∪ {s}
18:     Cyan := Cyan \ {s}
```

5 Multi-Core CNDFS with Subsumption

CNDFS [17] is a parallel algorithm for checking Büchi emptiness [17]. By Prop. 3, it is correct for SG_{fin}. In the current section, we extend CNDFS with subsumption, in a similar way as we have done for the sequential NDFS in the previous section.

In CNDFS (Alg. 3), each worker thread i runs a seemingly independent $dfsBlue_i$ and $dfsRed_i$, with a local stack colour $Cyan_i$, and its own random successor ordering (indicated by the subscript i of the NEXT-STATE function). Their search space is however pruned by sharing the colours *Blue* and *Red* globally.

The main problem that CNDFS has to solve is the loss of postorder on the accepting states due to the shared blue color (similar to the effects of item 3 as illustrated in Fig. 6). In the previous section, we have seen that a loss of postorder may cause *dfsRed* to visit non-seed accepting states, i.e. I1 is violated. CNDFS demonstrates that repairing the latter *dangerous situation* is sufficient to preserve correctness [17, Sec. 3].

To detect a dangerous situation, CNDFS collects the states visited by $dfsRed_i$ in a set $\mathcal{R}_i$ at l.7. After completion of $dfsRed_i$, the algorithm then checks $\mathcal{R}_i$ for non-seed accepting states at l.21. By simply waiting for these states to become red, the dangerous situation is resolved as the blue state that caused the situation was always placed by some other worker, which will eventually continue [17, Prop. 3]. Once the situation is detected to be resolved, all states from the local $\mathcal{R}_i$ are added to *Red* at l.22.

CNDFS maintains similar invariants as NDFS:

I2' Red states do not lead to accepting cycles (Lemma 1 and Prop. 2 in [17]).
I3' same as I3 but adding states to $\mathcal{R}_i$ (Lemma 2 generalises $\mathcal{R}_i$ in [17]).

Because these invariants are as strong or stronger than I2 and I3, we can use subsumption in a similar way as for NDFS. Alg. 3 underlines the changes to algorithm w.r.t. Alg. 2 in [17]. We additionally have to extend the waiting procedure to include subsumption at l.21, because the use of subsumption in $dfsRed_i$ can cause other workers to find "larger" states.

Alg. 3 CNDFS with subsumption

```
1:  procedure cndfs(P)
2:      Blue := Red := ∅
3:      forall i in 1..P do Cyan_i := ∅
4:      dfsBlue_1(s_0) || .. || dfsBlue_P(s_0)
5:      report no cycle
6:  procedure dfsRed_i(s)
7:      R_i := R_i ∪ {s}
8:      for all t in NEXT-STATE_i(s) do
9:          if Cyan ⊑ t then cycle
10:         if t ∉ R_i ∧ t ⋢ Red then
11:             dfsRed_i(t)
12: procedure dfsBlue_i(s)
13:     Cyan_i := Cyan_i ∪ {s}
14:     for all t in NEXT-STATE_i(s) do
15:         if t ∉ Cyan_i ∪ Blue ∧ t ⋢ Red then
16:             dfsBlue(t)
17:     Blue := Blue ∪ {s}
18:     if s ∈ F then
19:         R_i := ∅
20:         dfsRed(s)
21:         await ∀s' ∈ R_i ∩ F \ {s}: s' ⊑ Red
22:         forall s' in R_i do Red := Red ∪ s'
23:     Cyan_i := Cyan_i \ {s}
```

In the next section, we will benchmark Alg. 3 on timed models. An important property that the algorithm inherits from CNDFS, is that its *runtime* is linear in the size of the input graph N. However, in the worst case, all workers may visit the same states. Therefore, the complexity of the amount of *work* that the algorithm performs (or the amount of power it consumes) equals $N \times P$, where P is the number of processors used. The randomized successor function NEXT-STATE$_i$ however ensures that this does not happen for most practical inputs. Experiments on over 300 examples confirmed this [17, Sec. 4], making CNDFS the current state-of-the-art parallel model checking algorithm.

6 Experimental Evaluation

To evaluate the performance of the proposed algorithms experimentally, we implemented Alg. 1, Alg. 2, CNDFS and Alg. 3 in LTSMIN 2.0[3]. The opaal [13] tool[4] functions as a front-end for UPPAAL models. Previously, we demonstrated scalable multi-core reachability for timed automata using these tools [14].

Experimental setup. We benchmarked[5] on a 48-core machine (a four-way AMD Opteron™ 6168) with a varying number of threads, averaging results over 5 repetitions. We consider the following models and LTL properties:

`csma`[6] is a protocol for Carrier Sense, Multiple-Access with Collision Detection with 10 nodes. We verify the property that if there is a collision, eventually the bus will be active again: $\Box$`((P0=bus_collision1)` $\Longrightarrow$ $\Diamond$`(P0=bus_active))`.

`fischer-1/2`[7] implements a mutual exclusion protocol; a canonical benchmark for timed automata, with 10 nodes. As in [26], we use the property (1): $\neg$`((`$\Box\Diamond$`k=1)`$\vee$`(`$\Box\Diamond$`k=0))`, where `k` is the number of processes in their critical section. We also add a weak fairness property (2): $\Box$`((`$\Box$`P_1=req)` $\Longrightarrow$ `(`$\Diamond$`P_1=cs))`, i.e. processes requesting infinitely often should eventually be served.

`fddi`[6] models a token ring system as described in [10], where a network of 10 stations are organised in a ring and can hand back the token in a synchronous or asynchronous fashion. We verify the property from [10] that every station will eventually send asynchronous messages: $\Box$`(`$\Diamond$`(ST1=station_z_sync))`.

`train-gate`[6] models a railway interlocking, with 10 trains. Trains drive onto the interconnect until detected by sensors. There they wait until receiving a signal for safe crossing. The property prescribes that each approaching train eventually should be serviced: $\Box$`(Train_1=Appr` $\Longrightarrow$ `(`$\Diamond$`Train_1=Cross))`.

The following command-line was used to start the LTSMIN tool: opaal2lts-mc –strategy=[A] –ltl-semantics=textbook –ltl=[f] -s28 –threads=[P] -u[0,1] [m]. This runs algorithm A on the cross-product of the model m with the Büchi of formula f. It uses a fixed hash table of size 2^{28} and P threads, and either subsumption (-u1)

[3] Available as open source at: http://fmt.cs.utwente.nl/tools/ltsmin

[4] Available as open source at: http://opaal-modelchecker.com

[5] All results are available at: http://fmt.cs.utwente.nl/tools/ltsmin/cav-2013

[6] From http://www.it.uu.se/research/group/darts/uppaal/benchmarks/

[7] As distributed with UPPAAL.

or not (-u0). The option ltl-semantics switches on textbook LTL semantics as for example defined in [2, Ch. 4] (as opposed to SPIN semantics). To investigate the overhead of CNDFS, we also run the reachability algorithms on this crossproduct, even though this does not make sense from a model checking perspective. To compare effects of the search order on subsumption, we use DFS and BFS.

Note finally, that we are only interested here in full verification, i.e. in LTL properties that are correct w.r.t the system under verification. This is the hardest case as the algorithm has to explore the full simulation graph. To test their on-the-fly nature, we also tried a few incorrect LTL formula for the above models, to which the algorithms all delivered counter examples within a second. But with parallelism this happens almost instantly [17, Sec. 4.2].

Implementation. LTSMIN defines a NEXT-STATE function as part of its PINS interface for language-independent symbolic/parallel model checking [9]. Previously, we extended PINS with subsumption [14]. opaal is used to parse the UPPAAL models and generate C code that implements PINS. The generated code uses the UPPAAL DBM library to implement the simulation graph semantics under *LU-extrapolated zones*. The LTL crossproduct [2] is calculated by LTSMIN.

LTSMIN's multi-core tool [8] stores states in one lockless hash/tree table in shared memory [22,24]. For the timed systems, this table is used to store *explicit state parts*, i.e. the locations and state variables [6]. The DBMs representing zones, here referred to as the *symbolic state parts*, are stored in a separate lockless hash table, while a lockless *multimap* structure efficiently stores full states, by linking multiple symbolic to a single explicit state part [14]. Global colour sets of CNDFS (*Blue* and *Red*) are encoded with extra bits in the multimap, while local colours are maintained in local tables to reduce contention to a minimum.

Hypothesis. CNDFS for plain model checking scaled mostly linearly. In the timed automata setting, several parameters could change this picture. In the first place, the *computational intensity* increases, because the DBM operations use more calculations than in plain model checking. In modern multi-core computers, this algorithm characteristic improves scalability, because it more closely matches the machine's high frequency/bandwidth ratio [22]. On the other hand, the lock granularity increases since a single lock on an explicit state part governs multiple DBMs stored in the multimap [14, Sec. 6.1]. Nonetheless, for multi-core timed reachability, we previously saw that almost linear scalability can still be attained [14, Sec. 7], also when using other model checkers, like UPPAAL, as a base line. On the other hand, the CNDFS algorithm requires more queries on the multimap structure to distinguish the different colour sets.

As for subsumption, we expect that models with more time behavior, i.e. more clocks and constraints, show more reduction via this abstraction. Moreover, it is well-known that reductions under subsumption depend strongly on the exploration order; BFS typically results in better reductions than DFS. Hence the DFS-based CNDFS seems to have a disadvantage. However, it was also shown in [14] that randomised parallel DFS performs as well using subsumption as BFS, which could translate into a benefit for CNDFS as well. But even if it does not, subsumption might still improve the performance of CNDFS relative to itself.

Table 1. Runtimes (sec) and states counts *without* subsumption.

Model	P	$\|L\|$	$\|\mathcal{R}\|_{bfs}$	$\|\mathcal{R}\|_{dfs}$	$\|\mathcal{R}\|_{cndfs}$	$\|V\|_{cndfs}$	$\|\Rightarrow\|_{bfs}$	T_{bfs}	T_{dfs}	T_{cndfs}
csma	1	135449	438005	438005	438005	438005	1016428	26.1	26.2	27.8
csma	48	135449	438005	438005	438005	453658	1016428	1.0	0.9	0.9
fddi	1	119	179515	179515	179515	179515	314684	26.3	26.6	34.2
fddi	48	119	179515	179515	179515	566093	314684	1.6	0.7	2.7
fischer-1	1	521996	4987796	4987796	4987796	4987796	19481530	195.9	196.7	212.2
fischer-1	48	521996	4987796	4987796	4987796	5190490	19481530	4.8	4.6	5.1
fischer-2	1	358901	3345866	3345866	3345866	3345866	10426444	135.8	136.5	145.5
fischer-2	48	358901	3345866	3345866	3345866	3541373	10426444	3.4	3.3	3.7
train-gate	1	119989268	119989268	119989268	119989268	119989268	177201017	1608	1621	1724
train-gate	48	119989268	119989268	119989268	119989268	319766765	177201017	34.9	45.4	145.8

Experimental results. We first compare the algorithms without subsumption. Table 1 shows the runtimes (T), and the number of explicit state parts ($|L|$), full states ($\mathcal{R}$), transitions ($|\Rightarrow|$) in BFS, and visited states in CNDFS, of the different algorithms on the above models. As discussed above, these numbers reflect the established properties of CNDFS: For the sequential runs ($P = 1$), the runtimes show little difference, indicating little overhead in CNDFS. For the parallel runs ($P = 48$) however the number of visited states in CNDFS ($|V|$) can increase due to work duplication.

To investigate the scalability of the timed CNDFS algorithm, we plot the speedups in Fig. 7. Vertical bars represent the standard deviations over the five benchmarks, which is negligible rendering the bars barely visible. Three benchmarks exhibit linear scalability, while train-gate and fddi show a sub-linear, yet still positive, trend. For train-gate, we suspect that this is caused by the structure of the state space. Because fddi also has very few explicit state parts (118), we can attribute this behaviour to the lock contention as discussed above. Increasing lock contention can very well translate into work duplication in CNDFS.

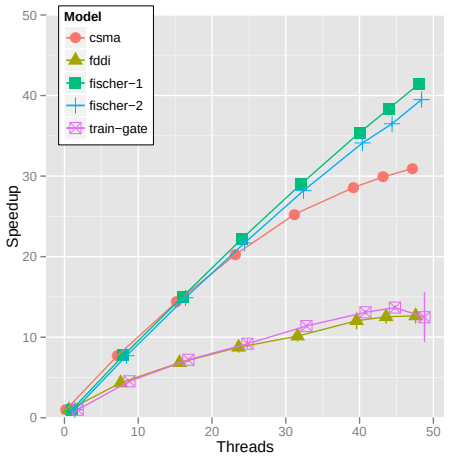


Fig. 7. Speedups in LTSMIN/opaal

Subsumption. Table 2 shows the same data as in Table 1 for benchmarks with subsumption (Alg. 2 and Alg. 3). The number of explicit state parts $|L|$, does not vary among the different algorithms, as location reachability is preserved under subsumption (Prop. 2). Furthermore, the column $|\mathcal{R}|_{bfs}$ confirms the known benefits of reachability under subsumption: fischer has almost 6-fold reductions, while fddi even more than 30-fold. Only the train-gate model yields no reductions. Notice that this closely reflects the ratio $X = |\mathcal{R}|/|L|$; the higher the ratio, the better the reductions, e.g. $X \approx 1500$ for fddi and $X \approx 10$ for fischer. The column $|\mathcal{R}|_{dfs}$ shows the well-known penalty for this search order under subsumption.

Table 2. Runtimes and states counts *with* subsumption (in % relative to Table 1).

Model	P	$\lvert L\rvert$	$\lvert\mathcal{R}\rvert_{bfs}$	$\lvert\mathcal{R}\rvert_{dfs}$	$\lvert\mathcal{R}\rvert_{cndfs}$	$\lvert V\rvert_{cndfs}$	$\lvert\Rightarrow\rvert_{bfs}$	T_{bfs}	T_{dfs}	T_{cndfs}
csma	1	135449	48.7	88.9	58.3	94.7	41.2	41.3	90.3	95.2
csma	48	135449	48.7	77.5	58.3	93.6	41.2	64.5	85.3	97.8
fddi	1	119	3.1	3.4	50.8	53.1	3.4	4.3	4.7	132.3
fddi	48	119	3.1	2.4	50.8	80.1	3.4	51.0	19.5	121.0
fischer-1	1	521996	17.9	72.4	55.2	91.9	27.0	25.6	78.7	97.3
fischer-1	48	521996	17.9	71.1	55.2	95.9	27.0	33.1	79.6	103.0
fischer-2	1	358901	18.6	68.5	77.5	95.8	28.7	27.0	75.3	98.9
fischer-2	48	358901	18.6	62.7	77.5	95.8	28.7	37.4	72.5	98.3
train-gate	1	119989268	100.0	100.0	100.0	100.0	100.0	100.6	100.6	104.3
train-gate	48	119989268	100.0	100.0	100.0	100.0	100.0	101.7	83.5	83.1

CNDFS also benefits from subsumption, although not as much as the reachability algorithms, e.g. only a 2 fold reduction in reachable states for fddi, fischer and csma. In the case of fischer-1, the CNDFS reductions lie between those obtained by BFS and by DFS, confirming the expected benefits of randomised parallel DFS. However, comparing the differences with BFS reductions, we come to suspect that Alg. 3 might still be improved. As currently only the red states can attribute to subsumption reduction, there are likely some 'large' states that are not coloured red. For all of these models, we measured that about 20%–50% of all reachable states are coloured red (only fischer-2 has no red states). Unfortunately, we do not witness an improved reduction for the parallel runs of CNDFS (as for parallel DFS), while the scalability remains unaffected.

7 Conclusions

We implemented the first parallel model checking algorithm for liveness properties on timed systems. We also contributed to solving the open problem [28] of using the inclusion abstraction in this field, while combining it with parallelism. Experimentally, we established that these techniques have their own merits: most models with many discrete states yield great speedups of up to 40 on a 48 core machine, while models with more symbolic states can benefit from abstraction, which halves the state space in one example. There is however also room for improvement: The use of a non-blocking algorithm [14, Sec. 6.4] could further improve speedups and subsumption might be exploitable in still different ways.

References

1. R. Alur and D. L. Dill. A Theory of Timed Automata. *TCS*, 126(2):183–235, 1994.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
3. G. Behrmann. Distributed Reachability Analysis in Timed Automata. *STTT*, 7(1):19–30, 2005.
4. G. Behrmann, P. Bouyer, E. Fleury, and K. Larsen. Static Guard Analysis in Timed Automata Verification. *TACAS*, pages 254–270, 2003.
5. G. Behrmann, P. Bouyer, K.G. Larsen, and R. Pelánek. Lower and Upper Bounds in Zone Based Abstractions of Timed Automata. *TACAS*, pages 312–326, 2004.

6. G. Behrmann, A. David, and K.G. Larsen. A Tutorial on Uppaal. In *FMDRTS*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.
7. J. Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala University, 2002.
8. F.I. van der Berg and A.W. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In *PASM/PDMC*, ENTCS. Elsevier, 2012.
9. S. C. C. Blom, J. C. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *CAV*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.
10. A. Bouajjani, S. Tripakis, and S. Yovine. On-the-Fly Symbolic Model Checking for Real-Time Systems. In *18th IEEE*, RTSS, pages 25–34. IEEE, 1997.
11. P. Bouyer. Forward Analysis of Updatable Timed Automata. *Formal Methods in System Design*, 24(3):281–320, 2004.
12. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *CAV*, volume 531 of *LNCS*, pages 233–242. Springer, 1990.
13. A.E. Dalsgaard, R.R. Hansen, K. Jørgensen, K.G. Larsen, M.C. Olesen, P. Olsen, and J. Srba. opaal: A Lattice Model Checker. In *NFM*, volume 6617, chapter LNCS, pages 487–493. Springer, 2011.
14. A.E. Dalsgaard, A.W. Laarman, K.G. Larsen, M.C. Olesen, and J.C. van de Pol. Multi-Core Reachability for Timed Automata. In *Formats*, LNCS. Springer, 2012.
15. C. Daws and S. Tripakis. Model Checking of Teal-Time Reachability Properties Using Abstractions. In *TACAS*, volume 1384, pages 313–329. Springer, 1997.
16. D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *AVMFSS*, volume 407, pages 197–212. Springer, 1989.
17. S. Evangelista, A.W. Laarman, L. Petrucci, and J.C. van de Pol. Improved Multi-Core Nested Depth-First Search. In *ATVA*, LNCS. Springer, 2012.
18. S. Evangelista, L. Petrucci, and S. Youcef. Parallel Nested Depth-First Searches for LTL Model Checking. In *ATVA*, LNCS 6996, pages 381–396. Springer, 2011.
19. F. Herbreteau and B. Srivathsan. Efficient On-the-Fly Emptiness Check for Timed Büchi Automata. In *ATVA*, volume 6252, pages 218–232. Springer, 2010.
20. A.W. Laarman, R. Langerak, J.C. van de Pol, M. Weber, and A. Wijs. Multi-Core Nested Depth-First Search. In *ATVA*, LNCS 6996, pages 321–335. Springer, 2011.
21. A.W. Laarman and J.C. van de Pol. Variations on Multi-Core Nested Depth-First Search. In *PDMC*, pages 13–28, 2011.
22. A.W. Laarman, J.C. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *FMCAD*. IEEE Computer Society, 2010.
23. A.W. Laarman, J.C. van de Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *NFM*, LNCS 6617, pages 506–511. Springer, 2011.
24. A.W. Laarman, J.C. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In *SPIN*, LNCS, pages 38–56. Springer, 2011.
25. K. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *STTT*, 1:134–152, '97.
26. G. Li. Checking Timed Büchi Automata Emptiness using LU-Abstractions. In *FORMATS*, volume 5813 of *LNCS*, pages 228–242. Springer, 2009.
27. S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *TACAS*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.
28. S. Tripakis. Checking timed Büchi Automata Emptiness on Simulation Graphs. *TOCL*, 10(3):15, 2009.
29. S. Tripakis, S. Yovine, and A. Bouajjani. Checking Timed Büchi automata Emptiness Efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.
30. M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 332–344. IEEE, 1986.

A Correctness Proof and Corollaries for NDFS

We use the modal operator in e.g. $s \in \Box X$ to express that NEXT-STATE$(s) \subseteq X$. Whenever a worker reaches a **report** statement, it terminates after reporting. We prove our propositions by doing induction over the lines in the algorithm. If, for example, we prove that successors of blue states are either blue or cyan, it suffices to show that this holds (1) before the algorithm starts (when all states are white) and (2) after execution of each line where either blue or cyan is modified, assuming that it held before. Theorem 2 and Theorem 3 prove Alg. 1's soundness and completeness. We state some additional corollaries at the end, used to explain the algorithm more intuitively.

Lemma 3. *In Alg. 1 holds: $Blue \cap Cyan = \emptyset$ except for l.18. States are never removed from $Cyan \cup Blue$.*

Lemma 4. *In Alg. 1, successors of blue states are blue or cyan: $Blue \subseteq \Box(Blue \cup Cyan)$.*

Proof. Initially, *Blue* is empty and the proposition holds. States are coloured blue at l.17, at which point all successors t have been considered at l.13–14. If $t \notin Blue \cup Cyan$ then $dfsBlue(t)$ is executed adding t to *Blue* at l.17. States are only removed from cyan at l.18, but after being coloured blue at l.17. □

Corollary 1. *Lemma 4 holds for s at l.17, so at l.15 also: $s \in \subseteq \Box(Blue \cup Cyan)$.*

Lemma 5. *The red search visits only blue states except for the (cyan) seed.*

Proof. The search at l.16 starts at the seed $s \in \mathcal{F} \cap Cyan$ (by l.11 and Lemma 3), with $s \in \Box(Blue \cup Cyan)$ (Corollary 1). Before the recursive *dfsRed* call at l.9, if a successor t of s is *Cyan*, Alg. 1 terminates at l.8. Hence, $t \in Blue$ at l.9. □

Lemma 6. *Alg. 1, when dfsRed is finished, red states have blue successors: $Red \subseteq \Box Blue$*

Proof. By Corollary 1, we have $s \in \Box Blue \cup Cyan$ at l.16. By Lemma 4 and Lemma 5, we have $t \in \Box Blue \cup Cyan$ at l.9. So always $s \in \Box Blue \cup Cyan$ at l.6. If a successor $t \in Cyan$ at l.8, Alg. 1 would have terminated, so: $s \in \Box Blue$ (Lemma 3). □

Lemma 7. *In Alg. 1, invariantly: $Blue \cap \mathcal{F} \subseteq Red$.*

Proof. A state $s \in \mathcal{F}$ is marked blue at l.17. There, we have $s \in Red$ because l.6 happens before l.17 if $s \in \mathcal{F}$ (see l.15). □

Lemma 8. *In Alg. 1, when dfsRed is finished, then $Red \subseteq \Box Red$.*

Proof. Follows from induction on *dfsRed* calls at l.16, which perform reachability on non-red states. □

Lemma 9. *Alg. 1 ensures that blue accepting states never lie on accepting cycles: $Blue \cap \mathcal{F} \cap Cycle = \emptyset$.*

Proof. At l.17, a state $s \in \mathcal{F}$ is coloured blue. By Lemma 7, we must already have $s \in \mathit{Red}$. Also, $s \in \mathit{Cyan}$ by l.11 and Lemma 3. Assume towards a contradiction that a cycle $s \Rightarrow^+ s$ exists. By induction on the length of the cycle, using $\mathit{Red} \subseteq \Box\mathit{Red}$ from Lemma 8, the immediate predecessor t of s has to be red. However, with $s \in \mathit{Cyan}$, this contradicts Lemma 6. □

Theorem 1 (Termination). *Alg. 1 always terminates with a report.*

Proof. By Lemma 3, the colour sets continuously grow, reducing the (finite) number of successors that have to be considered at l.14 and l.9. Therefore, both *dfsRed* and *dfsBlue* eventually return, including the initial *dfsBlue* call at l.3, generating a report at l.4. Unless a cycle is reported earlier at l.8. □

Theorem 2 (Soundness). report cycle $\implies \exists a \in \mathcal{F} \colon s_0 \Rightarrow^* a \Rightarrow^+ a$

Theorem 3 (Completeness). report no cycle $\implies \nexists a \in \mathcal{F} \colon s_0 \Rightarrow^* a \Rightarrow^+ a$

Proof. At l.4, $s_0 \in \mathit{Blue}$ by l.17, and $\mathit{Cyan} = \emptyset$ (always l.18 after l.11). By Lemma 4, all states are blue, hence no accepting cycle exists by Lemma 9. □

The following corollaries illustrate the working of the NDFS algorithm more intuitively: For each accepting state (Corollary 3), a red search is launched to find a path back to the cyan stack closing the accepting cycle. Th search may ignore states visited by previous red searches (red states) as these do not lead to accepting cycles Corollary 4, making the algorithm linear.

Corollary 2. *At l.13, $\mathit{Red} \subseteq \mathit{Blue}$.*

Proof. Lemma 5 and l.17 happens after the initial *dfsRed* call at l.16. □

Corollary 3. *The red search visits one single accepting state: the seed.*

Proof. The search starts at l.16 in a cyan seed s and then visits only blue states at l.9 (Lemma 5). Assuming that it also visits some $t \in \mathcal{F}$ with $s \neq t$, we have $t \in \mathit{Red}$ by Lemma 7 contradicting the condition $t \notin \mathit{Red}$ at l.9. □

Corollary 4. *Outside of dfsRed, no red state leads to an accepting cycle.*

Proof. Outside of *dfsRed*, we have $\mathit{Red} \subseteq \Box\mathit{Red}$ by Lemma 8. Assume towards a contradiction that there exists a state $s \in \mathit{Red}$ that leads to an accepting cycle. By Lemma 6, we have $\mathit{Red} \subseteq \Box\mathit{Blue}$. By induction on the lasso from s, we learn that the cycle is both blue and red. However, this contradicts Lemma 9. □

Corollary 5. *dfsRed is not dependent on DFS order, it can be implemented with any reachability algorithm (that marks visited states red).*