Timed Automata Verification and Synthesis via Finite Automata Learning

Ocan Sankur

Timed Automata Verification and Synthesis via Finite Automata Learning*

Ocan Sankur

Univ Rennes, Inria, CNRS, Rennes, France

Abstract. We present algorithms for model checking and controller synthesis of timed automata, seeing a timed automaton model as a parallel composition of a large finite-state machine and a relatively smaller timed automaton, and using compositional reasoning on this composition. We use automata learning algorithms to learn finite automata approximations of the timed automaton component, in order to reduce the problem at hand to finite-state model checking or to finite-state controller synthesis. We present an experimental evaluation of our approach.

1 Introduction

Timed automata [1] are a well-known formalism for modeling and verifying real-time systems. They can be used to model systems as finite automata, while using, in addition, clocks to impose timing constraints on the transitions. Using clock variables have advantages. They allow one to describe models that are expressive thanks to real-valued clock values; moreover, the use of specific clock variables enable optimizations such as sound and complete abstractions, also known as extrapolation operators [5]. Model checking algorithms have been developed and implemented in tools such as Uppaal [8], TChecker [29], PAT [50].

One approach for model checking timed automata is based on representing the set of clock values with *zones*, which are particular polyhedra, and using explicit enumeration on the discrete states. There has been extensive research on sound and complete abstractions on zones, which improved the performance of the model checking tools, and made it possible to handle models with more complex time constraints; see [11] for a survey. However this approach does not scale to models with large discrete spaces due to explicit enumeration. Several authors have developed algorithms to remedy this issue, and to attempt to adapt efficient model checking techniques finite-state systems to timed systems. Extensions of binary decision diagrams (BDD) with clock constraints have been considered both for continuous time [53,10,24] and discrete time [43,51]. Another approach is to use predicate abstraction on clock variables that enables efficient finite-state verification techniques based on BDDs or SAT solvers [17,16,47].

Controller synthesis is a related problem in which some transitions of the system are controllable and some are uncontrollable, and the objective is to

* This work was partially funded by ANR project Ticktac (ANR-18-CE40-0015).

compute a control strategy which guarantees that all induced runs of the system satisfy a given specification; see *e.g.* [52]. This problem is formalized using *games*, and in the case of real-time systems, using timed games [40,4]. Zone-based algorithms have been developed to solve timed games and compute control strategies [14], and are available in the Uppaal TIGA tool [7]. These algorithms suffer from the same limitations as the zone-based model checking algorithms. Although they can be efficient on instances with small discrete state spaces, they do not scale well to large systems. An attempt was made to implement the counter-example guided abstraction refinement scheme to handle larger discrete state space in timed games in [45]. On the other hand, there are several efficient finite-state game solvers, based on BDDs and SAT solvers, which can efficiently handle relatively large state spaces [32], but cannot handle real time.

In this work, we introduce an approach that is applied both to model checking and controller synthesis of timed automata with the objective of combining the advantages of both timed automata and finite-state model checkers and game solvers. Our suggestion is to see the input model, without loss of generality, as a parallel composition between a finite-state machine $\mathcal{A}$, and a timed automaton $\mathcal{T}$. We specifically target instances where $\mathcal{A}$ is large, and $\mathcal{T}$ is relatively small but nontrivial. Note that this point of view was considered before in the verification of synchronous systems within a real-time environment [9]. As a novelty, for model checking, we apply a compositional reasoning rule on the product $\mathcal{A}\|\mathcal{T}$ by replacing the timed automaton $\mathcal{T}$ by a (small) deterministic finite automaton (DFA) H which represents the behaviors of $\mathcal{T}$. To automatically select the DFA H, we adapt the algorithm [44] to our setting, and use a DFA learning algorithm (such as L* [3], or TTT [30]) to find an appropriate DFA either to prove the specification or to reveal a counterexanple.

Our approach enjoys the principle of *separation of concerns* in the following sense. A timed automaton model checker is used by the learning algorithm to answer *membership* and *equivalence queries* (see Section 2.2); these are answered without refering to $\mathcal{A}$, thus, by avoiding the large discrete state space. Therefore, the timed automaton model checker is used in this approach for what it is designed for: handling real-time constraints encoded in $\mathcal{T}$, not for dealing with excessive discrete state spaces. Once an appropriate DFA H is found by the learning algorithm, the system $\mathcal{A}\|H$ is model-checked using a finite-state model checker whose focus is to deal with large *discrete* state spaces. We can thus benefit from the best of the two worlds: a state-of-the-art model checker for timed automata, which is somewhat used here as a theory solver, and any finite-state model checker based on BDDs, SAT solvers, or even explicit-state enumeration.

The application of the learning-based compositional reasoning of [44] to controller synthesis is more involved. Our objective was to find a way to exploit efficient finite-state game solvers [32] in the context of timed automata even if this meant having an incomplete algorithm. We describe a setting where a one-sided abstraction is applied for controller synthesis by replacing the timed automaton component by a learned DFA. Contrarily to the model checking algorithm, our controller synthesis algorithm is sound but not complete, that is,

the algorithm may fail although there exists a control strategy, while any control strategy that is output is correct. More precisely, we consider timed games in the form $\mathcal{G}\|\mathcal{T}$ where $\mathcal{G}$ is a finite-state game, and $\mathcal{T}$ is a timed automaton. We describe an algorithm that alternates between two phases. In the first phase, the goal is to find a DFA $\overline{H}$ that is an overapproximation of $\mathcal{T}$. Once this is found, we use a finite-state game solver on $\mathcal{G}\|\overline{H}$; if there is a control strategy, we prove that it can be applied in the original system $\mathcal{G}\|\mathcal{T}$. If not, then we obtain a counterstrategy $\mathfrak{S}$. We then switch to the second phase whose goal is to check whether the counterstrategy is spurious or not; and it does so by learning an underapproximation DFA $\underline{H}$ of $\mathcal{T}$, and checking whether $\mathfrak{S}$ induces runs that are all in $\underline{H}$. Accordingly, we either reject the instance or switch back to the first phase. As in the model checking algorithm, the timed automaton model checker is only used to answer queries independently from $\mathcal{G}$, and a finite-state game solver and a model checker are used to compute and analyze strategies in a discrete state-space.

To the best of our knowledge, apart from [45], we present the first algorithm that can solve timed games with large discrete state spaces. Although the algorithm applies to a subset of timed games and is not complete, we believe it is of utmost importance to make progress on the scalability of timed game solvers in order for these methods to be applied in convincing applications. Our paper makes an attempt in this direction.

We evaluate our algorithms in comparison with state-of-the-art tools and show that our approach is competitive with the existing tools, and can allow both model checking and synthesis to scale to larger models. The approach offers an alternative treatment of timed models, which might be applied in other settings.

We present the model checking algorithm in Section 2 which contains formal definitions, the description of the algorithm, and the experiments. Section 3 presents our contributions on the controller synthesis problem, and includes formal definitions, the description of the algorithm, and the experiments. In Section 4, we provide a broader discussion on related works, and present our conclusions and perspectives.

2 Compositional Model Checking

2.1 Preliminaries

Labeled Transition Systems and Finite Automata. We denote finite *labeled transition systems (LTS)* as tuples (Q, q^0, Σ, T) where Q is the set of states, $q^0 \in Q$ is the initial state, Σ is a finite alphabet, $T \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$ is the transition relation (ϵ labels silent transitions). Because we will consider synchronous product of LTSs, we will use silent transitions to define internal transitions not exposed for synchronization. A *finite automaton* is an LTS given with a set of accepting states $F \subseteq Q$, and is written (Q, q^0, Σ, T, F). A *run* of an automaton is a sequence $q_1 e_1 q_2 e_2 \dots q_n$ where $q_1 = q^0$, $e_i = (q_i, \sigma_i, q_{i+1}) \in T$ for some $\sigma_i \in \Sigma \cup \{\epsilon\}$ for each $1 \leq i \leq n-1$. The *trace* of the run is the sequence $\sigma_1 \sigma_2 \dots \sigma_{n-1}$. An

accepting run starts at q^0 and ends in F. The language of a finite automaton $\mathcal{A}$ is the set of the traces of all accepting runs of $\mathcal{A}$, and is denoted by $\mathcal{L}(\mathcal{A})$. We will consider *deterministic finite automata (DFA)* which do not have silent transitions, and have at most one edge for each label from each state.

The *parallel composition* of two automata $\mathcal{A}_i = (Q_i, q_i^0, \Sigma, T_i, F_i)$, $i \in \{1, 2\}$, defined on the same alphabet, is the automaton $\mathcal{A}_1 \parallel \mathcal{A}_2 = (Q, q^0, \Sigma, T, F)$ with $Q = Q_1 \times Q_2$, $q^0 = (q_1^0, q_2^0)$, $F = F_1 \times F_2$, and T contains $((q_1, q_2), \sigma, (q_1', q_2'))$ for all $(q_1, \sigma, q_1') \in T_1$, and $(q_2, \sigma, q_2') \in T_2$; and $((q_1, q_2), \epsilon, (q_1', q_2))$ for all $(q_1, \epsilon, q_1') \in T_1$, and $q_2 \in Q_2$; and symmetrically, $((q_1, q_2), \epsilon, (q_1, q_2'))$ for all $(q_2, \epsilon, q_2') \in T_2$, and $q_1 \in Q_1$.

Finite Automata Learning. We use finite automata learning algorithms such as L^* [3,46] and TTT [30]. In the *online* learning model, the learning algorithm interacts with a teacher in order to learn a deterministic finite automaton recognizing a hidden regular language known to the teacher. The algorithm can make two types of queries. A *membership query* consists in asking whether a given word belongs to the language, to which the teacher answers by yes or no. An *equivalence query* consists in creating a hypothesis automaton H, and asking the teacher whether H recognizes the language. The teacher either answers yes, or no and provides a counterexample word which is in the symmetric difference of $\mathcal{L}(H)$ and of the target language. Learning algorithms typically make a large number of membership queries, and a smaller number of equivalence queries.

Timed Automata. We fix a finite set of clocks $\mathcal{C}$. *Clock valuations* are the elements of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$. For $R \subseteq \mathcal{C}$ and a valuation v, $v[R \leftarrow 0]$ is the valuation defined by $v[R \leftarrow 0](x) = v(x)$ for $x \in \mathcal{C} \setminus R$ and $v[R \leftarrow 0](x) = 0$ for $x \in R$. Given $d \in \mathbb{R}_{\geq 0}$ and a valuation v, $v + d$ is defined by $(v + d)(x) = v(x) + d$ for all $x \in \mathcal{C}$. We extend these operations to sets of valuations in the standard way. We write $\vec{0}$ for the valuation that assigns 0 to every clock.

We consider a clock named 0 which has the constant value 0, and let $\mathcal{C}_0 = \mathcal{C} \cup \{0\}$. An *atomic guard* is a formula of the form $x \bowtie k$, or $x - y \bowtie k$ where $x, y \in \mathcal{C}_0$, $k, l \in \mathbb{N}$, and $\bowtie \in \{<, \leq, >, \geq\}$. A *guard* is a conjunction of atomic guards. A valuation v satisfies a guard g, denoted $v \models g$, if all atomic guards are satisfied when each $x \in \mathcal{C}$ is replaced by $v(x)$. Let $\Phi_{\mathcal{C}}$ denote the set of guards for $\mathcal{C}$.

A *timed automaton* $\mathcal{T}$ is a tuple $(L, \ell_0, \Sigma, \mathrm{Inv}, \mathcal{C}, E, F)$, where L is a finite set of locations, $\ell_0 \in L$ is the initial location, Σ is the alphabet, $\mathrm{Inv} : L \to \Phi_{\mathcal{C}}$ the invariants, $\mathcal{C}$ is a finite set of clocks, $E \subseteq L \times \Sigma \times \Phi_{\mathcal{C}} \times 2^{\mathcal{C}} \times L$ is a set of edges. An edge $e = (\ell, g, \sigma, R, \ell')$ is also written as $\ell \xrightarrow{g,\sigma,R} \ell'$. $F \subseteq L$ is the set of accepting locations.

A *run* of $\mathcal{T}$ is a sequence $r = q_1 e_1 q_2 e_2 \dots q_n$ where $q_i \in L \times \mathbb{R}_{\geq 0}^{\mathcal{C}}$, $q_1 = (\ell_0, \vec{0})$, and writing $q_i = (\ell, v)$ for each $1 \leq i \leq n$, we have $v \in \mathrm{Inv}(\ell)$. If $i < n$, then either $e_i \in \mathbb{R}_{>0}$ and $v + e_i \in \mathrm{Inv}(\ell)$, in which case $q_{i+1} = (\ell, v + e_i)$, or $e_i = (\ell, g, \sigma, R, \ell') \in E$, in which case $v \models g$ and $q_{i+1} = (\ell', v[R \leftarrow 0])$. The run is accepting if the last location is in F. The *trace* of the run r is the word $\sigma_0 \sigma_1 \dots \sigma_n$ where σ_i is the label of edge e_i.

The *untimed language* of the timed automaton $\mathcal{T}$ is the set the traces of the accepting runs of $\mathcal{T}$, and is denoted by $\mathcal{L}(\mathcal{T})$.

A timed automaton is *label-deterministic* if at each location ℓ, for each label $\sigma \in \Sigma$, there is at most one transition leaving ℓ labelled by σ; in other terms, the finite automaton obtained by removing all clocks is deterministic.

We consider the parallel composition of a finite automaton $\mathcal{A} = (Q, q^0, \Sigma, T, F)$ and a timed automaton $\mathcal{T} = (L, \ell_0, \Sigma, \mathrm{Inv}, \mathcal{C}, E, F_{\mathcal{T}})$ which is a new timed automaton. Intuitively, a transition labeled by σ consists in an arbitrary number of silent transitions of $\mathcal{A}$, followed by a joint σ-transition of both components. The guard and the reset of the overall transition are those of the transition of $\mathcal{T}$. Formally, let $\mathcal{A} \| \mathcal{T} = (L', \ell'_0, \Sigma, \mathrm{Inv}', \mathcal{C}, E', F')$ with $L' = Q \times L$, $\mathrm{Inv}' : (q, \ell) \mapsto \mathrm{Inv}(\ell)$, $\ell'_0 = (q_0, \ell_0)$, and E' contains all edges of the form $((q, \ell), g, \sigma, R, (q', \ell'))$ such that $(\ell, g, \sigma, R, \ell') \in E$, and there exists a sequence $q = q_0, q_1, \ldots, q_k, q_{k+1} = q'$ of states of $\mathcal{A}$ such that $(q_0, \epsilon, q_1), \ldots, (q_{k-1}, \epsilon, q_k), (q_k, \sigma, q_{k+1})$ are transitions of $\mathcal{A}$. We let $F' = F \times F_{\mathcal{T}}$.

It follows from the definition of the parallel composition that $\mathcal{L}(\mathcal{A} \| \mathcal{T}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{T})$.

Target Timed Automata Instances. Our main motivation is to consider real-time systems that are modeled naturally as $\mathcal{A} \| \mathcal{T}$. Typically, $\mathcal{A}$ has a large (discrete) state space, and $\mathcal{T}$ is a relatively small timed automaton, but with potentially complex time constraints involving several clocks.

It should be clear however that any timed automaton $\mathcal{T}$ can be seen as such a product as follows. Let $\mathcal{A}$ be a finite automaton identical to $\mathcal{T}$ except that guards and resets are removed; and for each pair of guard g and reset r, a fresh label $\sigma_{g,r}$ is defined and added to each edge with the said guard and reset. Now, define the timed automaton $\mathcal{T}'$ as a single state with the same clocks as $\mathcal{T}$, with one self-loop for each pair (g, r): such an edge is labeled by $\sigma_{g,r}$, has guard g, and reset r. We have that $\mathcal{T}$ is isomorphic to $\mathcal{A} \| \mathcal{T}'$.

An example is given in Figure 1 which shows how a simple scheduling setting can be modeled in this way. Here, the finite automaton is simple and only stores the mapping from machines to the tasks they are executing. Typically, if the machines or the processes executing tasks have internal states, these could be modeled in $\mathcal{A}$ as well without altering the timed automaton.

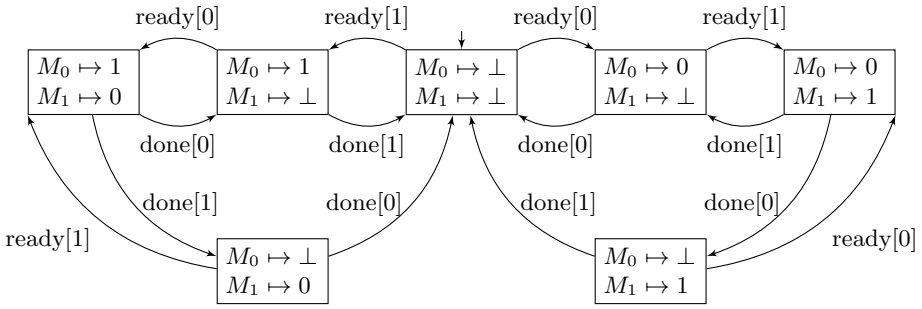
2.2 Learning-Based Compositional Model Checking Algorithm

We present an algorithm for model checking the untimed language $\mathcal{L}(\mathcal{A} \| \mathcal{T})$.

Although it is known that the untimed language is regular [1], the size of the corresponding finite automaton can be exponential so a direct computation is not efficient. We will be looking for a finite automaton H which is an *overapproximation* of $\mathcal{T}$ *i.e.* $\mathcal{L}(\mathcal{T}) \subseteq \mathcal{L}(H)$. H stands for *hypothesis* made by the learning algorithm. We will in fact use the following lemma.

Lemma 1. *For all finite automata $\mathcal{A}$ and H, and timed automata $\mathcal{T}$ on common alphabet Σ, if $\mathcal{L}(\mathcal{T}) \subseteq \mathcal{L}(H)$, then $\mathcal{L}(\mathcal{A} \| \mathcal{T}) \subseteq \mathcal{L}(\mathcal{A} \parallel H)$.*

Finite automaton $\mathcal{A}$:



Timed automaton $\mathcal{T}$:

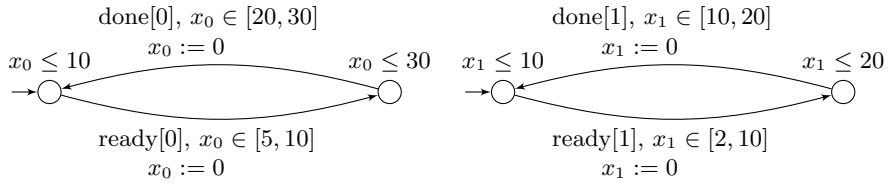


Fig. 1. Timed automaton $\mathcal{A}\|\mathcal{T}$ modeling a simple scheduling policy. The finite automaton $\mathcal{A}$ is given above and models a scheduler which schedules tasks (0 and 1) immediately when they become ready (ready[0] and ready[1]) on machines M_0 and M_1, using M_0 first if it is available. The timed automaton $\mathcal{T}$ is below, here, as a network of the timed automata, and models interarrival and computation times for each task.

In other terms, by replacing the timed automaton $\mathcal{T}$ by its overapproximation, we obtain an overapproximation of the compound system in terms of untimed language. So if a linear property can be established on $\mathcal{A}\|H$ for an appropriate H, then the property also holds on the original system.

Let us present the above property as a verification rule. Assuming that we want to establish $\mathcal{A}\|\mathcal{T} \subseteq \mathsf{Spec}$ for some language Spec, we have

$$\frac{\mathcal{L}(\mathcal{T}) \subseteq \mathcal{L}(H) \quad \mathcal{L}(\mathcal{A}\|H) \subseteq \mathsf{Spec}}{\mathcal{L}(\mathcal{A}\|\mathcal{T}) \subseteq \mathsf{Spec}.}\ \mathsf{Asym} \tag{1}$$

Here, H serves as an *assumption* we make on $\mathcal{T}$ when verifying $\mathcal{A}$; so as in Lemma 1, we can use H instead of $\mathcal{T}$ during model checking. The rule (1) is well known as the assume-guarantee verification rule [19], and has been used in model checking finite-state systems as well as timed automata [36]. The assumption H can either be provided by the user, or automatically computed using automata learning as in [44]. Intuitively, the model checking algorithm we present in this section is an application of [44] to our specific case.

Figure 2 presents the overview of the algorithm. The membership queries of the learning algorithm are answered by the membership oracle; the equivalence query with conjecture H is answered by the inclusion oracle. When the conjecture H

passes the inclusion check, we model-check $H \| \mathcal{A}$. When this is successful, we stop and declare that the original system $\mathcal{A} \| \mathcal{T}$ satisfies the specification. Otherwise, a counterexample $w \in \mathcal{L}(\mathcal{A} \| H) \setminus \mathsf{Spec}$ was found, and we use a realizability check to see whether $w \in \mathcal{L}(\mathcal{T})$ (this is actually done by the membership oracle). If the answer is yes, then the counterexample is confirmed, and we stop. Otherwise, we inform the learning algorithm that w must be excluded, and continue the learning process.

Note that this algorithm can be used for any regular language specificaton Spec. We focus on safety properties in our experiments, presented next.

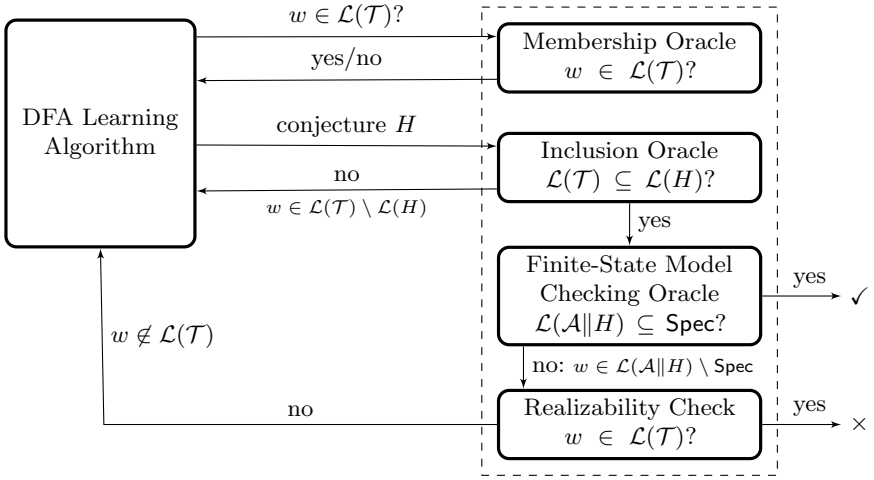


Fig. 2. The learning-based compositional model checking algorithm. The box on the left is a DFA learning algorithm, while the oracles answering the queries of the learning algorithm are shown on the right and correspond to the teacher.

2.3 Experiments

We built a prototype implementation of our algorithm in Scala, using the TTT automata learning algorithm [30] from the learnlib library [31], and the associated automatalib for manipulating finite automata. We used the TChecker [29] model checker for implementing membership and inclusion oracles. For the latter, we complement H into H^c, and check the emptiness of the parallel composition of $\mathcal{T}$ with H^c. We use the NuSMV model checker for finite-state model checking. The implementation is available at https://github.com/osankur/compRTMC.

The overall input consists in an SMV file describing $\mathcal{A}$, and of a TChecker timed automaton describing $\mathcal{T}$. We use define expressions in SMV to define the labels Σ, while TChecker allows us to tag each transition with a label.

We compare our algorithm on a set of benchmarks with the model checkers Uppaal [8] and nuXmv which has a timed automata model checker [16]. The

Table 1. Model checking benchmarks. The column #Clk is the number of clocks; #C is the number of conjectures made by the DFA learning algorithm; #M is the number of membership queries; and |DFA| is the size of the final finite automaton learned. The safety specification holds on all models but those marked with *. In each cell, — means out of memory (8GB), and - means time out (30 minutes).

		Compositional				Uppaal	nuXmv
	#Clk	#C	#M	|DFA|	Time	Time	Time
Leader Election A	3	13	232	15	**157s**	—	—
Leader Election B	3	26	661	29	**198s**	—	—
Leader Election C	3	33	997	53	**149s**	—	-
Leader Election D	3				-	—	-
Leader Election (Stateless) A	3	13	232	15	15s	**6s**	—
Leader Election (Stateless) B	3	28	776	33	44s	**8s**	—
Leader Election (Stateless) C	3	33	997	53	17s	**6s**	-
Leader Election (Stateless) D *	3	134	6965	240	10m7s	**6s**	-
FTSP-abstract-2	2	3	54	8	**2s**	**2s**	-
FTSP-abstract-3	3	17	340	23	**47s**	7m8s	-
FTSP-abstract-4	4				-	-	-
STS-2	5				**7s**	19s	-
STS-3	6				-	-	-
Rt-broadcast A	4	49	1324	63	**59s**	-	87s
Rt-broadcast B	4	41	1100	63	101s	-	**90s**
Rt-broadcast C	4	21	590	39	**31s**	-	86s
Rt-broadcast D	4	27	901	52	**49s**	-	80s
Priority Scheduling 2 A	3	35	9859	49	34s	**1s**	7s
Priority Scheduling 2 B	3	29	1162	42	16s	—	**2s**
Priority Scheduling 3 C	4				-	—	**6s**
Priority Scheduling 3 D	4				-	—	**8s**
Priority Scheduling 3 E *	4				-	—	**11s**

former implements a zone-based enumerative algorithm, while the latter uses predicate abstraction through IC3IA. We describe some of the benchmarks here.

The leader election protocol is a distributed protocol that can recover from crashes [22], extended here with periodic activation times and crash durations. The first four rows of Table 1 correspond to the case where one of the processes crashes when its internal state enters an error state. Internal states are modeled using Boolean circuits from from the synthesis competition (SYNTCOMP) benchmarks. The stateless version is more abstract: there is no internal state model, and crashes can occur at any time. The letters A, B, C, D indicate different timed automaton models. Uppaal was more efficient at solving the stateless version but failed in the full version due to the large discrete state space. The compositional algorithm was effective in verifying all instances but the D case which required a large finite automaton to be learned. One can notice an overhead of the compositional algorithm in the stateless version due to the computation of the finite automaton H. This was particularly an issue in the stateless D case where Uppaal could find a counterexample trace faster; nuXmv was not able to solve these instances.

The flooding time synchronization protocol (FTSP) is a leader election algorithm for multi-hop wireless sensor networks used for clock synchronization [41],

and has been the subject of formal verification before [42,35]. We consider the abstract model used in [48] for parameterized verification allowing one to verify the model for a large number of topologies. Our algorithm was faster for the model with 3 processes, although none of the tools scaled to 4 processes.

Overall, the experiments show that our algorithm is competitive with the state of the art tools; while it does not improve the performance uniformly on all considered benchmarks, it does allow us to solve instances that are not solvable by other tools, and sometimes to improve performance both compared to a zone-based approach (Uppaal) and SAT-based algorithms (nuXmv).

3 Compositional Controller Synthesis

3.1 Preliminaries

Games. A *finite safety game* is a pair $(\mathcal{G}, \mathsf{Bad})$ where $\mathcal{G}$ is an LTS $(Q_E \dot\cup Q_C, q_0, \Sigma, T)$ with the set of states given as a partition $Q_E \dot\cup Q_C$, namely, *Environment states* (Q_E), and *Controller states* (Q_C), and $\mathsf{Bad} \subseteq Q_E \dot\cup Q_C$ is an *objective*. The game is played between two players, namely, *Controller* and *Environment*. At each state $q \in Q_C$, Controller determines the successor by choosing an edge from q, and Environment determines the successor from states $q \in Q_E$. A *strategy* for Controller (resp. Environment) maps finite runs of $(Q_E \dot\cup Q_C, q_0, \Sigma, T)$ ending in Q_C (resp. Q_E) to an edge leaving the last state. A pair of strategies, one for each player, induces a unique infinite run from the initial state. A run is *winning* for Controller if it does not visit Bad; it is winning for Environment otherwise. A *winning strategy* for Controller is such that for all Environment strategies, the run induced by the two strategies is winning for Controller. Symmetrically, Environment has a winning strategy if for all Controller strategies, the induced run is winning. A strategy is *positional* (a.k.a. memoryless) if it only depends on the last state of the given run.

The parallel composition of $(\mathcal{G}, \mathsf{Bad})$ and a deterministic finite automaton $\mathcal{F} = (Q', q'_0, \Sigma, T', F)$ on alphabet Σ is a new game whose LTS is $\mathcal{G} \| \mathcal{F}$ in which the Controller states are $Q_C \times Q'$, the Environment states are $Q_E \times Q'$, and the objective is $\mathsf{Bad} \times F$ (Notice that Controller thus has a safety objective).

Finite games were extended to the real-time setting as *timed games* [40,4]. A *timed game* is a timed automaton $\mathcal{T} = (L_E \dot\cup L_C, \ell_0, \Sigma, \mathrm{Inv}, \mathcal{C}, E, \mathsf{Bad})$ with the exception that its edges are labeled by $\Sigma \cup \{\epsilon\}$ (and not just by Σ as in the previous section), and the locations are partitioned as $L_E \dot\cup L_C$ into *Environment locations* and *Controller locations*. The semantics is defined by letting Environment choose the delay and the edge to be taken at locations L_E, while Controller choose these from L_C. Formally, a *strategy* for Environment (resp. Controller) is a function which associates a run that ends in L_E (resp. L_C) to a pair of delay and an edge enabled from the state reached after the delay. A run is winning for Controller if it does not visit Bad. A Controller (resp. Environment) strategy is *winning* for objective Bad if for all Environment (resp. Controller) strategies, the induced run from the initial state is winning (resp. not winning) for Controller. A run r

is *compatible* with a strategy $\mathfrak{S}$ for Controller (resp. Environment) if there exists an Environment (resp. Controller) strategy $\mathfrak{S}'$ such that r is induced by $\mathfrak{S}, \mathfrak{S}'$.

The parallel composition of a finite safety game $(\mathcal{G}, \mathsf{Bad})$ and a timed automaton $\mathcal{T} = (L, \ell_0, \Sigma, \mathrm{Inv}, \mathcal{C}, E, F)$ on common alphabet Σ is the timed game $\mathcal{G} \| \mathcal{T}$ where Controller locations are $Q_C \times L$, and Environment locations are $Q_E \times L$.

Positional strategies exist both for reachability and safety objectives in finite and timed games. Both finite and timed games are known to be *determined* for reachability and safety objectives. For instance, if Controller does not have a winning strategy for the safety objective, then Environment has a strategy ensuring the reachability of Bad [40,4].

Target Timed Game Instances. We consider controller synthesis problems described as timed games in the form of $(\mathcal{G} \| \mathcal{T}, \mathsf{Bad} \times F)$ where $(\mathcal{G}, \mathsf{Bad})$ is a finite safety game, and $\mathcal{T}$ is a timed automaton. In addition, we assume that $\mathcal{G} \| \mathcal{T}$ is *Controller-silent*, defined as follows.

Definition 1. *The timed game $(\mathcal{G} \| \mathcal{T}, \mathsf{Bad} \times F)$ on alphabet Σ is Controller-silent if 1) all Controller transitions are silent; and 2) all Controller locations in $\mathcal{T}$ are* urgent*, that is, an invariant ensures that no time can elapse.*

Hence, we again separate the game $\mathcal{G}$ defined on a possibly large discrete state space while real-time constraints are separately given in $\mathcal{T}$.

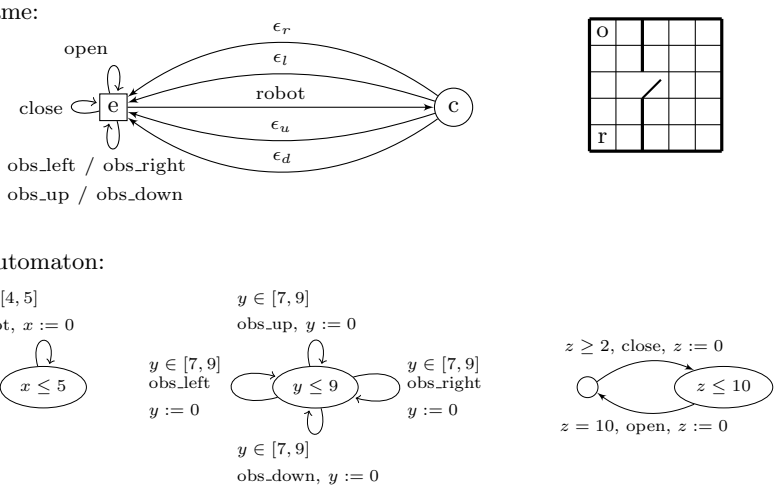


Fig. 3. The sketch of a timed game $\mathcal{G} \| \mathcal{T}$ modelling a planning problem. The finite game models a robot and an obstacle moving in a grid world as shown on top right. The cells r and o show, respectively, the initial positions of the robot and the obstacle. The robot cannot cross walls (shown in thick segments), and can only cross the door if it is open. Here four silent transitions were marked with $\epsilon_r, \epsilon_l, \epsilon_u, \epsilon_d$ for readability; in reality, these are all labeled by ϵ.

The intuition behind the semantics is the following: because the game is played in $\mathcal{G}\|\mathcal{T}$ and $\mathcal{G}$ is Controller-silent, the timed automaton model $\mathcal{T}$ is only used to disallow some of the Environment transitions according to real-time constraints, while Controller's actions are instantaneous responses to Environment's actions and thus are unaffected by the contraints of $\mathcal{T}$. One can think of the timed automaton as some form of scheduler that schedules uncontrollable events in the system, so the order of these is determined by Environment. This assumption is restrictive; for instance, this excludes controller synthesis problems where the control strategy is to choose delays to execute some events. Nonetheless, this asymetric view enables a one-sided abstraction framework presented in the next section, where Environment transitions are approximated by a DFA.

An example is given in Figure 3. The finite game drawn here only shows the structure of the game. It has, in addition, integer variables `rob_x, rob_y, obs_x, obs_y` encoding the positions of the robot and of the obstacle, and a Boolean variable `door` to encode the state of the door. The state e belongs to Environment, which can move the obstacle in any direction, close or open the door, or let the robot move by going to state c. The state c belongs to Controller. All its leaving transitions are silent, and correspond to moving the robot in four directions. These transitions have preconditions, not shown in the figure, that check whether the moves are possible, and have updates that modify the state variables. The timed automaton, given as a network of three timed automata, determine the timings of these events. One can notice, for example, that the robot is moving faster than the obstacle, and that whenever the door is closed, it remains so for 10 time units.

3.2 One-Sided Abstraction

Thanks to the assumption we make on considered timed games, we show that by replacing $\mathcal{T}$ by a DFA H that is an overapproximation, we obtain an abstract game in which Controller strategies can be transferred to the original game. This is formalized in the next lemma (the proof is in the appendix).

Lemma 2. *Consider a Controller-silent timed game* $(\mathcal{G}\|\mathcal{T}, \mathsf{Bad}\times F)$*, and a complete DFA* H *with accepting states* F_H*, satisfying* $\mathcal{L}(\mathcal{T}) \subseteq \mathcal{L}(H)$*.*

- *If Controller wins* $(\mathcal{G}\|H, \mathsf{Bad}\times F_H)$*, then it wins* $(\mathcal{G}\|\mathcal{T}, \mathsf{Bad}\times F)$*.*
- *If Environment wins* $(\mathcal{G}\|\mathcal{T}, \mathsf{Bad}\times F)$*, then it wins* $(\mathcal{G}\|H, \mathsf{Bad}\times F_H)$*, and has a strategy in* $(\mathcal{G}\|H, \mathsf{Bad}\times F_H)$ *whose all compatible runs have traces in* $\mathcal{L}(\mathcal{T})$*.*

Note that in the above lemma, it is crucial that the game is Controller-silent. In fact, if Controller could take edges that synchronize with $\mathcal{T}$, then we may not be able to apply a strategy in $\mathcal{G}\|H$ to $\mathcal{G}\|\mathcal{T}$, since such a strategy may prescribe traces that are not accepted in $\mathcal{T}$. Moreover, if Controller locations are not urgent, we would not know how to select the delays when mapping the strategy to $\mathcal{G}\|\mathcal{T}$.

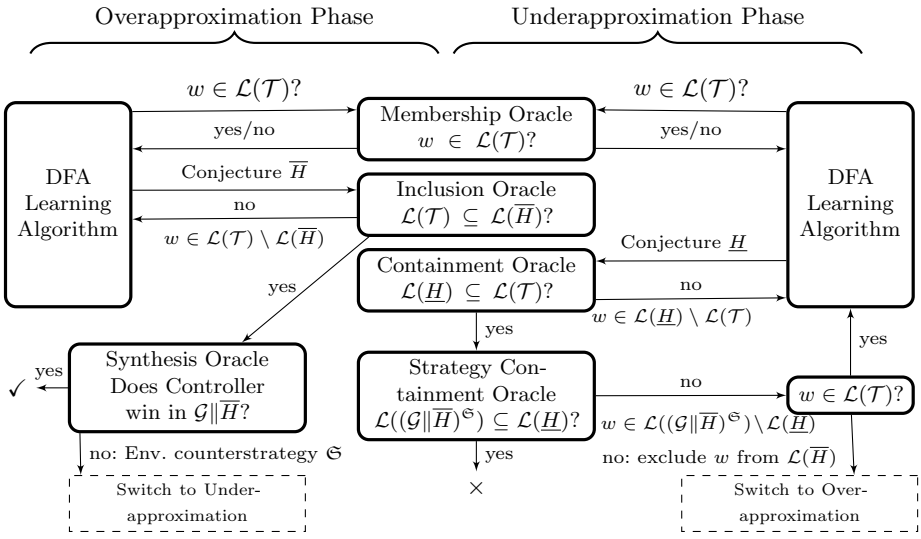


Fig. 4. The learning-based compositional controller synthesis algorithm for the input timed game $\mathcal{G} \| \mathcal{T}$, with $\mathcal{G}$ a Controller-silent finite game, and $\mathcal{T}$ a label-deterministic timed automaton. Two automata learning algorithms run in parallel to learn under- and over-approximations $\underline{H}$ and $\overline{H}$ such that $\underline{H} \subseteq \mathcal{L}(\mathcal{T}) \subseteq \overline{H}$.

3.3 Learning-Based Compositional Controller Synthesis Algorithm

We now present our compositional controller synthesis algorithm whose overview is given in Figure 4. The algorithm for controller synthesis is more involved than the model checking algorithm due to the alternating semantics for two players in games. It consists in two phases that alternate: the overapproximation phase, and the underapproximation phase. Each phase runs a DFA learning algorithm which is interrupted when we switch to the other phase, and continued when we switch back, until a decision is made. Together, both phases maintain two approximations, $\underline{H}$ and $\overline{H}$, such that $\mathcal{L}(\underline{H}) \subseteq \mathcal{L}(\mathcal{T}) \subseteq \mathcal{L}(\overline{H})$.

The objective of the overapproximation phase is to attempt to learn a DFA $\overline{H}$ satisfying $\mathcal{L}(\mathcal{T}) \subseteq \overline{H}$, and such that Controller wins in $\mathcal{G} \| \overline{H}$. The learning algorithm uses membership and inclusion oracles just like in Section 2.2. Once such a candidate DFA $\overline{H}$ is found, the synthesis oracle checks, using finite-state techniques, whether Controller has a winning strategy in $\mathcal{G} \| \overline{H}$. If this is the case, we stop and conclude that Controller wins in $\mathcal{G} \| \mathcal{T}$ by Lemma 2. Otherwise, Environment has a winning strategy $\mathfrak{S}$ in this game; and we switch to the underapproximation phase.

The goal of the underapproximation is to check whether the given Environment strategy $\mathfrak{S}$ can be proved to be spurious. Intuitively, we would like to check whether $\mathcal{L}((\mathcal{G} \| \overline{H})^{\mathfrak{S}}) \subseteq \mathcal{L}(\mathcal{T})$ and reject if this is the case. In fact, by Lemma 2, we know that a winning Environment strategy in $\mathcal{G} \| \mathcal{T}$ implies that there is such a strategy $\mathfrak{S}$. This is the source of incompleteness of our algorithm, since this

condition is necessary but not sufficient for Environment to win; that is, the condition does not guarantee that Environment actually wins in $\mathcal{G}\|\mathcal{T}$.

While $\mathcal{L}((\mathcal{G}\|\overline{H})^{\mathfrak{S}}) \subseteq \mathcal{L}(\mathcal{T})$ can be checked with a timed automaton model checker (see *Checking Containment* below), this would mean exploring the large state space due to $\mathcal{G}$. Since we want to avoid using timed automata model checkers on such large instances, we rather learn an underapproximation $\underline{H}$ of $\mathcal{L}(\mathcal{T})$ using the membership and containment oracles, and use a finite-state model checker to check $\mathcal{L}((\mathcal{G}\|\overline{H})^{\mathfrak{S}}) \subseteq \mathcal{L}(\underline{H})$. Note that although the learning process does require inclusion checks of the form $\mathcal{L}(\underline{H}) \subseteq \mathcal{L}(\mathcal{T})$, this check is feasible with a timed automaton model checker since $\underline{H}$ is typically much smaller than $\mathcal{G}$. If the above check passes, then we reject the instance, that is, we declare the system not controllable. Otherwise, some trace w appears in $\mathcal{L}((\mathcal{G}\|\overline{H})^{\mathfrak{S}})$ but not in $\mathcal{L}(\underline{H})$. If $w \in \mathcal{L}(\mathcal{T})$, then we require that w be included in $\underline{H}$, and continue the learning process. Otherwise, $\mathfrak{S}$ is not valid since it induces w which is not in $\mathcal{L}(\mathcal{T})$. So we interrupt the current phase and switch back to the overapproximation phase requiring w to be removed from $\overline{H}$.

Membership and inclusion oracles are implemented with a timed automata model checker. Here, the synthesis oracle can be any finite game solver; we just need the capability of computing the controlled system $(\mathcal{G}\|\overline{H})^{\mathfrak{S}}$. Such a system is finite-state, so the strategy containment oracle can be implemented using a finite-state model checker (since $\underline{H}$ is deterministic and can thus be complemented). It remains to explain how the containment oracle is implemented.

Checking Containment $\mathcal{L}(\underline{H}) \subseteq \mathcal{L}(\mathcal{T})$. First, notice that, even with determinism assumptions on $\mathcal{T}$, the untimed language of the timed automaton complement of $\mathcal{T}$ is not the complement of $\mathcal{L}(\mathcal{T})$. To see this, consider a timed automaton with a single state which is both initial and accepting, a single clock x, and a self-loop with guard $x = 1$, labeled by σ. Then, both $\mathcal{L}(\mathcal{T})$ and $\mathcal{L}(\mathcal{T}^c)$ are the language σ^* where $\mathcal{T}^c$ denotes the timed automaton complement.

Nevertheless, assuming the label-determinism of $\mathcal{T}$, this check can be done by a simple adaptation of a zone-based exploration algorithm, as follows. Let us assume that accepting states are reachable from all states of $\underline{H}$, which can be ensured by a preprocessing step. We start exploring the timed automaton $\underline{H}\|\mathcal{T}$ using a zone-based exploration algorithm [11]. Consider any search node $((q_{\underline{H}}, q_{\mathcal{T}}), Z)$ encountered during the exploration algorithm, reachable by the trace w, where $(q_{\underline{H}}, q_{\mathcal{T}})$ is a location of $\underline{H}\|\mathcal{T}$, and Z a zone. The exploration algorithm generates all available successors for $\sigma \in \Sigma$. We make the following additional check: If there is $\sigma' \in \Sigma$ such that $q_{\underline{H}}$ has a successor by σ', but not $\mathcal{T}$ (either because there is no such edge, or because the guard of the unique edge labeled by σ' is not satisfied by Z), then we stop and return the trace $w\sigma' \in \mathcal{L}(\underline{H}) \setminus \mathcal{L}(\mathcal{T})$ as a counterexample to containment. If no such label can be found, the zone-based exploration will terminate and the algorithm confirms the containment.

As an alternative, one can use testing such as the Wp-method [37] to establish the containment, as it is customary in DFA learning. In this case, the answer is approximate in the sense that the conformance test can fail to detect that

containment does not hold. However, this does not affect the soundness of the overall algorithm since it can only increase false negatives.

3.4 Experiments

Our tool accepts instances $\mathcal{G}\|\mathcal{T}$ where $\mathcal{G}$ is given as a Verilog module, and $\mathcal{T}$ as a TChecker timed automaton. Some of the inputs of the Verilog module are *uncontrollable* (chosen by Environment), some others are controllable (chosen by Controller). We use outputs of the Verilog module to define the synchronization labels Σ; while TChecker models tag each transition with such a label.

Table 2. The results of the controller synthesis experiments. The columns #Clks, #C, #M respectively show the number of clocks in the model, the numbers of conjectures and membership queries made by the compositional algorithm; while $|\overline{H}|$, $|\underline{H}|$ show the sizes of the DFAs learned by the two phases.

		Compositional Algorithm					Uppaal TIGA	Controllable
	#Clks	#C	#M	$\lvert\overline{H}\rvert$	$\lvert\underline{H}\rvert$	Time	Time	
Scheduling genbuf A	3	50	2178	114		**26s**	—	yes
Scheduling genbuf B	3	40	1734	96		**15s**	—	yes
Scheduling genbuf C	3	45	1503	88		**4s**	—	yes
Scheduling counter64 D	3	54	2098	108		26s	**14s**	yes
Scheduling counter64 E	3	37	1454	83		**16s**	19s	yes
Scheduling counter64 F	3	19	21391	19	19	89s	**0s**	no
Planning genbuf A	2	2	17	4		**6s**	—	yes
Planning genbuf B	2	2	24	5		**9s**	—	yes
Planning genbuf C	2	9	1156	5	5	**266s**	—	no
Planning stateless D	2	3	50	9		**2s**	22s	yes
Planning stateless E	2	2	17	4		**2s**	4s	yes
Planning stateless F	2	8	973	5	5	10s	**2s**	no

Membership, inclusion, and containment queries are answered by TChecker. For the synthesis oracle, we used the game solver Abssynthe [12]. Abssynthe's input format is the and-inverter graphs format (AIG). For translating Verilog modules to AIG circuits, we use berkeley-abc and yosys. Abssynthe is able to compute the winning strategy $\mathfrak{S}$ for the winning player; it also computes the system controlled by $\mathfrak{S}$ in this case as an AIG circuit. The strategy containment oracle is implemented using NuSMV; since $\underline{H}$ is deterministic, one can complement it, and check whether the intersection with $(\mathcal{G}\|\overline{H})^{\mathfrak{S}}$ is empty.

The tool uses two Java threads to implement both learning phases, which are interrupted and continued while switching phases. Note that the very first learning step of $\overline{H}$ and $\underline{H}$ can be parallelized since the first underapproximation conjecture $\underline{H}$ does not depend on $\mathfrak{S}$.

We evaluate our algorithm with two classes of benchmarks. The only tool to which we compare is Uppaal-TIGA [6] since Synthia [45] is not available anymore, and we are not aware of any other timed game solver.

In the scheduling benchmarks, there are two sporadic tasks that arrive nondeterministically, but constrained by the timed automaton. The controller must schedule these using two machines which have internal states, modeled either by a simple 8-bit counter, or by a `genbuf` circuit from the SYNTCOMP database. The scheduling duration depends on the internal state: some states require executing two external tasks, some others require executing three. The external task has a nondeterministic duration constrained by the timed automaton. The internal states change when a task is finalized. The controller loses if all machines are busy upon the arrival of a new task, or if it schedules a task on a busy machine. Uppaal TIGA was able to solve the counter models since they induce a smaller state space, but failed at the genbuf models. The compositional algorithm could efficiently handle these models. Uppaal was generally able to determine very quickly when the model is not controllable by finding a small counterstrategy, while the compositional algorithm had a overhead: it had to learn $\overline{H}$ and $\underline{H}$ before it can find and check the counterexample.

In the planning benchmarks, a robot and an obstacle is moving in a 6×6 grid (or 9×9 for the stateless case). Each agent can decide to move to an adjacent cell when they are scheduled, and the scheduling times are determined by a timed automaton. The goal of the robot is to avoid the obstacles. In the genbuf case, there are moreover internal states that can cause a glitch and prevent the agents from performing their moves, depending on their states. Uppaal TIGA was not able to manage the large state space unlike the compositional algorithm in this case, but both were able to solve the stateless case.

4 Conclusion

Related Works Perhaps the most closely related approach to our compositional model checking algorithm is trace abstraction refinement [26]. This was originally applied to program verification, and consists in building a network of finite automata that recognizes the program's control flow paths that are infeasible. One refines this language by model checking the control flow graph intersected with the complement of the automaton. Thus, the semantics of the variables of the program are abstractly represented by the finite automaton. This idea was applied to timed automata as well [54,15]. However, the generalization of the counterexamples which ensures convergence turns out to be less effective in timed automata. We attempted at obtaining an implementation, but could only confirm the poor performance for model checking timed automata as in [15] (we do not include these results here). It might be that simpler graph structures such as control flow graphs of programs are necessary for this approach to scale; further investigation is also necessary to study better generalization methods.

The learning-based compositional reasoning approach of [44] is also related to counter-example guided abstraction refinement (CEGAR) [18]. In fact, the automata learning algorithm builds an overapproximation of one of the components, and refines it as needed, guided by counterexamples. The difference is that,

instead of using predicates, one uses automata to represent the overapproximation. A discussion can also be found in [44].

Learning algorithms for event-recording automata, a subset of timed automata were studied in [25]. The algorithm of [44] was extended for these automata in [36]. In the context of parameter synthesis with learning, parameterized systems were seen as a parallel composition of a non-parameterized component, and a parameterized component in [2].

Other approaches targeting the formal verification of real-time systems with large discrete state spaces include encodings of timed automata semantics in Boolean logic include [34,49]. An extension of and-inverter graphs were used in [20] that uses predicates to represent the state space of linear hybrid automata.

The abstract interpretation of games were studied in [28] that presents a theory allowing one to define under- and over-approximations. Abstraction-refinement algorithms based on counterexamples were given in [27,21]. These ideas were applied to timed games in [45]. Several abstraction-refinement and compositional algorithms were given in [12,13] for solving finite-state games given as Boolean circuits. The synthesis competition gathers every year researchers who present their game solvers [32,33].

Perspectives The algorithm we presented builds finite-state abstractions of real-time constraints, that it represents as DFA. The approach is well adapted when the interaction alphabet between $\mathcal{A}$ and $\mathcal{T}$ is small; this is the case, for instance, for distributed systems where the time constraints are used to describe the approximate period with which each process communicates with its neighbors; so the alphabet contains only a few symbols per process. Some of the benchmarks we considered are models of such systems. The approach is less convenient for time-intensive systems such as, say, job shop scheduling problems where a separate alphabet symbol is needed for each task.

As future work, we would like to understand when various abstraction schemes are efficient among the approach presented here, the predicate-abstraction approach, and zone-based state-space exploration. Currently, all algorithms fail in some benchmarks. Understanding the strengths of each algorithm might help designing a uniformly better solution. Currently, we can only verify linear properties; one might verify branching-time properties by learning automata with a stronger notion of equivalence such as bisimulation. In fact, an important limitation is due to learning being slow for large alphabets. Our setting could be extended to deal with large or symbolic alphabets *e.g.* [38,39].

For synthesis, our setting is currently restricted by the abstractions we use since when the algorithm rejects the instace, we cannot conclude whether the system is controllable or not. Using both the under- and overapproximations within the finite-state synthesis, for instance, using the three-valued abstraction approach [21] might allow us to render the approach complete, and to consider a larger class of timed games such as those that allow Controller to select nonzero delays.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. Étienne André and Shang-Wei Lin. Learning-based compositional parameter synthesis for event-recording automata. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 17–32, Cham, 2017. Springer International Publishing.
3. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
4. Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, volume 999 of *LNCS*, pages 1–20. Springer, 1995.
5. Gerd Behrmann, Patricia Bouyer, Kim G. Larsen, and Radek Pelanek. Lower and upper bounds in zone-based abstractions of timed automata. *Int. J. Softw. Tools Technol. Transf.*, 8(3):204–215, June 2006.
6. Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In *International Conference on Computer Aided Verification*, pages 121–125. Springer, 2007.
7. Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. UPPAAL-Tiga: Time for playing games! In *Proc. 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 121–125. Springer, 2007.
8. Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*, pages 125–126, 2006.
9. V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys=esterel+kronos. a tool for verifying real-time properties of embedded systems. In *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228)*, volume 3, pages 2875–2880 vol.3, 2001.
10. Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for BDD-based verification of real-time systems. In *Proc. 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 122–125. Springer, 2003.
11. Patricia Bouyer, Paul Gastin, Frédéric Herbreteau, Ocan Sankur, and B. Srivathsan. Zone-based verification of timed automata: extrapolations, simulations and what next? In *20th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2022)*. Springer, 2022.
12. Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin, and Ocan Sankur. Abssynthe: abstract synthesis from succinct safety specifications. In Krishnendu Chatterjee, Rüdiger Ehlers, and Susmit Jha, editors, *Proceedings 3rd Workshop on Synthesis (SYNT'14)*, volume 157 of *Electronic Proceedings in Theoretical Computer Science*, pages 100–116. Open Publishing Association, 2014.
13. Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin, and Ocan Sankur. Compositional algorithms for succinct safety games. In Pavol Černý, Viktor Kuncak, and Madhusudan Parthasarathy, editors, *Proceedings Fourth Workshop on Synthesis (SYNT'15), San Francisco, CA, USA, 18th July 2015*, volume 202 of *Electronic Proceedings in Theoretical Computer Science*, pages 98–111. Open Publishing Association, 2016.

14. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proc. 16th International Conference on Concurrency Theory (CONCUR'05)*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2005.
15. Franck Cassez, Peter Gjøl Jensen, and Kim Guldstrand Larsen. Verification and parameter synthesis for real-time programs using refinement of trace abstraction. *Fundam. Informaticae*, 178(1-2):31–57, 2021.
16. Alessandro Cimatti, Alberto Griggio, Enrico Magnago, Marco Roveri, and Stefano Tonetta. Extending nuxmv with timed transition systems and timed temporal properties. In *International Conference on Computer Aided Verification*, pages 376–386. Springer, 2019.
17. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *Lecture Notes in Computer Science*, pages 46–61, 2014.
18. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.
19. Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.
20. Werner Damm, Henning Dierks, Stefan Disch, Willem Hagemann, Florian Pigorsch, Christoph Scholl, Uwe Waldmann, and Boris Wirtz. Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. *Science of Computer Programming*, 77(10):1122–1150, 2012.
21. Luca de Alfaro and Pritam Roy. Solving games via three-valued abstraction refinement. *Information and Computation*, 208(6):666–676, 2010. Special Issue: 18th International Conference on Concurrency Theory (CONCUR 2007).
22. Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Robust stabilizing leader election. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 219–233, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
23. Henning Dierks. *Time, abstraction and heuristics - automatic verification and planning of timed systems using abstraction and heuristics*, volume 01-06 of *Berichte aus dem Department für Informatik / Universität Oldenburg / Fachbereich Informatik*. 2006.
24. Rudiger Ehlers, Daniel Fass, Michael Gerke, and Hans-Jorg Peter. Fully symbolic timed model checking using constraint matrix diagrams. In *Proc. 31th IEEE Real-Time Systems Symposium (RTSS'10)*, pages 360–371. IEEE Computer Society Press, 2010.
25. Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010.
26. Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *International Static Analysis Symposium*, pages 69–85. Springer, 2009.
27. Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. In *International Colloquium on Automata, Languages, and Programming*, pages 886–902. Springer, 2003.
28. Thomas A. Henzinger, Rupak Majumdar, Freddy Mang, and Jean-François Raskin. Abstract interpretation of game properties. In Jens Palsberg, editor, *Static Analysis*, pages 220–239, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

29. Frédéric Herbreteau and Gérald Point. The TChecker tool and librairies. `https://github.com/ticktac-project/tchecker`.
30. Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: a redundancy-free approach to active automata learning. In *International Conference on Runtime Verification*, pages 307–322. Springer, 2014.
31. Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 487–495, Cham, 2015. Springer International Publishing.
32. Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, et al. The first reactive synthesis competition (syntcomp 2014). *International journal on software tools for technology transfer*, 19(3):367–390, 2017.
33. Swen Jacobs, Guillermo A Perez, Remco Abraham, Veronique Bruyere, Michael Cadilhac, Maximilien Colange, Charly Delfosse, Tom van Dijk, Alexandre Duret-Lutz, Peter Faymonville, et al. The reactive synthesis competition (syntcomp): 2018-2021. *arXiv preprint arXiv:2206.00251*, 2022.
34. Roland Kindermann, Tommi Junttila, and Ilkka Niemela. Modeling for symbolic analysis of safety instrumented systems with clocks. In *Proc. 11th International Conference on Application of Concurrency to System Design (ACSD'11)*, pages 185–194. IEEE Computer Society Press, 2011.
35. Branislav Kusy and Sherif Abdelwahed. Ftsp protocol verification using spin. May 2006.
36. Shang-Wei Lin, Étienne André, Yang Liu, Jun Sun, and Jin Song Dong. Learning assumptions for compositional verification of timed systems. *Transactions on Software Engineering*, 40(2):137–153, mar 2014.
37. Gang Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, 1994.
38. Oded Maler and Irini-Eleftheria Mens. Learning regular languages over large alphabets. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 485–499, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
39. Oded Maler and Irini-Eleftheria Mens. A generic algorithm for learning symbolic automata from membership queries. In Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay, and Radu Mardare, editors, *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, pages 146–169, Cham, 2017. Springer International Publishing.
40. Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.
41. Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 39–49, New York, NY, USA, 2004. ACM.
42. A. I. McInnes. Model-checking the flooding time synchronization protocol. In *Control and Automation, 2009. ICCA 2009. IEEE International Conference on*, pages 422–429, Dec 2009.
43. Truong Khanh Nguyen, Jun Sun, Yang Liu, Jin Song Dong, and Yan Liu. Improved BDD-based discrete analysis of timed systems. In *Proc. 20th International Symposium on Formal Methods (FM'12)*, volume 7436, pages 326–340. Springer, 2012.

44. Corina S Păsăreanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.
45. Hans-Jörg Peter, Rüdiger Ehlers, and Robert Mattmüller. Synthia: Verification and synthesis for timed automata. In *International Conference on Computer Aided Verification*, pages 649–655. Springer, 2011.
46. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
47. Victor Roussanaly, Ocan Sankur, and Nicolas Markey. Abstraction refinement algorithms for timed automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification (CAV'19)*, pages 22–40, Cham, 2019. Springer International Publishing.
48. Ocan Sankur and Jean-Pierre Talpin. An abstraction technique for parameterized model checking of leader election protocols: Application to FTSP. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 23–40, 2017.
49. Sanjit A. Seshia and Randal E. Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 154–166, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
50. Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In *Proceedings of the 21th International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.
51. Yann Thierry-Mieg. Symbolic model-checking using ITS-tools. In *Proc. 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, pages 231–237. Springer, 2015.
52. Wolfgang Thomas. On the synthesis of strategies in infinite games. In Ernst W. Mayr and Claude Puech, editors, *STACS 95*, pages 1–13, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
53. Farn Wang. Symbolic verification of complex real-time systems with clock-restriction diagram. In *Proc. 21st International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01)*, volume 197 of *IFIP Conference Proceedings*, pages 235–250. Kluwer, 2001.
54. Weifeng Wang and Li Jiao. Trace abstraction refinement for timed automata. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis*, pages 396–410, Cham, 2014. Springer International Publishing.

A Appendix

Proof (Proof of Lemma 1). This follows from the observation that $\mathcal{L}(\mathcal{A}\|\mathcal{T}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{T})$, and $\mathcal{L}(\mathcal{A} \parallel H) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(H)$.

Proof (of Lemma 2). Consider a winning strategy $\mathfrak{S}$ for Controller in $(\mathcal{G}\|H, \mathsf{Bad}\times F_H)$.

Let us define a set A of runs of $\mathcal{G}\|\mathcal{T}$ as well as a mapping f from A to the runs of $\mathcal{G}\|H$. We will at the same time define a strategy $\mathfrak{S}'$ on $\mathcal{G}\|\mathcal{T}$ whose compatible

runs will be A. We will do so by imitating $\mathfrak{S}$ using the mapping f. We define f on the runs inductively for increasing lengths.

Initially, we let $f((q_0^{\mathcal{G}}, q_0^{\mathcal{T}})) = (q_0^{\mathcal{G}}, q_0^H)$ where $q_0^{\mathcal{T}}$ is the initial state of $\mathcal{T}$, and $q_0^{\mathcal{G}}$ that of $\mathcal{G}$, and respectively for q_0^H and H.

For $n \geq 0$, consider a run $\rho = (q_0^{\mathcal{G}}, q_0^{\mathcal{T}}) \dots (q_n^{\mathcal{G}}, q_n^{\mathcal{T}})$ of A and assume that $f(\rho) = (q_0^{\mathcal{G}}, q_0^H) \dots (q_n^{\mathcal{G}}, q_n^H)$ is already defined. We distinguish two cases.

If $(q_n^{\mathcal{G}}, q_n^{\mathcal{T}})$ belongs to Environment, then for all its successors $(q_{n+1}^{\mathcal{G}}, q_{n+1}^{\mathcal{T}})$ with label σ, we add $\rho \cdot (q_{n+1}^{\mathcal{G}}, q_{n+1}^{\mathcal{T}})$ to A, and we define $f(\rho \cdot (q_{n+1}^{\mathcal{G}}, q_{n+1}^{\mathcal{T}})) = (q_0^{\mathcal{G}}, q_0^H) \dots (q_n^{\mathcal{G}}, q_n^H)(q_{n+1}^{\mathcal{G}}, q_{n+1}^H)$ where q_{n+1}^H is the unique successor of q_n^H with label $\sigma \in \Sigma$, and $q_{n+1}^H = q_n^H$ if $\sigma = \epsilon$. Such a successor exists because H is complete, and it is unique because H is deterministic.

Otherwise, $(q_n^{\mathcal{G}}, q_n^{\mathcal{T}})$ belongs to Controller, and we consider the move prescribed by $\mathfrak{S}$ at $(q_0^{\mathcal{G}}, q_0^H) \dots (q_n^{\mathcal{G}}, q_n^H)$ in $\mathcal{G} \| H$. This consists in a silent edge e of $\mathcal{G}$ by assumption. Let $q_{n+1}^{\mathcal{G}}$ be the successor of $q_n^{\mathcal{G}}$ through e. We then define $f(\rho \cdot (q_{n+1}^{\mathcal{G}}, q_{n+1}^{\mathcal{T}})) = (q_0^{\mathcal{G}}, q_0^H) \dots (q_n^{\mathcal{G}}, q_n^H)(q_{n+1}^{\mathcal{G}}, q_n^H)$. In this case, we let $\mathfrak{S}'$ choose this move from ρ after a 0 delay (recall that this is the only delay possible for Controller).

Notice that f maps runs that have the same trace and whose first components are identical.

Consider any infinite run ρ of $\mathcal{G} \| \mathcal{T}$ compatible with σ'. For any finite prefix ρ' of ρ, $f(\rho')$ is a run of $\mathcal{G} \| H$ compatible with $\mathfrak{S}$. By assumption, $f(\rho')$ does not end in $\mathsf{Bad} \times F_H$. If the first component of the last state of $f(\rho')$ is not in Bad, then this is also the case for ρ'. Otherwise, the second component must be outside of F_H, that is, the trace of $f(\rho')$ is not in $\mathcal{L}(H)$. But then the trace of ρ' is not in $\mathcal{L}(\mathcal{T})$ by $\mathcal{L}(\mathcal{T}) \subseteq \mathcal{L}(H)$. Therefore, ρ is winning for Controller.

The first part of the second statement follows from the determinacy of the games we consider. In fact, if Environment does not win $(\mathcal{G} \| H, \mathsf{Bad} \times F_H)$, then Controller wins. By the first case, Controller wins $(\mathcal{G} \| \mathcal{T}, \mathsf{Bad} \times F)$, so Environment loses.

It remains to show that when Environment wins in $(\mathcal{G} \| \mathcal{T}, \mathsf{Bad} \times F)$, then it has a strategy in $(\mathcal{G} \| H, \mathsf{Bad} \times F_H)$ whose all compatible runs have their traces in $\mathcal{L}(\mathcal{T})$. This can be done by a construction that is similar to the first case. Let $\mathfrak{S}$ be a winning strategy for Environment in $\mathcal{G} \| \mathcal{T}$. We define a set B of runs of $\mathcal{G} \| H$ and a mapping g from B to the runs of $\mathcal{G} \| \mathcal{T}$. We also define $\mathfrak{S}'$ as an Environment strategy in $\mathcal{G} \| H$ whose compatible runs are exactly B. First, B contains the initial state $(q_0^{\mathcal{G}}, q_0^H)$ and we define $g((q_0^{\mathcal{G}}, q_0^H)) = (q_0^{\mathcal{G}}, q_0^{\mathcal{T}})$. Consider a run $\rho = (q_0^{\mathcal{G}}, q_0^H) \dots (q_n^{\mathcal{G}}, q_n^H)$ of B on which g is already defined. Assume that it ends in an Environment state. We consider the delay and edge (d, e) prescribed by $\mathfrak{S}$ to $g(\rho)$. Let $(q_{n+1}^{\mathcal{G}}, q_{n+1}^{\mathcal{T}})$ be the successor of $g(\rho)$ after (d, e). Let q_{n+1}^H be the successor of q_n^H on the label of e, and $q_{n+1}^H = q_n^H$ if the label is ϵ. We add $\rho(q_{n+1}^{\mathcal{G}}, q_{n+1}^H)$ to B, and define $g(\rho(q_{n+1}^{\mathcal{G}}, q_{n+1}^H)) = g(\rho)(q_{n+1}^{\mathcal{G}}, q_{n+1}^{\mathcal{T}})$. We define $\mathfrak{S}'$ so that it assings $(q_{n+1}^{\mathcal{G}}, q_{n+1}^H)$ to ρ. If ρ ends in a Controller state, then for all successors $q_{n+1}^{\mathcal{G}}$ of ρ (via silent transitions), we add $\rho(q_{n+1}^{\mathcal{G}}, q_n^H)$ to B and define $g(\rho(q_{n+1}^{\mathcal{G}}, q_n^H)) = g(\rho)(q_{n+1}^{\mathcal{G}}, q_n^{\mathcal{T}})$. It follows that $\mathfrak{S}'$ is winning for Environment

in $\mathcal{G} \| H$. As in the proof of the first statement, $\mathfrak{S}$ and $\mathfrak{S}'$ induce the same traces by the mapping g, so we have that all compatible runs under $\mathfrak{S}'$ are in $\mathcal{L}(\mathcal{T})$. □

B Description of Benchmarks

The STS benchmarks are programming logic controller models from [23] and are part of the TChecker benchmark database. These are known to be difficult to model check. The compositional algorithm was faster in STS-2, but none of the tools could solve STS-3.

The real-time broadcast protocol (rt-broadcast) implements a distributed algorithm made of $n = 3$ processes that wake up within a period interval, and stay active within a given time interval. If at least $m = 2$ of them are awake at the same time, they perform one step of a computation together and go back to sleep. The specification is whether a particular common configuration is reachable within a time bound. Uppaal could not solve these benchmarks. nuXmv could solve these although the compositional algorithm was often faster.

In the priority-based scheduling examples, a priority-based scheduling algorithm schedules tasks on a single machine. The interarrival times depend on the internal state of the processes which evolve over time. The internal states are modeled again by circuits from the SYNTCOMP benchmarks. The first two models have two tasks, and the last one has three. nuXmv performed very well on these benchmarks while the compositional algorithm failed on instances with three processes.