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

Laarman, Alfons; Olesen, Mads Chr.; Dalsgaard, Andreas; Larsen, Kim G.; Pol, Jaco van de

Published in:
Proceedings of the 25th International Conference on Computer Aided Verification (CAV)

DOI (link to publication from Publisher):
10.1007/978-3-642-39799-8_69

Publication date:
2013

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

Citation for published version (APA):
Laarman, A., Olesen, M. C., Dalsgaard, A., Larsen, K. G., & Pol, J. V. D. (2013). Multi-Core Emptiness Checking of Timed Büchi Automata using Inclusion Abstraction. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)* (Vol. 8044, pp. 968-983). Springer Publishing Company. Lecture Notes in Computer Science Vol. 8044 https://doi.org/10.1007/978-3-642-39799-8_69

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,J.C.vandePol}@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, but some other structural properties are preserved. Based on those, we propose a variation of the classical nested depth-first search (NDFS) algorithm that exploits subsumption. In addition, we extend the multi-core CNDFS algorithm with subsumption, providing the first parallel LTL model checking algorithm for timed automata.
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 [27] solves the problem efficiently by translating it to the Büchi emptiness problem, which has been shown decidable for real-time systems as well [1]. However, its complexity is exponential, both in the size of the system specification and of 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.

[*] Danish authors partially supported by the MBAT ARTEMIS project, the MT-LAB VKR Centre of Excellence and the IDEA4CPS Sino-Danish Basic Research Centre.

Related work. The verification of timed automata was made possible by Alur and Dill's *region construction* [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 TA, coarser abstractions such as the symbolic *zone abstraction* have been studied [13], and also implemented in, among others, the state-of-the-art model checker UPPAAL [22]. Later, the k-extrapolation for zones was refined to include lower clock constraints in the so-called lower/upper-bound (LU) abstraction proposed in [4]. Finally, the *inclusion abstraction*, or simply *subsumption*, prunes reachability according to the partial order of the symbolic states [12]. All these abstractions preserve reachability properties [12,4].

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. [8] showed that the region-closed simulation graph preserve TBA emptiness. Tripakis [25] 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 [23] showed that the LU abstraction does in fact preserve TBA emptiness. The status of subsumption in LTL model checking is still open.

One way of establishing TBA emptiness on a finite simulation graph is the nested depth-first (NDFS) algorithm [9,16]. Recently, some multi-core version of these algorithms were introduced by Evangelista and Laarman et al [17,15,14]. 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 [18] and yield short counter examples [14, Sec. 4.3]. The latest version, called CNDFS, combines all these qualities and decreases memory usage [14].

In previous work, we parallelised reachability for timed automata using the mentioned abstractions [11]. 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 prove that subsumption preserves several structural state space properties (Sec. 3), and 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 [20] and opaal [10]. Finally, (4) our experiments show considerable state space reductions by subsumption and good parallel scalability of CNDFS with speedups of up to 40 using 48 cores.

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 [5].

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:*

$$g ::= c \bowtie n \mid g \wedge g \mid \mathit{true}$$

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 $\bowtie \in \{<, \leq, =\}$.

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 a 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 $v + \delta \models I_{\mathcal{C}}(\ell)$.*

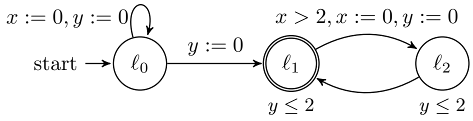


Fig. 1. A timed Büchi automaton.

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 $\mathit{Time}(\pi) = \sum_{i \geq 1} \delta_i$. An infinite path π in $\mathcal{TS}_v^{\mathbb{B}}$ is *time convergent*, or zeno, if $\mathit{Time}(\pi) < \infty$, 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. However, any TBA $\mathbb{B}$ can be syntactically transformed to a *strongly non-zeno* $\mathbb{B}$' [26], 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) [6].

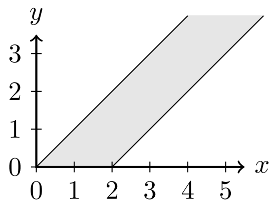


Fig. 2. A graphical representation of a zone over 2 clocks: $0 \leq x - y \leq 2$.

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 [6], 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: $[\![Z \wedge Z']\!] = [\![Z]\!] \cap [\![Z']\!]$.

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 $(\ell_0, Z_0\uparrow \wedge I_{\mathcal{C}}(\ell_0))$ 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 $Z \wedge g \neq \mathit{false}$, $Z' = (((Z \wedge g)[R])\uparrow) \wedge\ I_{\mathcal{C}}(\ell')$ and $Z' \neq \mathit{false}$.*

Any simulation graph is a discrete graph, hence cycles and lassos are defined in the standard way. We write $s \Rightarrow^+ s'$ iff there is a non-empty path in $SG(\mathbb{B})$ from s to s', or $s \Rightarrow^* s'$ if the path can be empty. 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: $s_0 \Rightarrow^* s \Rightarrow^+ s$.

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 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 [4]. Extrapolation can be refined on a per-clock basis [4], and on a per-location basis.

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

Definition 10. *Abstraction α over zone transition system $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ induces a zone transition system $SG_\alpha(\mathbb{B}) = (\mathcal{S}_\alpha, \alpha(s_0), \mathcal{T}_\alpha)$ where:*

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

We call an abstraction α an *extrapolation* if there exists a simulation relation R s.t. if $\alpha((\ell, Z)) = (\ell, Z')$ then for all $v' \in Z'$ there exist a $v \in Z$ s.t. $v'Rv$ [23]. This means extrapolations do not introduce behaviour that the un-extrapolated system cannot simulate. The abstraction defined by k-extrapolation is denoted by α_k, while the abstraction defined by LU-extrapolation is called α_{lu}. Hence, α_k and α_{lu} induce finite simulation graphs, written $SG_k(\mathbb{B})$ 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 [12], 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 [12]). *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.* $s \sqsubseteq \alpha_\sqsubseteq(s)$

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 [11].

Property preservation under abstractions. We now consider the preservation by the abstractions above of the property of *location reachability* (a location ℓ is reachable iff $s_0 \Rightarrow^* (\ell, \ldots)$) and that of Büchi emptiness.

Proposition 2. *For any of the abstractions* α*:* α_k *[12],* α_{lu} *[4],* $\alpha_k \circ \alpha_\sqsubseteq$ *[12], and* $\alpha_{lu} \circ \alpha_\sqsubseteq$ *[4], it holds that* ℓ *is reachable in* $\mathcal{TS}_v^\mathbb{B} \iff \ell$ *is reachable in* $SG_\alpha(\mathbb{B})$

Proposition 3. *For any finite extrapolation [23]* α*, e.g. the abstractions* α_k *[25] and* α_{lu} *[23] it holds that* $\mathcal{L}(\mathbb{B}) = \emptyset \iff \mathcal{L}(SG_\alpha(\mathbb{B})) = \emptyset$

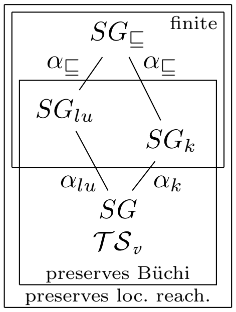


Fig. 3. Abstractions.

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* [27,2,12], we use a NEXT-STATE(s) function which returns the set of successor states for s: $\{s' \in \mathcal{S}_{fin} \mid s \Rightarrow s'\}$.

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 a lasso:* $s_0 \Rightarrow s \Rightarrow^+ s$. Tripakis [25] 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 $SG_{\sqsubseteq}(\mathbb{B}) = (SG_{\mathit{fin}\circ\sqsubseteq}(\mathbb{B}))$.

Proposition 4. *For all abstractions α, $s \in \mathcal{F} \Leftrightarrow \alpha(s) \in \mathcal{F}$ (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 [25] in $SG_{\mathit{fin}}(\mathbb{B})$, and because $\sqsubseteq$ is a simulation relation a similar path exists in $SG_{\sqsubseteq}(\mathbb{B})$. □

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 $SG_{\mathit{fin}}(\mathbb{B})$, as Fig. 4 illustrates. The figure visualises $SG_{\sqsubseteq}(\mathbb{B})$ by drawing subsumed states inside subsuming states (e.g. $s_3 \sqsubseteq s_1$).

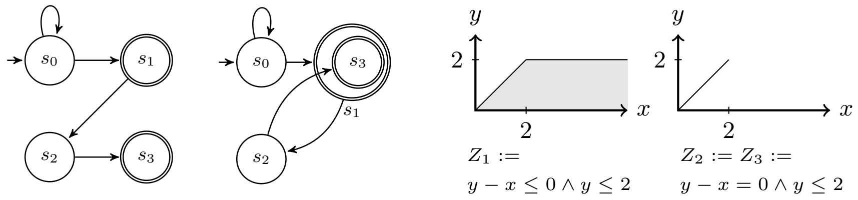


Fig. 4. The state space $SG_{\sqsubseteq}(\mathbb{B})$ of the model in Fig. 1 with $\ell_1 \in \mathcal{F}$ contains 4 states (shown on the left): s_0, $s_1 = (\ell_1, Z_1)$, $s_2 = (\ell_2, Z_2)$ and $s_3 = (\ell_1, Z_3)$. The graphical representation of the zones Z_1–Z_3 (right) reveals that $Z_3 \subseteq Z_1$ 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 $SG_{\sqsubseteq}(\mathbb{B})$.

However, subsumption introduces strong properties on paths and cycles to which we devote the rest of the current section. In subsequent sections, we exploit these properties to improve algorithms that implement the TBA emptiness check.

Lemma 1 (Accepting cycles under $\sqsubseteq$). *If $SG_{\mathit{fin}}(\mathbb{B})$ 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. $t \sqsubseteq x$:

$$\begin{tikzcd}
s' \arrow[r, Rightarrow, "*"] & t' \arrow[r, Rightarrow] & \cdots \arrow[r, Rightarrow] & x \\
s \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow, "*"] & t \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow] & \cdots \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow] & t \arrow[u, phantom, sloped, "\sqsubseteq"]
\end{tikzcd}$$

From x, we again have a similar path, to some x'. This sequence will eventually repeat some x'', because $SG_{\mathit{fin}}(\mathbb{B})$ 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 $SG_{\mathit{fin}}(\mathbb{B})$ 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 $SG_{\mathit{fin}}(\mathbb{B})$ is finite, the sequence of t's will eventually repeat some element x, s.t. $x \Rightarrow^+ \cdots \Rightarrow^+ x$.

$$\begin{tikzcd}
s' \arrow[r, Rightarrow, "+"] & t \arrow[r, Rightarrow, "+"] & t' \arrow[r, Rightarrow, "+"] & \cdots \arrow[r, Rightarrow, "+"] & t'' \arrow[r, Rightarrow, "+"] & x \\
s \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow, "+"] & s' \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow, "+"] & t \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow, "+"] & \cdots \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow, "+"] & x \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow, "+"] & x \arrow[u, phantom, sloped, "="]
\end{tikzcd}$$

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 [9,24] algorithm to exploit subsumption. The algorithm detects accepting cycles, the absence of which implies Büchi emptiness. It is correct for the graph $SG_{\mathit{fin}}(\mathbb{B})$ 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:

Alg. 1 NDFS

```
procedure ndfs( )
    $Cyan := Blue := Red := \emptyset$
    $dfsBlue(s_0)$
    report no cycle
procedure dfsRed(s)
    $Red := Red \cup \{s\}$
    for all $t$ in NEXT-STATE($s$) do
        if ($t \in Cyan$) then report cycle
        if ($t \notin Red$) then $dfsRed(t)$
procedure dfsBlue(s)
    $Cyan := Cyan \cup \{s\}$
    for all $t$ in NEXT-STATE($s$) do
        if $t \notin Blue \wedge t \notin Cyan$ then
            $dfsBlue(t)$
    if $s \in \mathcal{F}$ then
        $dfsRed(s)$
    $Blue := Blue \cup \{s\}$
    $Cyan := Cyan \setminus \{s\}$
```

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 [24], and
3. *Red*, visited by *dfsRed*.

Alg. 1 maintains a few strong invariants, which are already mentioned in [9,24]:

I0: At l.13 all red states are blue.
I1: The only accepting state visited by *dfsRed* is the seed.
I2: Outside of *dfsRed*, accepting cycles are not reachable from red states.
I3: A sufficient postcondition for $\mathit{dfsRed}(s)$ is that all reachable states from s are included in *Red* and no cyan state is reachable from it.

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, leading to the following options:

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 $t \not\sqsubseteq \mathit{Red}$.

Subsumption on cyan for cycle detection as in option 1a makes the algorithm unsound: cycles in $SG_{\sqsubseteq}(\mathbb{B})$ are not always reflected in $SG_{\mathit{fin}}(\mathbb{B})$ (Fig. 4). There is also no hope of "unwinding" 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 option 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.

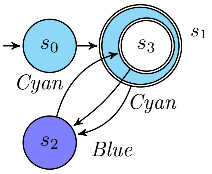
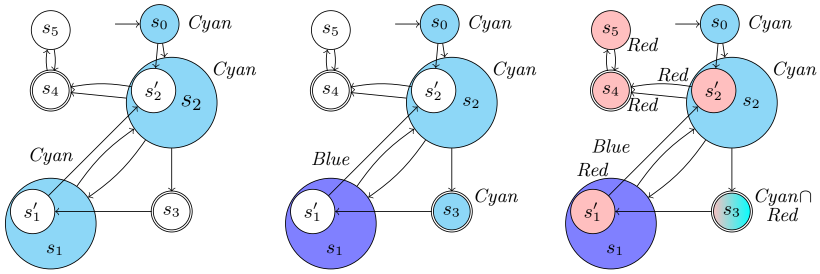


Fig. 5. Counter example to subsumption on *Blue* and *Cyan* (option 2).

It is tempting therefore to use subsumption on blue only, as in option 3. However, Fig. 6 shows an "animation" of a run with the modified NDFS which is incomplete. Here state s_1 is first backtracked in the blue search as all successors are cyan (left). Then state s_1 is marked blue; The blue search backtracks to s_2, proceeds to s_3 and backtracks because it finds $s_1' \sqsubseteq s_1 \in \mathit{Blue}$ (middle). Then a red search is started from s_3, which 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) $\mathit{dfsBlue}(s_1)$ (b) $\mathit{dfsBlue}(s_3)$ (c) dfsRed from s_3

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

A viable option however is to use inverse subsumption on cyan as in option 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 option 4. By definition (Def. 11), $SG_{\sqsubseteq}(\mathbb{B})$ contains a "larger" state for all reachable states in $SG_{\mathit{fin}}(\mathbb{B})$. So in combination with option 1b this is sufficient to find all accepting cycles.

The strong invariant (I2) states accepting cycles are not reachable from red states, so red states can prune the blue search. We can strengthen the condition on l.13 to $t \notin \mathit{Blue} \cup \mathit{Cyan} \cup \mathit{Red}$. However, this is of no use since by (I0), $\mathit{Red} \subseteq \mathit{Blue}$. Luckily, even states subsumed by red do not lead to accepting cycles (contraposition of Lemma 1), so we can use subsumption again: $t \notin \mathit{Blue} \cup \mathit{Cyan} \wedge t \not\sqsubseteq \mathit{Red}$. 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. 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:      $\mathit{Cyan} := \mathit{Blue} := \mathit{Red} := \emptyset$
3:      $\mathit{dfsBlue}(s_0)$
4:      report no cycle
5:  procedure $\mathit{dfsRed}(s)$
6:      $\mathit{Red} := \mathit{Red} \cup \{s\}$
7:      for all $t$ in NEXT-STATE($s$) do
8:          if ($\underline{\mathit{Cyan} \sqsubseteq t}$) then report cycle
9:          if ($\underline{t \not\sqsubseteq \mathit{Red}}$) then $\mathit{dfsRed}(t)$
10: procedure $\mathit{dfsBlue}(s)$
11:     $\mathit{Cyan} := \mathit{Cyan} \cup \{s\}$
12:     for all $t$ in NEXT-STATE($s$) do
13:         if ($t \notin \mathit{Blue} \cup \mathit{Cyan} \underline{\wedge t \not\sqsubseteq \mathit{Red}}$)
14:         then $\mathit{dfsBlue}(t)$
15:     if $s \in \mathcal{F}$ then
16:         $\mathit{dfsRed}(s)$
17:     $\mathit{Blue} := \mathit{Blue} \cup \{s\}$
18:     $\mathit{Cyan} := \mathit{Cyan} \setminus \{s\}$
```

5 Multi-Core CNDFS with Subsumption

CNDFS [14] is a parallel algorithm for checking Büchi emptiness [14]. 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 without underlined parts), 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). However, the workers assist each other by sharing the colours *Blue* and *Red* globally, thus pruning each other's search space.

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 option 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 [14, 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 [14, 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 [14]).
I3' After $dfsRed_i(s)$ states reachable from s are red or in $\mathcal{R}_i$ (from [14, Lem. 2]).

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 [14]. We additionally have to extend the waiting procedure

Alg. 3 CNDFS with subsumption

```
1: procedure $cndfs(P)$
2:     $Blue := Red := \emptyset$
3:     forall $i$ in $1..P$ do $Cyan_i := \emptyset$
4:     $dfsBlue_1(s_0) \parallel .. \parallel dfsBlue_P(s_0)$
5:     report no cycle
6: procedure $dfsRed_i(s)$
7:     $\mathcal{R}_i := \mathcal{R}_i \cup \{s\}$
8:     for all $t$ in NEXT-STATE$_i(s)$ do
9:         if $\underline{Cyan \sqsubseteq t}$ then cycle
10:        if $t \notin \mathcal{R}_i \wedge \underline{t \not\sqsubseteq Red}$ then
11:            $dfsRed_i(t)$
12: procedure $dfsBlue_i(s)$
13:     $Cyan_i := Cyan_i \cup \{s\}$
14:     for all $t$ in NEXT-STATE$_i(s)$ do
15:         if $t \notin Cyan_i \cup Blue \underline{\wedge t \not\sqsubseteq Red}$ then
16:             $dfsBlue(t)$
17:     $Blue := Blue \cup \{s\}$
18:     if $s \in \mathcal{F}$ then
19:         $\mathcal{R}_i := \emptyset$
20:         $dfsRed(s)$
21:         await $\forall s' \in \mathcal{R}_i \cap \mathcal{F} \setminus \{s\}$: $\underline{s' \sqsubseteq Red}$
22:         forall $s'$ in $\mathcal{R}_i$ do $Red := Red \cup s'$
23:     $Cyan_i := Cyan_i \setminus \{s\}$
```

to include subsumption at l.21, because the use of subsumption in $dfsRed_i$ can cause other workers to find "larger" states.

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 randomised successor function NEXT-STATE$_i$ however ensures that this does not happen for most practical inputs. Experiments on over 300 examples confirmed this [14, Sec. 4], making CNDFS the current state-of-the-art parallel LTL model checking algorithm.

6 Experimental Evaluation

To evaluate the performance of the proposed algorithms experimentally, we implemented CNDFS without [14] and with subsumption (Alg. 3) in LTSmin 2.0[3]. The opaal [10] tool[4] functions as a front-end for UPPAAL models. Previously, we demonstrated scalable multi-core reachability for timed automata [11].

Experimental setup. We benchmarked[5] on a 48-core machine (a four-way AMD Opteron$^{\text{TM}}$ 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 on collisions, eventually the bus will be active again: $\Box(\texttt{(P0=bus\_collision1)} \implies \Diamond\texttt{(P0=bus\_active)})$.

`fischer-1/2`[7] implements a mutual exclusion protocol; a canonical benchmark for timed automata, with 10 nodes. As in [23], we use the property (1): $\neg((\Box\Diamond\texttt{k=1})\vee(\Box\Diamond\texttt{k=0}))$, where `k` is the number of processes in their critical section. We also add a weak fairness property (2): $\Box((\Box\texttt{P\_1=req}) \implies (\Diamond\texttt{P\_1=cs}))$: processes requesting infinitely often will eventually be served.

`fddi`[6] models a token ring system as described in [8], 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 [8] that every station will eventually send asynchronous messages: $\Box(\Diamond\texttt{(ST1=station\_z\_sync)})$.

`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: $\Box(\texttt{Train\_1=Appr} \implies (\Diamond\texttt{Train\_1=Cross}))$.

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]`.

[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.

This runs algorithm A on the cross-product of the model m with the Büchi automaton of formula f. It uses a fixed hash table of size 2^{28} and P threads, and either subsumption (-u1) or not (-u0). The option ltl-semantics selects textbook LTL semantics as defined in [2, Ch. 4]. To investigate the overhead of CNDFS, we also run the multi-core algorithms for plain reachability 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 both 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 [14, Sec. 4.2].

Implementation. LTSMIN defines a NEXT-STATE function as part of its PINS interface for language-independent symbolic/parallel model checking [7]. Previously, we extended PINS with subsumption [11]. 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 [20] stores states in one lockless hash/tree table in shared memory [19,21]. For timed systems, this table is used to store *explicit state parts*, i.e. the locations and state variables [5]. 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 [11]. 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 untimed 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 many calculations. In modern multi-core computers, this feature improves scalability, because it more closely matches the machine's high frequency/bandwidth ratio [19]. On the other hand, the lock granularity increases since a single lock now governs multiple DBMs stored in the multimap [11, Sec. 6.1]. Nonetheless, for multi-core timed reachability, previous experiments showed almost linear scalability [11, Sec. 7], even when using other model checkers (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.

Subsumption probably improves the absolute performance of CNDFS. We expect that models with many clocks and constraints exhibit a better reduction than others. Moreover, it is known [3] that the reduction due to subsumption depends strongly on the exploration order: BFS typically results in better reductions than DFS, since "large" states are encountered later. CNDFS might share

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

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

this disadvantage with DFS. However, as shown in [11], subsumption with random parallel DFS performs much better than sequential DFS, which could be beneficial for the scalability of CNDFS. So it is really hard to predict the relative performance and scalability of these algorithms, and the effects of subsumption.

Experimental results without subsumption. We first compare the algorithms BFS, DFS (parallel reachability) and CNDFS (accepting cycles) without subsumption. Table 1 shows their sequential ($P = 1$) and parallel ($P = 48$) runtimes (T). Note that sequential CNDFS is just NDFS. We show the number of explicit state parts ($|L|$), full states ($|\mathcal{R}|$), transitions ($|\Rightarrow|$), and also the number of states visited in CNDFS ($|V|$). These numbers confirm the findings reported previously for CNDFS applied to untimed systems: The sequential run times ($P = 1$) are very similar, indicating little overhead in CNDFS. For the parallel runs ($P = 48$), however, the number of states visited by CNDFS ($|V|$) increases due to work duplication.

To further investigate the scalability of the timed CNDFS algorithm, we plot the speedups in Fig. 7. Vertical bars represent the (mostly negligible) standard deviation over the five benchmarks. 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 has only 119 explicit state parts, we attribute the poor scalability to lock contention, harming more with a growing number of workers.

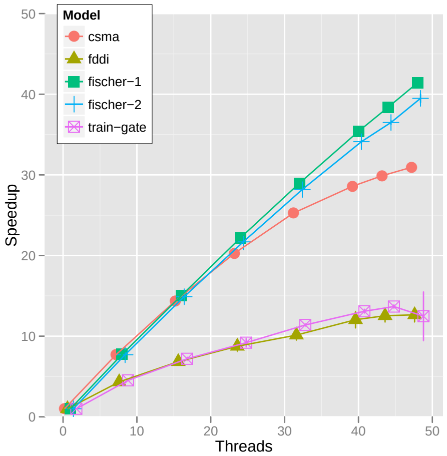


Fig. 7. Speedups in LTSMIN/opaal

Subsumption. Table 2 shows the experimental data for BFS, DFS and CNDFS with subsumption (Alg. 3). The number of explicit state parts $|L|$ is stable, since

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

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

reachability of locations is preserved under subsumption (Prop. 2). However, the achieved reduction of full states depends on the search order, so we now report $|\mathcal{R}|$ per algorithm, as a percentage of the original numbers.

We confirm [3] that subsumption works best for BFS reachability, with even more than 30-fold reduction for fddi, but none for fischer (cf. column $|\mathcal{R}|_{bfs}$). For these benchmarks, the reduction is correlated to the ratio $X = |\mathcal{R}|/|L|$; e.g. $X \approx 1500$ for fddi and $X \approx 10$ for fischer. Subsumption is much less effective with sequential DFS, but parallel DFS improves it slightly (cf. column $|\mathcal{R}|_{dfs}$).

CNDFS benefits considerably from subsumption, but less so than BFS: we observe around 2-fold reduction for fddi, fischer and csma (cf. column $|\mathcal{R}|_{cndfs}$). Surprisingly, the reduction for parallel runs of CNDFS is not better than for sequential runs. One disadvantage of CNDFS compared to BFS is that only red states attribute to subsumption reduction. Probably some "large" states are never coloured red. We measured that for all benchmark models, 20%–50% of all reachable states are coloured red (except for fischer-2, which has no red states).

Subsumption decreases the running times for reachability: a lot for BFS, and still considerably for DFS, both in the sequential case and the parallel case, up to 48 workers. However, subsumption is less beneficial for the running time of CNDFS (it might even increase), but the speedup remains unaffected.

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, K.G. Larsen, and R. Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In *TACAS*, LNCS 2988, pages 312–326. Springer, 2004.
5. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In *FMDRTS*, LNCS 3185, pages 200–236. Springer, 2004.
6. J. Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala University, 2002.

7. S. C. C. Blom, J. C. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In *CAV*, LNCS 6174, pages 354–359. Springer, 2010.
8. 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.
9. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *CAV*, LNCS 531, pages 233–242. Springer, 1990.
10. 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*, LNCS 6617, pages 487–493. Springer, 2011.
11. 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 7595, 2012.
12. C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS*, LNCS 1384, pages 313–329. Springer, 1997.
13. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *AVMFSS*, LNCS 407, pages 197–212. Springer, 1989.
14. S. Evangelista, A.W. Laarman, L. Petrucci, and J.C. van de Pol. Improved multi-core nested depth-first search. In *ATVA*, LNCS 7561, pages 269–283, 2012.
15. 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.
16. G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *The Spin Verification System, 2nd SPIN workshop*, pages 23–32. AMS, 1996.
17. 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.
18. A.W. Laarman and J.C. van de Pol. Variations on multi-core nested depth-first search. In *PDMC*, EPTCS 72, pages 13–28, 2011.
19. 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.
20. 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.
21. A.W. Laarman, J.C. van de Pol, and M. Weber. Parallel recursive state compression for free. In *SPIN*, LNCS 6823, pages 38–56. Springer, 2011.
22. K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *STTT*, 1:134–152, ’97.
23. G. Li. Checking timed Büchi automata emptiness using LU-abstractions. In *FORMATS*, LNCS 5813, pages 228–242. Springer, 2009.
24. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *TACAS*, LNCS 3440, pages 174–190. Springer, 2005.
25. S. Tripakis. Checking timed Büchi automata emptiness on simulation graphs. *TOCL*, 10(3):15, 2009.
26. S. Tripakis, S. Yovine, and A. Bouajjani. Checking timed Büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.
27. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344. IEEE, 1986.