An Efficient TCTL Model Checking Algorithm and A Reduction Technique for Verification of Timed Actor Models

Ehsan Khamespanah[a,b,**], Ramtin Khosravi[a,*], Marjan Sirjani[b,*]

[a]*School of Electrical and Computer Engineering, University of Tehran*
[b]*School of Computer Science, Reykjavik University*

Abstract

NP-hard time complexity of model checking algorithms for TCTL properties in dense time is one of the obstacles against using model checking for the analysis of real-time systems. Alternatively, a polynomial time algorithm is suggested for model checking of discrete time models against $\text{TCTL}_{\leqslant,\geqslant}$ properties (i.e. TCTL properties without $\mathbf{U}^{=c}$ modalities). The algorithm performs model checking against a given formula Φ for a state space with V states and E transitions in $O(V(V+E)\cdot|\Phi|)$. In this work, we improve the model checking algorithm of $\text{TCTL}_{\leqslant,\geqslant}$ properties, obtaining time complexity of $O((V\lg V+E)\cdot|\Phi|)$. We tackle the model checking of discrete timed actors as an application of the proposed algorithms. We show how the result of the fine-grained semantics of discrete timed actors can be model checked efficiently against $\text{TCTL}_{\leqslant,\geqslant}$ properties using the proposed algorithm. This is illustrated using the timed actor modeling language Timed Rebeca. In addition to introducing a new efficient model checking algorithm, we propose a reduction technique which safely eliminates instantaneous transitions of transition systems (i.e. transition with zero time duration). We show that the reduction can be applied on-the-fly during the generation of the original timed transition system without a significant cost. We demonstrate the effectiveness of the reduction technique via a set of case studies selected from various application domains. Besides, while $\text{TCTL}_{\leqslant,\geqslant}$ can be model checked in polynomial time, model checking of TCTL properties with $\mathbf{U}^{=c}$ modalities is an NP-complete problem. Using the proposed reduction technique, we provide an efficient algorithm for model checking of complete TCTL properties over the reduced transition systems.

Keywords: Actor Model; Timed Rebeca; Model Checking; TCTL; Discrete Time Transition System; Durational Transition Graph

1. Introduction

As a basic computational model for modeling of real-time systems, Timed Transition System (TTS) generalizes the basic computational model of transition systems by associating an interval with each transition to indicate how long a transition takes [1]. TTS is expressive enough for modeling the behavior of the majority of real-time distributed systems; however, the formal verification of TTS is PSPACE-complete [1]. Therefore, currently there is no polynomial time algorithm for the verification of TTSs. Another option for analysis of real-time systems is to use Alur and Dill's *Timed Automata* [2]. There exists a large amount of theoretical knowledge and practical experiences about timed automata which all agree on the main drawback of using timed automata being the inefficient analysis techniques which are at least PSPACE-hard [3]. The most widely used model checking toolset for timed automata, UPPAAL, only supports a limited subset of Timed Computation Tree Logic (TCTL) which can be model checked efficiently [4]. The source of this

*Corresponding author
**Principal corresponding author
Email addresses: e.khamespanah@ut.ac.ir (Ehsan Khamespanah), r.khosravi@ut.ac.ir (Ramtin Khosravi), marjan@ru.is (Marjan Sirjani)

inefficiency in the analysis of TTS and timed automata is in how the passage of time is modeled. The model of time in TTS and timed automata is *dense time*, i.e. the passage of time from a state to another state is a nondeterministically chosen real number from an interval.

On the other hand, a wider range of properties can be analyzed for simpler families of timed models in polynomial time. The simplicity of these models lies in the discretization of the passage of time. In these models, the passage of time is modeled by a natural number which is chosen nondeterministically from an interval. The basic approach of such simplifications is proposed in [5, 6] by assuming that each transition takes exactly one time unit. Later, a minor extension has been added to this work by allowing existence of instantaneous transitions (zero time transitions) in [7]. Finally, Timed Transition Graph (TTG) [8] and Durational Transition Graph (DTG) [9] extended the former works by associating discrete time duration with transitions. Although TTG and DTG are less expressive than TTS and timed automata, they can be model checked in polynomial time for a wide range of properties. For example, there is a polynomial time algorithm for model checking of DTGs against $\mathrm{TCTL}_{\leqslant,\geqslant}$ properties (i.e. TCTL properties without any sub-formula of the form $\Phi\ \mathbf{U}^{=c}\ \Psi$). The algorithm performs model checking against formula Φ for a transition system with V states and E transitions in time $O(V \cdot (V+E) \cdot |\Phi|)$ [9]. The details of these model checking algorithms are reviewed in Section 2. Note that, while $\mathrm{TCTL}_{\leqslant,\geqslant}$ can be model checked for DTGs in polynomial time, the model checking against $\mathrm{TCTL}_{=}$ properties (i.e. TCTL properties with sub-formulas of the form $\Phi\ \mathbf{U}^{=c}\ \Psi$) is a NP-hard problem. In this work, we improve the running time of the algorithm of [9] from $O(V \cdot (V+E) \cdot |\Phi|)$ to $O((V \lg V + E) \cdot |\Phi|)$. The newly proposed algorithm is worst-case optimal for model checking of $\mathrm{TCTL}_{\leqslant,\geqslant}$ properties, since its running time is the same as the tight running time of the CTL model checking algorithm [3]. This algorithm is presented in detail in Section 3.

In addition to improving the running time of $\mathrm{TCTL}_{\leqslant,\geqslant}$ model checking algorithm, we propose a reduction technique which safely eliminates instantaneous transitions of timed transition systems. Applying this technique, a new transition system is created, called folded timed transition system (FTS). As discussed in Section 4, in addition to reducing the size of transition systems, the algorithm of *exact path search* in graphs can be used for model checking of FTS against $\mathrm{TCTL}_{=}$ properties. Having instantaneous transitions, the problem of model checking against $\mathrm{TCTL}_{=}$ properties is reducible to the Subset Sum problem which is well-known to be NP-complete [9]. By eliminating instantaneous transitions, FTS can be model checked against TCTL properties efficiently. In the proposed algorithm, for a given TCTL formula, if small values are used as timed quantifiers of TCTL modalities, FTS can be model checked in $O((V \lg V + E) \cdot |\Phi|)$.

We tackle the problem of analyzing discrete timed actors[1] to illustrate the applicability of the proposed approaches. The actor model is a well-established paradigm for modeling the functional behavior of distributed systems with asynchronous message passing. This model was originally introduced by Hewitt [10] and then elaborated by Agha [11, 12] and Talcott [13]. Actors are attracting more and more attention both in academia and industry; whoever, little work has been done on timed actors and even less on analyzing timed actor models. To the best of our knowledge, only a few timed actor modeling languages such as RT-synchronizer [14], real-time Creol [15], and Timed Rebeca [16] exist. Although there are some studies on verification of timed actors [17, 18], the lack of efficient model checking algorithms has limited the use of model checking for this purpose. As DTG is expressive enough to be used as the semantics of discrete timed actors, we show how it can be used for efficient model checking of timed actors. We develop this approach for Timed Rebeca models. Timed Rebeca [19] has been proposed as an extension of the Rebeca language [20, 21] with time constraints and analysis support. Timed Rebeca is an actor-based modeling language which can be used in model-driven methodologies. Using Timed Rebeca a designer builds an abstract model in which each component is a reactive object communicating with other objects through non-blocking asynchronous message passing. In Section 5, we show how the transition systems which are generated based on the fine-grained semantics of Timed Rebeca can be assumed as DTGs to be efficiently model checked against $\mathrm{TCTL}_{\leqslant,\geqslant}$ properties. Although we demonstrate our approach on Timed Rebeca, it can be easily generalized to other timed actor models.

We also elaborate on the execution cost of generating FTSs from DTGs. Using the approach of this paper, the FTS of a Timed Rebeca model is generated without a significant cost, in parallel with the generation

[1]We use "timed actor" and "discrete timed actor" in this paper interchangeably.

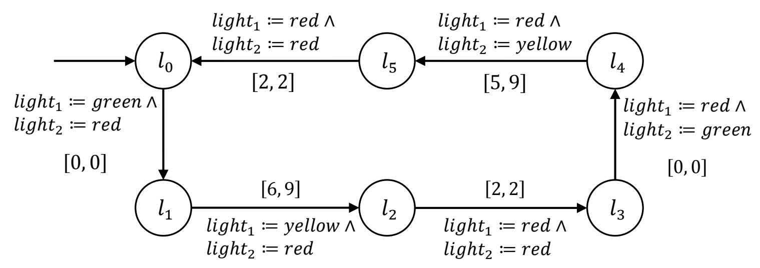

$light_1 \coloneqq red \wedge$
$light_2 \coloneqq red$
[2, 2]
$light_1 \coloneqq red \wedge$
$light_2 \coloneqq yellow$
[5, 9]
l_0
l_5
l_4
$light_1 \coloneqq green \wedge$
$light_2 \coloneqq red$
[0, 0]
$light_1 \coloneqq red \wedge$
$light_2 \coloneqq green$
[0, 0]
l_1
l_2
l_3
[6, 9]
$light_1 \coloneqq yellow \wedge$
$light_2 \coloneqq red$
[2, 2]
$light_1 \coloneqq red \wedge$
$light_2 \coloneqq red$

Figure 1: A TTS model of two traffic lights at a crossroad

of its original transition system and checking for Zeno freedom of models. In Section 4 we show how the algorithm of transition system generation and checking for Zeno behavior are modified to generate FTSs. We have developed a tool (added to the rich Rebeca toolset [22]), to illustrate the impact of using these techniques by applying them on a set of case studies in different application domains (Section 6).

This paper is an extended version of the workshop paper [23]. In [23], we showed how the TCTL model checking algorithm of [9] can be used for the model checking of Timed Rebeca models. We also introduced FTS in that paper. Apart from adding more detail about the proposed approaches, this paper extends [23] as follows:

- We propose a new model checking algorithm with an asymptotically smaller running time in comparison with the existing model checking algorithms of discrete time systems (Sections 3.1 and 3.2).
- We determine a condition where the newly proposed algorithm is optimal for discrete time systems (Theorem 3 and its corresponding discussion).
- We improve the execution time of the reduction technique by combining transition system generation, checking for the Zeno freedom, and applying the reduction technique (Section 4.2).
- The experimental results are improved for better illustration of the effectiveness of this work (Section 6).

2. Timed Model Checking of Discrete Time Systems against TCTL properties

Timed transition system (TTS), as a basic computational model of real-time systems, generalizes the basic computation model of transition systems by associating an interval with each transition to indicate how long a transition takes [1]. Figure 1 illustrates how the behavior of a real-time system is modeled by TTS. The example models the behavior of a controller of two traffic lights at a crossroad. Initially, the controller is in state l_0. It immediately makes a transition to l_1 as the duration of its only outgoing transition is $[0, 0]$. The controller stays in l_1 for a duration of $[6, 9]$ units of time. It means that for a nondeterministically chosen real number from the interval $[6, 9]$, light$_1$ remains green. Then, the state changes to l_2 and for two units of time light$_1$ is yellow. Then, both lights are set to red and immediately light$_2$ changes to green, and so on. In this example, the dense time model is used to show the passage of time.

Definition 1 (Timed Transition System (TTS)). *A timed transition system is defined as a tuple $TTS = (S, s_0, \rightarrow, Act, AP, L, T)$, where S is the set of states, s_0 is the initial state, $\rightarrow \subseteq S \times Act \times S$ is the transition relation, and Act is the set of possible actions. Here, for a given set of atomic propositions AP, the labeling function $L : S \rightarrow 2^{AP}$ relates a set of atomic propositions its given state. Finally, $T : S \times Act \times S \rightarrow \mathbb{N} \times \mathbb{N}$ associates an interval with each transition.* □

As discussed in [1], TTS is expressive enough for modeling the behavior of the majority of real-time systems. However, the verification algorithms of TTSs are PSPACE-complete. In practice, it is hard to use TTS for the efficient analysis of real world systems. The same holds for the verification of real-time systems with dense time presented in other semantics (region transition system, etc.) [3].

In contrast, there are many timed models for modeling of *discrete time* systems which can be verified efficiently in polynomial time. Discrete time is the time model in which passage of time is modeled by natural numbers. A Durational Transition Graph (DTG), as one of these models, is a TTS where the duration intervals of transitions are interpreted in the domain of natural numbers [9]. This way, a transition with a bounded duration interval $[a, b]$ between two states s and s' can be assumed as $b - a + 1$ different nondeterministic transitions from s to s' with different duration values of $a, a+1, \cdots, b$.

Definition 2 (Durational Transition Graph). *A durational transition graph is a tuple* $DTG = (S, s_0, \rightarrow, AP, L)$ *where* S *is the set of states,* s_0 *is the initial state,* $\rightarrow \subseteq S \times \Upsilon \times S$ *is the transition relation,* AP *is the set of atomic propositions, and* $L : S \rightarrow 2^{AP}$ *is a labeling function.*
Here, Υ *is the set of all the possible finite* ($\upsilon \in \Upsilon \wedge \upsilon = [n, m] \cdot n, m \in \mathbb{N}$) *or right-open infinite* ($\upsilon \in \Upsilon \wedge \upsilon = [n, \infty) \cdot n \in \mathbb{N}$) *intervals.* □

DTGs can be model checked against Timed CTL (TCTL) properties [3] efficiently. TCTL is a real-time variant of CTL aimed to express properties of timed systems. TCTL is used for model checking of both discrete time and dense time systems. In TCTL, the until modality is equipped with a time constraint such that the TCTL formula $\Phi\, \mathrm{U}^{\rho}\, \Psi$ holds for the state s if and only if Ψ holds in the state s' while Φ holds in all states from s to s' and the time difference between s and s' satisfies condition ρ. The syntax of TCTL is formally described in the following definition.

Definition 3 (Syntax of TCTL). *Any TCTL formula is formed according to the following grammar:*

$$\Phi ::= p \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid E\,\varphi \mid A\,\varphi$$

where p *is an atomic proposition and* φ *is a path formula. A path formula in TCTL is formed according the following grammar:*

$$\varphi ::= \Phi_1\, \mathrm{U}^{\sim c}\, \Phi_2$$

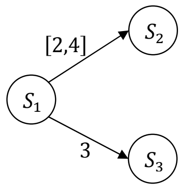
where c *is a natural number and* $\sim \in \{<, \leqslant, =, \geqslant, >\}$. *In addition to the until modality, the globally and finally path modalities can be equipped with time constants. As in CTL, these modalities can be constructed using the until modality [9], and can be safely omitted from the syntax and semantics of (T)CTL. However, in this paper, we use these modalities to make formulas easier to read and understand. Also, note that to address the size of a given formula* Φ*, which is the number of modalities in* Φ*, is shown by* $|\Phi|$. □

In the following, we present the semantics of TCTL properties over DTGs based on the work of [9]. The clauses of Definition 5 show the conditions when a given TCTL formula Φ holds for state $s \in S$ of $DTG = (S, s_0, \rightarrow, AP, L)$. Here, we assume that $Paths(DTG, s)$ represents the set of all valid timed paths of DTG starting from $s \in S$ in the form of $s \xrightarrow{d_0} s_1 \xrightarrow{d_1} \cdots$, as described below.

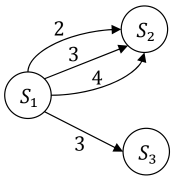
Definition 4 (The Set of Timed Paths). *In a given durational transition system* $DTG = (S, s_0, \rightarrow, AP, L)$*, a sequence* $\pi = (s_0, d_0), (s_1, d_1), \cdots$ *where* $s_i \in S$ *and* $d_i \in \Upsilon$ *is a valid timed path if and only if for any pair of* (s_i, d_i) *there is* $(s_i, \upsilon, s_{i+1}) \in \rightarrow$ *and* $d_i \in \upsilon$. *The set of* $Paths(DTG, s)$ *is defined as the set of all valid timed paths of* DTG *which are started from the state* s. □

Definition 5 (Semantics of TCTL). *A given TCTL formula* Φ *holds for state* s *of* $DTG = (S, s_0, \rightarrow, AP, L)$ *as described by the following items.*

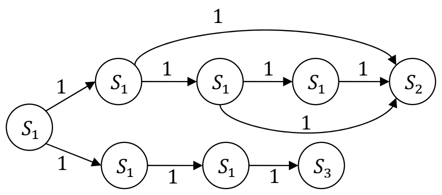
- $s \models p$ if and only if $p \in L(s)$
- $s \models \neg\Phi$ if and only if $s \not\models \Phi$
- $s \models \Phi_1 \wedge \Phi_2$ if and only if $s \models \Phi_1$ *and* $s \models \Phi_2$)
- $s \models \mathbf{E}\,\Phi_1 \mathbf{U}^{\sim \mathbf{c}} \Phi_2$ if and only if $\exists\, \pi \in Paths(DTG, s) \wedge \exists\, n \geqslant 0 \cdot (s_n \models \Phi_2) \wedge (\sum_{i \in [0,n)} d_i$ *satisfies condition* $\sim c) \wedge (\forall\, 0 \leqslant j < n \,\cdot\, s_j \models \Phi_1)$



(a) A DTG with three states



(b) TTS of the DTG assuming the jump semantics



(c) TTS of the DTG assuming the continuous semantics

Figure 2: An intuitive representation of the TTSs with respect to the jump and continuous semantics (it shows the continuous semantics as nondeterminism is resolved immediately in the first S_1) [9].

- $s \models \mathbf{A}\,\Phi_1\mathbf{U}^{\sim \mathbf{c}}\Phi_2$ if and only if $\forall\,\pi \in \mathit{Paths}(DTG, s) \wedge \exists\, n \geqslant 0 \cdot (s_n \models \Phi_2) \wedge (\sum_{i\in[0,n)} d_i$ *satisfies condition* $\sim c) \wedge (\forall\, 0 \leqslant j < n \;\cdot\; s_j \models \Phi_1)$

□

Using the above semantics for model checking DTGs against TCTL formulas requires resolving the nondeterminism of durations of transitions. The meaning of a duration on a transition between two states can be interpreted in two different ways, called *jump semantics* and *continuous semantics* [9]. As a result, two different TTSs are generated for any DTG. In these two TTSs, instead of an interval, only one natural number is associated with each transition as its duration. For a given transition $(s, [n, m], s')$ the mentioned semantics are interpreted as follows.

Jump Semantics. In this semantics, moving from the state s to the state s' takes an integer time $d \in [n, m]$. Here, before starting transition from s to s', the value of d is determined, and then the system waits for d units of time and it reaches state s' at time $t + d$. Figure 2(b) shows how this semantics works for the DTG of Figure 2(a). The idea of this semantics is the same as the semantics of Timed Transition Graph [8] and the semantics of Timed Rebeca as it will be described in Section 5.

Continuous Semantics. In contrast to the jump semantics, in the case of a transition from the state s to the state s' with a duration $d \in [n, m]$, the waiting time is not specified at the start time of the transition. Using the continuous semantics, the system first waits for n units of time in state s, then, at each point in time interval $[0, m - n]$ it can leave s and go to s'. Figure 2(c) shows how this semantics works for the DTG of Figure 2(a). The idea of this semantics is the same as the semantics of timed automata of Alur [2].

For a given DTG, two TTSs generated based on jump semantics and continuous semantics are not bisimilar. This can be shown by a TCTL formula which is satisfied by one of them but is violated by the other one. For example, in the DTG of Figure 2(a), TCTL property $\mathbf{A}(\mathbf{EF}(s_3)\ \mathbf{U}^{\leqslant 5}\ (s_3 \vee s_2))$ is satisfied in the TTS of its jump semantics. As shown in Figure 2(b) state s_1 satisfies $EF(s_3)$, and after leaving s_1, formula $s_2 \vee s_3$ is satisfied in less than 5 units of time. In contrast, as shown in Figure 2(c), after passage of time by one unit, the second s_1 in the path to s_2 does not satisfy neither $EF(s_3)$ nor $s_2 \vee s_3$. Therefore, the property is violated in this case.

Using either jump or continuous semantics, there are polynomial time model checking algorithms for $\text{TCTL}_{\leqslant,\geqslant}$ properties; however, model checking of $\text{TCTL}_{=}$, TLTL, and TCTL* properties remains PSPACE-complete. In the following, we review the model checking algorithm of DTGs in jump semantics against $\text{TCTL}_{\leqslant,\geqslant}$ properties according to [9]. As in model checking of CTL properties, here, we show how the satisfaction set $Sat(\Phi)$ is computed for a given formula Φ. The running time of the algorithm to find $Sat(\cdot)$ for a DTG with the jump semantics is $O(V \cdot (E+V) \cdot |\Phi|)$ where V is the number of states, E is the number of transitions, and $|\Phi|$ is the size of formula Φ.

Let $DTG_{\mathcal{M}} = (S, s_0, \rightarrow, AP, L)$ be a DTG of a given model $\mathcal{M}$. The extended version of the standard CTL model checking algorithm is used to support $\mathbf{EU^{\sim c}}$ and $\mathbf{AU^{\sim c}}$ sub-formulas. The cases for p, $\neg\Phi$, and $\Phi_1 \wedge \Phi_2$ are the same as their counterparts in CTL. The following four cases show how the extension works for timed sub-formulas of types $\mathbf{E}(\Phi\mathbf{U^{\sim c}}\Psi)$ and $\mathbf{A}(\Phi\mathbf{U^{\sim c}}\Psi)$.

$Sat(\mathbf{E}(\Phi\,\mathbf{U^{\leqslant c}}\Psi))$: Assume that $DTG^{sub}_{\mathcal{M}}$ is the induced subgraph of $DTG_{\mathcal{M}}$ over $S' = Sat(\mathbf{E}(\Phi\mathbf{U}\Psi))$, including only the states satisfying $\mathbf{E}(\Phi\mathbf{U}\Psi)$. This can be done using standard CTL model checking in $O(V+E)$. In addition, the weight of each transition of $DTG^{sub}_{\mathcal{M}}$ is set to the lower bound of its corresponding duration interval in $DTG_{\mathcal{M}}$. This way, state $s \in S'$ is in $Sat(\mathbf{E}(\Phi\mathbf{U^{\leqslant c}}\Psi))$ if and only if running a *single source shortest path* algorithm from state $s \in S'$ results in finding a path from s to s' where $s' \models \Psi$ and the weight of the path is not bigger than c. So, one round of the algorithm (with the running time of $O(V+E)$) is needed for each state of S'. As a result, the total running time of this algorithm is $O((V+E) + V \cdot (V+E)) = O(V \cdot (V+E))$.

$Sat(\mathbf{E}(\Phi\,\mathbf{U^{\geqslant c}}\Psi))$: Assume that a new atomic proposition $P_{SCC^{+}(\Phi)}$ is defined. Each state s is labeled by $P_{SCC^{+}(\Phi)}$ iff s is a member of a strongly connected component (SCC), in which all of the states satisfy Φ and at least one of the transitions inside the SCC results in non-zero progress in time. Labeling S' with $P_{SCC^{+}(\Phi)}$ can be done using an extension of Tarjan's algorithm [24] for detecting SCCs in $O(V+E)$, resulting in $DTG'_{\mathcal{M}}$.

The induced subgraph $DTG^{sub}_{\mathcal{M}}$ of $DTG'_{\mathcal{M}}$ is defined over $S' = Sat(\mathbf{E}(\Phi\mathbf{U}\Psi))$, including only the states satisfying $\mathbf{E}(\Phi\mathbf{U}\Psi)$. This way, $s \in S'$ is in $Sat(\mathbf{E}(\Phi\mathbf{U^{\geqslant c}}\Psi))$ if and only if one of the following conditions holds.

- There is an acyclic path from s to a state satisfying Ψ and the overall weight of the path between them is not less than c.
- State s satisfies CTL formula $\mathbf{E}(\Phi\,\mathbf{U}(P_{SCC^{+}(\Phi)} \wedge \mathbf{E}(\Phi\,\mathbf{U}\,\Psi)))$. Satisfying this formula, there is a path from s to a state which satisfies Ψ through some state s' where $s' \models P_{SCC^{+}(\Phi)}$. This way, by cycling in the SCC containing s', the elapsed time can be increased to more than any constant value c.

For each state, checking for both conditions requires a search algorithm in $O(V+E)$. As a result, the total running time of this algorithm is $O((V+E) + V \cdot (V+E)) = O(V \cdot (V+E))$.

$Sat(\mathbf{A}(\Phi\,\mathbf{U^{\leqslant c}}\Psi))$: using the equivalence relations $\mathbf{A}(\Phi\,\mathbf{U^{\leqslant c}}\Psi) \equiv \mathbf{AF^{\leqslant c}}\,\Psi \wedge \neg\mathbf{E}((\neg\Psi)\mathbf{U}(\neg\Phi \wedge \neg\Psi))$ and $\mathbf{AF^{\leqslant c}}\,\boldsymbol{\Psi} \equiv \neg\mathbf{E}(\neg\Psi\,\mathbf{U}^{>c}\,\top) \wedge \neg\mathbf{E}(\neg\Psi\,\mathbf{U}\,P_{SCC^{0}(\neg\Psi)})$, this case is reduced to a combination of the previous cases. A given state s satisfies proposition $P_{SCC^{0}(\neg\Psi)}$ if and only if s is in a SCC in which all of the states satisfy $\neg\Psi$, and zero is associated with all transitions of the SCC as the progress of time. Using an extension of Tarjan's algorithm, states with $P_{SCC^{0}(\neg\Psi)}$ are determined in $O(V+E)$; so, the total running time of this algorithm is $O((V+E) + V \cdot (V+E)) = O(V \cdot (V+E))$.

$Sat(\mathbf{A}(\Phi\,\mathbf{U^{\geqslant c}}\Psi))$: using the equivalence relation $\mathbf{A}(\Phi\,\mathbf{U^{\geqslant c}}\Psi) \equiv \mathbf{AG}^{<c}\,(\Phi) \wedge \mathbf{A}(\Phi\,\mathbf{U}^{>0}\Psi)$ and $\mathbf{AG}^{<c}\,\Phi \equiv \neg\mathbf{EF}^{<c}\,\neg\Phi$, this case is reduced to a combination of the previous cases. So, the total running time of this algorithm is $O(V \cdot (V+E))$.

As model checking of DTGs in the continuous semantics is out of the scope of this work, it is not described here.

3. Improving the TCTL$_{\leqslant,\geqslant}$ Model Checking Algorithm

In the previous section, we illustrated how DTGs can be model checked against TCTL$_{\leqslant,\geqslant}$ properties with running time $O(V \cdot (V + E) \cdot |\Phi|)$. In this section, we show how the two phases of the TCTL$_{\leqslant,\geqslant}$ model checking algorithm are combined to develop a new TCTL$_{\leqslant,\geqslant}$ model checking algorithm with running time $O(V \lg V + E)$. Here, we show how the algorithm works for calculating $Sat(.)$ for two primitive cases $\mathbf{E}(\Phi\, \mathbf{U}^{\leqslant c}\, \Psi)$ and $\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi)$. As shown in the previous section, other TCTL formulas can be constructed using $\mathbf{EU}^{\leqslant c}$, $\mathbf{EU}^{\geqslant c}$, and other untimed CTL operators and modalities with the maximum overhead of $O(V + E)$. Therefore, the overall cost of the preparation and the model checking is $O(V \lg V + E)$ for all cases.

Before describing the new algorithm, we review how the CTL model checking algorithm calculates the value of $Sat(\mathbf{E}(\Phi\, \mathbf{U}\, \Psi))$. One of the implementations of this algorithm is an iterative algorithm, called enumerative backward search [3]. As shown in Algorithm 1, in the initial step, T is defined to be the set of states satisfying Ψ (line 2). Based on the semantics of the until modality, these states satisfy $\mathbf{E}(\Phi\, \mathbf{U}\, \Psi)$. Then, iteratively, other states are added to T. In each iteration, a state $s \in S\backslash T$ is added to T if and only if $s \models \Phi$ and at least one of the successors of s is in T (lines 4 to 8). Note that in this section, we assume that for a given formula $\mathbf{E}(\Phi\mathbf{U}^{\leqslant c}\Psi)$ or $\mathbf{E}(\Phi\mathbf{U}^{\geqslant c}\Psi)$ the values of $Sat(\Phi)$ and $Sat(\Psi)$ are computed in advance.

Algorithm 1: Enumerative backward search for computing $Sat(\mathbf{E}(\Phi\, \mathbf{U}\, \Psi))$ [3]

```
Input: Finite transition system TS with set of states $S$ and CTL formula $\mathbf{E}(\Phi\,\mathbf{U}\,\Psi)$
Output: Set of $Sat(\mathbf{E}(\Phi\,\mathbf{U}\,\Psi)) = \{s \in S \mid s \models \mathbf{E}(\Phi\,\mathbf{U}\,\Psi)\}$
1  begin
2      $T \leftarrow Sat(\Psi)$
3      $Q \leftarrow T$
4      foreach state $s \in Q$ do
5          $Q \leftarrow Q\backslash\{s\}$
6          foreach state $s' \in$ PREDECESSORS($s$) do
7              if $s' \notin T \wedge s' \models \Phi$ then
8                  $Q \leftarrow Q \cup \{s'\}$
9                  $T \leftarrow T \cup \{s'\}$
10     return $T$
```

As described in the following sections, some modifications are applied to this algorithm to support the timed until modality. The major modification of the algorithm is in the state selection policy (in line 4 of the algorithm). Two different policies are required for the timed modalities $\mathbf{EU}^{\leqslant c}$ and $\mathbf{EU}^{\geqslant c}$.

3.1. Calculating $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\leqslant c}\, \Psi))$

The main idea of the new model checking algorithm is in performing reversed Dijkstra single source shortest path (SSSP) instead of using classic Dijkstra SSSP. The extension of reversed Dijkstra SSSP used here traverses a given state space from the goal states (which are states of $Sat(\Psi)$) to their ancestors. This way, as both finding states satisfying $\mathbf{E}(\Phi\, \mathbf{U}\, \Psi)$ and checking time constraint are started from the goal states, they can be combined together. The details of the new algorithm for calculating $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\leqslant c}\, \Psi))$ are depicted in Algorithm 2. In the new algorithm, Q is defined as a Fibonacci max-heap which stores pairs of $(key, value)$ where key is an integer number and $value$ is a state. The value of a key in Q is interpreted as *the minimum distance to one of the states which satisfies* Ψ (denoted by δ) of its paired state. Four functions EMPTY_HEAP, PUT, EXTRACT_MIN, and DECREASE_KEY are used for creating an empty Fibonacci max-heap, putting a pair $(key, state)$ in a heap, extracting the pair with the minimum key, and decreasing the key of a given state, respectively. In addition, the function $time : S \times S \rightarrow \mathbb{N}$ is defined to retrieve the amount of progress of time in the transition between two given states.

As shown in Algorithm 2, the initialization part of the algorithm is in lines 2 to 10. During the initialization, all of the states of $Sat(\Psi)$ are added to T (the return value of the algorithm) as they satisfy $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{0}\,\Psi))$. The other states of S are added to Fibonacci max-heap Q. The key of a given state $s \in S \backslash T$ is set to infinity except in case a state $s' \in T$ is an immediate successor of s. For such a state the key is set to $time(s, s')$. If s has transitions to more than one state in T, the key is the minimum time value of those transitions. The initialization running time is $O(V + E)$ as the vertices and edges are visited once.

Algorithm 2: Enumerative backward search for calculating $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi))$

```
Input: Finite transition system TS with set of states $S$ and TCTL$_{\leqslant,\geqslant}$ formula $\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi)$
Output: $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi)) = \{s \in S \mid s \models \mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi)\}$
1  begin
2      $T \leftarrow$ Sat($\Psi$)
3      $Q \leftarrow$ EMPTY_HEAP
4      foreach state $s \in S \backslash T$ do
5          if $s \models \Phi$ then
6              $\delta_s \leftarrow \infty$
7              foreach state $s' \in$ SUCCESSORS($s$) do
8                  if $s' \in T$ then
9                      $\delta_s \leftarrow \min\{\delta_s,$ time($s$,$s'$)$\}$
10             PUT($Q$, $\delta_s$, $s$)
11     while $Q \neq \emptyset$ do
12         $(\delta_s, s) \leftarrow$ EXTRACT_MIN($Q$)
13         if $\delta_s > c$ then
14             break
15         $T \leftarrow T \cup \{s\}$
16         foreach state $s' \in$ PREDECESSORS($s$) do
17             if $s' \notin T \wedge s' \models \Phi$ then
18                 $\delta_{s'} \leftarrow \delta_s +$ time($s'$,$s$)
19                 DECREASE_KEY($Q$, $s'$, $\delta_{s'}$)
20     return $T$
```

In addition to some changes in the initialization part, some modifications to the main part of the CTL model checking algorithm are required. The main part of the new algorithm is in lines 11 to 19. One of the differences between the main part of the new algorithm and the main part of the algorithm of CTL model checking in Algorithm 1 is in the termination condition of line 13. The termination condition is required in the new algorithm as the backward search must stop when δ is bigger than c. The other difference is in updating δ of states in lines 18 and 19. Intuitively, when a new state s is added to T, maybe δ of the predecessors of s is changed as there is a new path via s to the states which satisfy Ψ. Therefore, δ of PREDECESSORS(s) is decreased in lines 18 and 19. Note that if the newly found value is bigger than the previous value, DECREASE_KEY does nothing. The new algorithm requires $O(V)$ number of extractions from the Fibonacci max-heap Q and $O(E)$ number of decreasing keys (in the worst case, extracting a state results in decreasing the keys of all of its predecessors). In a Fibonacci max-heap of size n, the amortized running time of extracting an element is $O(\lg n)$ and decreasing a key is $O(1)$. Hence, the running time of the main part of the algorithm is $O(V \lg V + E)$. As a result, the total running time of the new algorithm is $O(V \lg V + E)$.

Theorem 1. *The aforementioned algorithm computes the set of states which satisfy a given TCTL$_{\leqslant}$ property* $\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi)$.

Proof. Assume that there is a state $s \in S$ which satisfies the TCTL$_{\leqslant}$ formula $\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi)$. As s satisfies $\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi)$; there is a state $s' \in S$ such that s' satisfies Ψ, there is a path between s and s' where the length of the path is less than c, and all of the states between s and s' satisfy Φ. Using the new algorithm, reversed Dijkstra starts from s' as it satisfies Ψ (lines 2 to 10). Using reversed Dijkstra (ignoring the modifications which are made to support property satisfaction in lines 13 and 17), starting from s', the algorithm visits s and associates a value which is less than c with s (as there is a path between s and s' with the length of less than c). Reversed Dijkstra is not terminated before reaching s because of the conditional statement of line 13 as the length of the path is less than c. Also, as all of the states between s and s' satisfy Φ, the algorithm does not miss the states of the path between s and s' because of the conditional statement of line 17. Therefore, $s \in Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi))$ which is computed by the new algorithm. The same argument is valid for proving that if the new algorithm puts a state s in $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi))$, the state s satisfies the formula $\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi)$. □

3.2. Calculating $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geqslant c}\,\Psi))$

As described in Section 2, the algorithm to find $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geqslant c}\,\Psi))$ is reduced to two cases. A given state $s \in S$ is in $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geqslant c}\,\Psi))$ if and only if there exists a simple path from s to one of the states of $Sat(\Psi)$ and the duration of the path is at least c, or there exists a path with at least one non-zero cycle from s to one of the states of $Sat(\Psi)$ (the elapse of time can be increased to more than c by traversing the cycle). Note that all the states on the mentioned paths satisfy Φ.

The new approach to calculate $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geqslant c}\,\Psi))$ is like the approach of calculating $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leqslant c}\,\Psi))$. In this case, enumerative backward search starts from a state which has the maximum value for *now*, called *oldest state*. This state is selected because of the fact that the simple path from one of the ancestors of the oldest state which has the maximum elapse of time ends in an oldest state. Therefore, selecting an oldest state in each iteration of the backward search, guarantees that the longest simple path in the state space is seen before processing of its predecessor states. This way, the states conforming to the first case are calculated. To handle the second case, during the backward search, if the search reaches a state which is marked by the label $P_{SCC^{+}(\Phi)}$, the state is put in $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geqslant c}\,\Psi))$.

For the efficient implementation of this algorithm, we define Q as an ordinary max-heap. Three functions `EMPTY_HEAP`, `PUT`, and `EXTRACT_MAX` are used for creating an empty heap, putting a pair $(key, value)$ in a heap, and extracting the pair with the maximum key, respectively. The function $now : S \to \mathbb{N}$ is defined to retrieve the times of states. We also assume that each state has an additional field which shows the maximum distance from this state to one of the states which satisfy Ψ (denoted by Δ). The details of the new algorithm are depicted in Algorithm 3.

The initialization part of Algorithm 3 is in lines 2 to 7. During the initialization, Δ of all the states are set to zero and any state $s \in Sat(\Psi)$ is added to Q in the form of a pair $(now(s), s)$. As none of the states in this step satisfies the timing constraint of the formula, T has no member and it is set to the empty set. The initialization part running time is $O(V \lg V)$ as all of the vertices must be visited once and in the worst case Q is built by calling `PUT` for V times.

The main part of the algorithm is in lines 8 to 19. One of the differences between the main part of this algorithm and the standard CTL algorithm's main part (Algorithm 1) is in the policy of adding elements to T. Here, instead of adding s' to T immediately after extracting it, s' is added to T when it satisfies a timing constraint, as shown in line 19. The other difference is in lines 12 to 16 where Δ of states are updated. Normally, Δ of a state s is set based on the value of Δ of its successors. But, in case s is a member of SCC, there is the possibility of increasing Δ to an arbitrarily large value by cycling from s to itself. So, Δ of s is set to infinity to address this fact. The new algorithm requires $O(V)$ number of extractions from heap Q and $O(E)$ number of processing the predecessors of states (i.e. the maximum number of edges). As the running time of extracting from a heap of n elements is $O(\lg n)$, the running time of the main part of the algorithm is $O(V \lg V + (V + E)) = O(V \lg V + E)$. As a result, the total running time of the algorithm is $O(V \lg V + E)$.

Theorem 2. *The aforementioned algorithm computes the set of states which satisfy a given TCTL$_{\geqslant}$ property* $\mathbf{E}(\Phi\,\mathbf{U}^{\geqslant c}\,\Psi)$.

Proof. As this algorithm finds $Sat(.)$ in two different cases, we split the proof into the following two cases.

1. Assume that $s \in S$ satisfies $\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi)$ and $s' \in S$ is a state where s' satisfies Ψ and there is a path between s and s' such that all of the states in the path satisfy Φ. Also, assume that there is a state $s'' \in S$ in the path between s and s' such that the label $P_{SCC^+(\Phi)}$ is associated with s''. In this case, as the algorithm is developed based on Algorithm 1, all of the states in the path between s and s' are explored as they satisfy $\mathbf{E}(\Phi\, \mathbf{U}\, \Psi)$. During this exploration, upon visiting s'' the value of Δ is set to the infinity, and it is added to $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi))$. The same procedure happens for all of the ancestors of s'' too, because of the statement of line 16. Therefore, all of the ancestors of s'' are put in $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi))$, including s.
2. Assume that $s \in S$ satisfies $\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi)$ and $s' \in S$ is a state where s' satisfies Ψ and there is a path between s and s' such that all of the states in the path satisfy Φ. Also, assume that this path is the longest acyclic path between s and other states which satisfy Ψ. In this case, the algorithm starts exploring the state s' before the other states which satisfy Ψ and are reachable from s. It is because of the fact that $now(s')$ is bigger than the value of $now(.)$ of them, results in extracting s' from Q sooner than the others. So, the value of Δ of the ancestors of s' in the path from s is updated based on the distance between the ancestors and s'. As a result, reaching s results in setting the value of Δ to the length of the maximum acyclic path between s and s' and adding s to $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi))$.

The same argument is valid for proving that if the new algorithm puts a state s in $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi))$, the state s satisfies the formula $\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi)$. □

Algorithm 3: Enumerative backward search for computing $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi))$

```
Input: Finite transition system TS with set of states $S$, TCTL$_{\leqslant,\geqslant}$ formula $\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi)$, and the set
       of states $SCC$ as the states in cycles of which all members are in $Sat(\Phi)$
Output: $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi)) = \{s \in S \mid s \models \mathbf{E}(\Phi\, \mathbf{U}^{\geqslant c}\, \Psi)\}$
1  begin
2      $T \leftarrow \varnothing$
3      $Q \leftarrow$ EMPTY_HEAP()
4      foreach state $s \in S$ do
5          $\Delta_s \leftarrow 0$
6          if $s \models \Psi$ then
7              PUT($Q$, now($s$), $s$)
8      while $Q \neq \varnothing$ do
9          $(now_s, s) \leftarrow$ EXTRACT_MAX($Q$)
10         foreach state $s' \in$ PREDECESSORS($s$) do
11             if $s' \notin T \wedge s' \models \Phi$ then
12                 if $s' \in SCC$ then
13                     $\Delta \leftarrow \infty$
14                 else
15                     $\Delta \leftarrow \Delta_s +$ time($s'$,$s$)
16                 $\Delta_{s'} \leftarrow \max\{\Delta, \Delta_{s'}\}$
17                 PUT($Q$, now($s'$), $s'$)
18                 if $\Delta_{s'} \geqslant c$ then
19                     $T \leftarrow T \cup \{s'\}$
20     return $T$
```

Combining the above two algorithms, we have the following result.

Theorem 3. *There is an $O((V \lg V + E) \cdot |\Phi|)$ algorithm for model checking of DTGs with V states and E transitions against a $TCTL_{\leqslant,\geqslant}$ property Φ.* $\square$

Note that for the dense transition systems where the number of transitions is asymptotically larger than $V \lg V$ (i.e., $E = \Omega(V \lg V)$), this algorithm is the most efficient algorithm for model checking against $\text{TCTL}_{\leqslant,\geqslant}$ properties. This is because the running time of the algorithm is $O(E \cdot |\Phi|)$, which is the same as the running time of the optimal CTL model checking algorithm [3].

Corollary 1. *The proposed $TCTL_{\leqslant,\geqslant}$ model checking algorithm is the asymptotically optimal algorithm for dense transition systems.*

4. A Reduction Technique Based on Folding Instantaneous Transitions

In this section, we propose a reduction technique, called "Folding Instantaneous Transitions", to make the model checking of object based models against $\text{TCTL}_{\leqslant,\geqslant}$ cheaper. Using this reduction technique, in Section 4.2, we will propose an approach for the model checking of $\text{TCTL}_{=}$ properties in polynomial time. This approach is developed based on the fact that after folding instantaneous transitions, there is no transition with zero time in the transition system. So, an efficient algorithm can be used for the model checking of $\text{TCTL}_{=}$ properties.

The idea of folding instantaneous transitions is developed based on the fact that the instantaneous transitions take no time to execute; so, the system cannot "stay" in the states whose outgoing transitions are all instantaneous. Hence, these states are not observable to the verifier (as an external observer). As generally assumed in modeling timed systems, instantaneous transitions take priority over non-instantaneous ones. So, any state which has an instantaneous outgoing transition cannot have non-instantaneous transitions. Hence, there are two types of states: the ones whose outgoing transitions are all instantaneous (called *transient states*), and the ones which have no outgoing instantaneous transition (called *progress-of-time states* as in Section 5.2). Note that the result of folding instantaneous transitions is not bisimilar with its original transition systems; so, it does not preserve the result of model checking against all properties. This is because of the fact that transient states are eliminated from the state spaces. However, atomic propositions may change in those states. But, in the object-oriented paradigm, transient states of objects are not observable, and upon completion of an action, the states of objects are examined. This fact is true for Timed Rebeca models. So, although folding instantaneous transitions eliminates some transient states, it can be used for the analysis of Timed Rebeca models which considers the observable behavior of actors.

4.1. Folding Instantaneous Transitions

Folding instantaneous transitions is a reduction technique that eliminates all instantaneous transitions as well as all transient states from the DTGs. There is a transition between two states of an FTS if and only if the two states are consecutive progress-of-time states in the original DTG. Figure 3 illustrates how a DTG (at the left side) is transformed to its corresponding FTS (at the right side). In the figure, the dotted states are the initial states and the states with thick borders are the progress-of-time states.

To present the formal definition of FTS, at the first step, we need to define $npts : S \to 2^S$ which finds the set of the nearest progress-of-time states from a given state. For a given state $s \in S$, all states in $npts(s)$ are progress-of-time states and there is no progress-of-time state in the paths from s to the states of $npts(s)$.

Definition 6 (Nearest Progress-of-Time States)**.** *For a given $DTG_{\mathcal{M}} = (S, s_0, \to, AP, L)$ and two states $s, s' \in S$, s' is in $npts(s)$ if and only if s' be a progress-of-time state and for all of valid paths between s and s' such as $\pi = (s, d), (s_1, d_1), (s_2, d_2), \cdots, (s_n, d_n), (s', d')$, none of $s_1, s_2, \cdots, s_n$ are progress-of-time states.* $\square$

Using the definition of the nearest progress-of-time state, the definition of FTS is straightforward as below.

Definition 7 (Folded Transition System)**.** *For a given $DTG_{\mathcal{M}} = (S, s_0, \to, AP, L)$, its corresponding folded transition system is defined as the tuple $FTS(DTG_{\mathcal{M}}) = (S', s_0, \hookrightarrow, AP, L)$, where:*

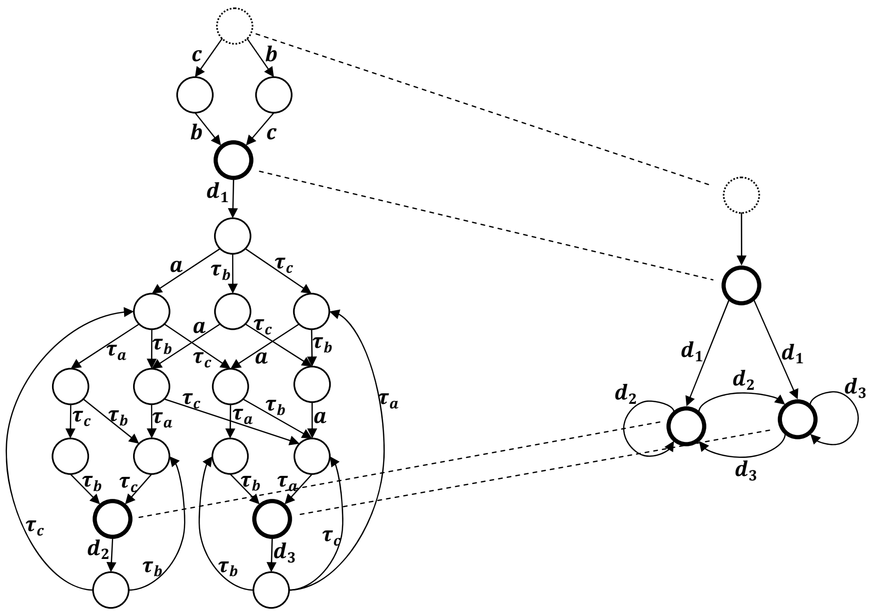


Figure 3: Example of how folding instantaneous transitions reduction works

- *$S' \subseteq S$ which contains all progress-of-time states, and the initial state.*
- *For all $s'_1, s'_2 \in S'$, there exists $(s'_1, d, s'_2) \in \hookrightarrow$ if and only if $s'_2 \in npts(s'_1)$. The value of d is the value of the time elapse associated with the outgoing transition of s'_1 (which is a progress-of-time transition). For the initial state, d is set to zero.*

□

As the states and transitions of a FTS can be assumed as the subset of its corresponding DTG, a FTS can be model checked against $\text{TCTL}_{\leqslant,\geqslant}$ properties using the previously described algorithm.

Corollary 2. *The FTS of a given DTG can be model checked against $\text{TCTL}_{\leqslant,\geqslant}$ property Φ in $O((V \lg V + E) \cdot |\Phi|)$.*

4.2. Complete TCTL Model Checking of DTGs

The model checking algorithms presented so far work for FTSs of DTGs with running time $O((V \lg V + E) \cdot |\Phi|)$. Here, we show that the approach of [25] can be used for efficient model checking of $\text{TCTL}_{=}$ properties respect to FTSs in pseudo-polynomial time. Then, we discuss that for a wide range of complete TCTL properties, the running time of model checking algorithm for a DTG is reduced to $O((V \lg V + E) \cdot |\Phi|)$ for TCTL property Φ.

As known in graph theory, the problem of finding a path between two vertices in a weighted graph of which the weight equals to a given number (called finding the exact path length (EPL) problem), is an NP-complete problem (using a reduction from finding the EPL between two states to the subset-sum problem [26]); so, there is no known polynomial time algorithm for solving the EPL problem. In the same way, the authors in [9] showed that the problem of model checking for the exact time condition is an NP-complete problem. Therefore, there is no known polynomial time algorithm for model checking of TCTL properties; however, the $\text{TCTL}_{\leqslant,\geqslant}$ subset can be model checked in polynomial time.

On the other hand, as discussed in [25], there is a pseudo-polynomial algorithm for finding the EPL between two vertices in a weighted graph. The running time of this algorithm is $O(W^2V^3 + |k| \cdot \min\{|k|, W\} \cdot (V + E))$, where V is the number of vertices, E is the number of edges, k is the value which EPL looks for, and W is the biggest number in the set of absolute values of weights of edges. This algorithm works in the following two phases.

- **Preprocessing:** In this phase, the given graph is processed with a relaxation algorithm. As a result, the weights of the edges are updated such that the signs of the weights are the same in different paths (this algorithm works for graphs with positive, negative, and zero weight edges). The running time of this phase is $O(W^2V^3)$.
- **Finding-Path:** In the second phase, the EPL between the two input vertices is found in the relaxed graph. The running time of this phase is $O(|k| \cdot \min\{|k|, W\} \cdot (V + E))$.

In the case of finding the EPL in the FTS of a DTG, W is the biggest time elapse of the FTS transitions. The value of k is the time quantifier of the given TCTL$_=$ formula (e.g., for TCTL$_=$ formula $\exists\, \Phi \mathbf{U}^{=5} \Psi$ the value of k is five). This way, finding the EPL is possible in polynomial time as for a wide range of TCTL formulas, the time quantifiers are small constant values (in comparison to the size of the transition system). However, there is no limitation on the value of W.

Lemma 1. *There is an $O((V + E) \cdot |\Phi|)$ algorithm for model checking of DTGs against TCTL$_=$ property Φ with a small constant time quantifier k.*

Proof. As the FTS of a DTG has only progress-of-time states and transitions, the weights of all of the transitions are positive integer numbers (assume that the biggest weight is W) and there is no need for a relaxation phase with cost $O(W^2V^3)$. Therefore, the running time of the model checking algorithm is reduced to $O(|k| \cdot \min\{|k|, W\} \cdot (V + E))$.

On the other hand, the time quantifier is assumed to be a small constant integer number. Hence, k is a constant number in finding its corresponding EPL. Having a constant value for k, the value of $\min\{|k|, W\}$ is at most k. As a result, the time running time of finding the EPL in a state space is reduced from $O(|k| \cdot \min\{|k|, W\} \cdot (V + E))$ to $O(|k|^2 \cdot (V + E)) = O(V + E)$. □

Theorem 4. *Model checking of DTGs against TCTL property Φ with small constant time quantifiers is an $O((V \lg V + E) \cdot |\Phi|)$ problem.*

Proof. This follows directly from Corollary 2 and Lemma 1. □

5. Efficient TCTL$_{\leqslant,\geqslant}$ Model Checking of Timed Rebeca Models

In the previous section, we showed how DTGs could be model checked efficiently against TCTL properties. But, the DTG formalism does not support compositional modeling; so, it is hard to use it for modeling of complex real-time systems. Contrarily, high-level modeling languages support compositional modeling; however, the existing techniques for analyzing those models are inefficient. In this section, we show how the efficient model checking algorithm of DTGs can be used for the model checking of higher-level modeling languages, using automatic generation of DTGs for higher-level models. To this aim, we consider actor models, a well-established paradigm for modeling the functional behavior of distributed systems with asynchronous message passing. This model was originally introduced by Hewitt [10] and then elaborated by Agha [12, 11] and Talcott [13]. We develop a toolset based on this approach for the model checking of Timed Rebeca models [19, 16], an actor-based language for modeling concurrent and time-critical reactive systems. Later in this section, we will show how the proposed approach can be used for the model checking of the transition systems of Timed Rebeca model against TCTL properties. Also, we will show how the FTS of a Timed Rebeca model can be generated on-the-fly, using well-known graph search algorithms (DFS or BFS) in $O(V + E)$. In practice, the runtime overhead of on-the-fly generation of FTSs is negligible for the Timed Rebeca models.

5.1. A Timed Rebeca Model

In this section, we introduce Timed Rebeca using the example of a simple ticket service system. In this system, a client asks a ticket from an agent and the agent tries to issue a ticket by interacting with a ticket service server. A Timed Rebeca model (as the real-time extension of the Rebeca modeling language

[27, 28, 29]) consists of a number of *reactive classes*, each describing the type of a certain number of *actors* (called *rebecs* in Timed Rebeca)[2]. In the ticket service example (Figure 4), we have three reactive classes `TicketService`, `Agent`, and `Customer` which the size of their message bags are set to two, two, and one respectively. Each reactive class declares a set of *state variables.* The local state of each actor is defined by the values of its state variables and the contents of its message bag. Following the actor model, communication in the Timed Rebeca models takes place by asynchronous message passing among actors. Each actor has a set of *known rebecs* to which it can send messages. For example, an actor of type `TicketService` knows an actor of type `Agent` (line 2), to which it can send `ticketIssued` message (line 12). Each reactive class of a Timed Rebeca model may have some constructors. Constructors have the same name as the declaring reactive class and do not have a return value (line 6). They have the task of initializing the actor's state variables (lines 7 and 8) and putting initially needed messages in the bag of that actor (line 33). A properly written constructor leaves the resulting actor in a valid state. Reactive classes declare the messages to which they can respond. The way an actor responds to a message is specified in a *message server.* An actor can change its state variables through assignment statements (e.g., line 13), makes decisions through conditional statements (not appearing in our example), communicates with other actors by sending messages (e.g., line 12), and performs periodic behavior by sending messages to itself (e.g., line 39). Since communication is asynchronous, each actor has a *message bag* from which it takes the next incoming message. The ordering of the messages in a message bag is based on the arrival times of messages. An actor takes the first message from its message bag, executes its corresponding message server in an isolated environment, takes the next message (or waits for the next message to arrive) and so on. A message server may have a *nondeterministic assignment* statement which is used to model the nondeterminism in the behavior of a message server.

Finally, the `main` block is used to instantiate the actors of the model. In the ticket service model, three actors are created receiving their known rebecs and the parameter values of their constructors upon instantiation (lines 44-46).

Timed Rebeca adds three primitives to Rebeca to address timing issues: *delay*, *deadline* and *after.* A *delay* statement models the passage of time for an actor during execution of a message server (line 11). Note that all other statements of Timed Rebeca are assumed to execute instantaneously. The keywords *after* and *deadline* are used in conjunction with a method call. The term `after(n)` indicates that it takes n units of time for a message to be delivered to its receiver. For example, the periodic task of requesting a new ticket is modeled in line 39 by the customer sending a `try` message to itself and allowing the receiver (itself) to take it from its message bag only after 30 units of time. The term `deadline(n)` expresses that if the message is not taken in n units of time, it will be purged from the receiver's message bag automatically. For example, line 23 indicates that a `requestTicket` message must be started to execute before the passage of five units from the sending time of the message.

Note that a Rebeca model may contain some private methods. These methods cannot be called from the other actors and used to make the model of a reactive class more modular. The definition of a method starts with the type of its return value (instead of the `msgsrv` keyword) and its body is the same as the body of a message server.

5.2. The Fine-Grained Semantics of Timed Rebeca

In this section, we present the fine-grained semantics of Timed Rebeca based on the work of [30]. In the first step, we present formal definitions of a number of primitive concepts in Timed Rebeca. For a Timed Rebeca model $\mathcal{M}$, we assume that there is a universal set $\mathcal{I}$ which contains identifiers of all of the rebecs of $\mathcal{M}$. The specification of rebec r_i with the unique identifier $i \in \mathcal{I}$ is defined as the tuple $(\mathcal{V}_i, \mathcal{M}_i, \mathcal{K}_i)$ where $\mathcal{V}_i$ is the set of its state variables names, $\mathcal{M}_i$ is the set of its message servers, and $\mathcal{K}_i$ is the set of its known rebecs. The set of all possible values of the state variables of r_i is denoted by $Vals_i$.

[2]In this paper we use rebec and actor interchangeably.

```
reactiveclass TicketService (2) {
    knownrebecs {Agent a;}
    statevars {
        int issueDelay, nextId;
    }
    TicketService(int myDelay) {
        issueDelay = myDelay;
        nextId = 0;
    }
    msgsrv requestTicket() {
        delay(issueDelay);
        a.ticketIssued(nextId);
        nextId = nextId + 1;
    }
}

reactiveclass Agent (2) {
    knownrebecs {
        TicketService ts;
        Customer c;
    }
    msgsrv requestTicket() {
        ts.requestTicket() deadline(5);
    }
    msgsrv ticketIssued(byte id) {
        c.ticketIssued(id);
    }
}

reactiveclass Customer (1) {
    knownrebecs {Agent a;}
    Customer() {
        self.try();
    }
    msgsrv try() {
        a.requestTicket();
    }
    msgsrv ticketIssued(byte id) {
        self.try() after(30);
    }
}

main {
    Agent a(ts, c):();
    TicketService ts(a):(3);
    Customer c(a):();
}
```

Figure 4: The Timed Rebeca model of the ticket service system.

A (timed) message is defined as $tmsg = ((sid, rid, mid), ar, dl)$, denoting a message $m_{mid} \in \mathcal{M}_{rid}$ sent from the rebec r_{sid} to the rebec r_{rid}. This message is delivered to the message bag of the rebec r_{rid} at time $ar \in \mathbb{N}_0$ (its arrival time) and execution of the message server must be started before time $dl \in \mathbb{N}_0$ (its deadline). For the sake of simplicity, we assume that messages do not have parameters.

Each rebec r_i has a message bag $\mathcal{B}_i$ which can be defined as a multiset of timed messages. $\mathcal{B}_i$ stores the timed messages sent to r_i. The set of possible values for $\mathcal{B}_i$ is denoted by $Bags_i$.

We specify the semantics of the Timed Rebeca model $\mathcal{M}$ as a transition system in the form of a tuple $TS = (S, s_0, Act, \rightarrow, AP, L)$ where S is the set of states, s_0 is the initial state, Act is the set of actions, and $\rightarrow \subseteq S \times Act \times S$ is the transition relation. AP is the set of atomic propositions and $L : S \rightarrow 2^{AP}$ is the labeling function.

States. A given state $s \in S$ consists of the local states of the rebecs, together with the current time of the state. The local state of rebec r_i in state s is defined as tuple $(V_{s,i}, B_{s,i}, pc_{s,i}, res_{s,i})$, where

- $V_{s,i} \in Vals_i$ consists of the values of the state variables of r_i
- $B_{s,i} \in Bags_i$ is the message bag of r_i
- $pc_{s,i} \in \{null\} \cup (\mathcal{M}_i \times \mathbb{N})$ is the program counter, tracking the execution of the current message server with a pair in form of $\langle$`message_server_name`, `line_number`$\rangle$ ($null$ if r_i is idle in s)
- $res_{s,i} \in \mathbb{N}_0$ is the resuming time, if r_i is executing a `delay` statement in s

Initial State. s_0 is the initial state of Timed Rebeca model $\mathcal{M}$, where the values of the state variables of the rebecs are set to their initial values (according to their types), the effect of executing constructors of the rebecs are applied on their state variables and message bags, the program counters of all of the rebecs are set to $null$, and the time of the state is set to zero.

Actions. There are three types of actions: taking a message $tmsg = (sid, rid, mid, ar, dl)$, executing a statement of an actor (which we consider as an internal transition τ), and progress of time by $n \in \mathbb{N}$ units.

Hence, the set of actions is defined as

$$Act = \Big(\bigcup_{i \in \mathcal{I}} (\mathcal{I} \times i \times \mathcal{M}_i \times \mathbb{N} \times \mathbb{N})\Big) \cup \{\tau\} \cup \mathbb{N}$$

Transition Relations. Before defining the transition relation, we introduce $E_{s,i}$ as the set of *enabled messages* of the rebec r_i in the state s. The set of enabled messages contains messages whose arrival times are less than or equal to now_s. Transition relation $\rightarrow \subset S \times Act \times S$ is defined such that $(s, act, t) \in \rightarrow$ (or $s \xrightarrow{act} t$ in short) if and only if one of the following conditions holds.

1. **(Taking a message for execution)** In the state s, there exists a rebec r_i such that $pc_{s,i} = null$ and at least one timed message $tmsg$ is in $E_{s,i}$. Here, we have a transition in the form $s \xrightarrow{tmsg} t$. This transition results in extracting $tmsg$ from the message bag of r_i, setting $pc_{t,i}$ to the first statement of its corresponding message server, and setting $res_{t,i}$ to now_t (which is the same as now_s). Note that $V_{t,i}$ remains the same as $V_{s,i}$. These transitions are called *taking-event transitions.*
2. **(Internal action)** In the state s, there exists a rebec r_i such that $pc_{s,i} \neq null$ and $res_{s,i} = now_s$. The statement of the message server of r_i specified by $pc_{s,i}$ is executed and one of the following cases occurs based on the type of the statement. Here, we have a transition of the form $s \xrightarrow{\tau} t$.
   (a) Non-delay statements: the execution of a statement may change values of the state variables of the rebec r_i or cause a number of messages to be sent. Here, $pc_{t,i}$ is set to the next statement (or *null* if there are no more statements). All other elements of t are the same as those of s [19, 16].
   (b) Delay statement with parameter $d \in \mathbb{N}$: the execution of a delay statement sets $res_{t,i}$ to $now_s + d$. All other elements of the state remain unchanged. Particularly, $pc_{t,i} = pc_{s,i}$ because the execution of the delay statement is not yet finished. The value of the program counter will be set to the next statement after completing the execution of the delay statement (as shown in the following case).

   These transitions are called *internal transitions.*
3. **(Progress of time)** If in the state s none of the conditions of cases 1 and 2 hold, meaning that $\nexists r_i \cdot ((pc_{s,i} = null \wedge E_{s,i} \neq \emptyset) \vee (pc_{s,i} \neq null \ \wedge \ res_{s,i} = now_s))$, the only possible transition is a progress in time. In this case, now_t is set to $now_s + d$ where $d \in \mathbb{N}$ is the minimum value such that after progressing d time units, one of the aforementioned conditions holds. The transitions of this case are of the form $s \xrightarrow{d} t$. For a rebec r_i, if $pc_{s,i} \neq null$ and $res_{s,i} = now_t$ (the current value of $pc_{s,i}$ points to a delay statement), $pc_{t,i}$ is set to the next statement (or to *null* if there are no more statements). These transitions are called *progress-of-time transitions.* Note that taking-event transitions and internal transitions have higher priorities than the progress-of-time transitions. Therefore, progress-of-time transitions are disabled if a taking-event transition or an internal transition is enabled. So, a state which has a progress-of-time transition does not have any other outgoing transitions and this state is called *progress-of-time state.*

Atomic Propositions. Intuitively, atomic propositions express simple known facts about the values of state variables of rebecs in the states of the model under verification.

Labeling Function. Function $L : S \rightarrow 2^{AP}$ relates a set of atomic propositions to each state, shown by $L(s)$ for a given state s. Intuitively, $L(s)$ labels state s by the atomic propositions it satisfies.

There is no explicit time reset operator in Timed Rebeca; so, progress of time results in an infinite number of states in transition systems of Timed Rebeca models. However, reactive systems which generally show periodic or recurrent behaviors are modeled using Timed Rebeca; in other word, they perform periodic behaviors over infinite time. Based on this fact, in [31] we presented a new notion for equivalence relation between two states to make the transition systems finite, called *shift equivalence relation.* In shift equivalence relation two states are equivalent if and only if they are the same except for the parts related to the time (value of *now*, arrival times of messages, and deadlines of messages) and shifting the times of those parts in one state makes it the same as the other one. This way, instead of preserving absolute value of time, only the relative difference of timing parts of states is preserved. As discussed in [31], shift equivalence relation makes transition systems of the majority of Timed Rebeca models finite.

5.3. Model Checking of Timed Rebeca Models

Prior to proposing model checking for Timed Rebeca models, given Timed Rebeca models must be analyzed to be Zeno-free [3], as the prerequisite of any further timed analysis.

As the model of time in Timed Rebeca is discrete, the execution of an infinite number of message servers in zero time is the only scenario of exhibiting Zeno behavior, since the minimum elapse of time in Timed Rebeca is one unit. Therefore, if there is a cycle in the state space of a Timed Rebeca model which does not contain progress-of-time states, the model exhibits Zeno behavior. This can be detected by a depth-first-search (DFS) in $O(V + E)$, as shown in Algorithm 4. In this algorithm, we assume that a Boolean variable is associated with each state indicating whether the state is in the search stack, called `recStack`. The condition in line 11 of the algorithm checks if the state s' is re-visited in zero time. As mentioned in the semantics of Timed Rebeca, the function $now(\cdot)$ returns the time of its given state.

Algorithm 4: *ZenoFree*(s) analyzes the state space of a model for Zeno-freedom.

```
Input: State $s$ of a fine-grained transition system $T$
Output: The part of $T$ reachable from $s$ is Zeno-free or not
1  begin
2      $visited \leftarrow \varnothing$
3      foreach $state\ s' \in$ SUCCESSORS($s$) do
4          if $s' \notin visited$ then
5              $visited \leftarrow visited \cup \{s'\}$
6              recStack($s'$) $\leftarrow true$
7              if ZenoFree($s'$)$=false$ then
8                  return $false$
9              recStack($s'$) $\leftarrow false$
10         else
11             if recStack($s'$) $= true \wedge$ now($s'$) $=$ now($s$) then
12                 return $false$
13     return $true$
```

In line 3 of Algorithm 4, the `foreach` statement traverses all transitions of the transition system. As the processing time of each transition is constant, the overall running time of the algorithm is $O(V + E)$.

Based on the fact that a given Timed Rebeca model is Zeno-free and its fine-grained transition system is a DTG, the newly proposed TCTL$_{\leqslant,\geqslant}$ model checking algorithm in Section 3 can be used for the model checking of Timed Rebeca models. This fact is stated in Lemma 2.

Lemma 2. *The fine-grained transition system of a Timed Rebeca model is a DTG.*

As a result, for a given TCTL$_{\leqslant,\geqslant}$ formula Φ, the polynomial time algorithm of DTG model checking can be used for model checking the fine-grained transition systems of Timed Rebeca models.

Corollary 3. *There is an $O((V \lg V + E) \cdot |\Phi|)$ algorithm for model checking Timed Rebeca models against TCTL$_{\leqslant,\geqslant}$ property Φ.* □

5.4. Model Checking of the FTSs of Timed Rebeca Models

As the second step for the efficient model checking of Timed Rebeca models, we will show that how the FTSs of Timed Rebeca models are generated without a significant runtime overhead. As the following lemma (together with Algorithm 5) illustrates, we can combine generating the state space, checking for Zeno behavior, and generating the FTS to decrease the execution cost of the generation of FTSs. In this algorithm, transition systems are generated using Bounded-DFS.

Algorithm 5: The state space is generated based on the given initial state or `null` is returned in case of Zeno behavior in the model.

```
Input: An initial state $s_0$ of a model
Output: The result FTS or null if the model has Zeno behavior
$V \leftarrow \{s_0\}$                                      ▸ The set of all of the states
$hasZeno \leftarrow false$                                  ▸ Flag for Zeno behavior detection
begin
    $S \leftarrow \{s_0\}$                                  ▸ The set of the states of result FTS
    $N \leftarrow$ ENQUEUE($s_0$)                           ▸ The set of the next level states
    $\hookrightarrow \leftarrow \varnothing$                ▸ The set of the transitions of FTS
    while HAS_ELEMENTS($N$) $\wedge \neg hasZeno$ do
        $s \leftarrow$ DEQUEUE($N$)
        $N' \leftarrow$ Bounded_DFS($s$)
        foreach state $s' \in N$ do
            $time \leftarrow$ PROGRESS_OF_TIME($s$)
            $S \leftarrow S \cup s'$
            $\hookrightarrow \leftarrow \hookrightarrow \cup (s, time, s')$
        $N \leftarrow$ ENQUEUE_ALL($N'$)
    if $hasZeno = true$ then
        return null
    else
        return $(S, s_0, \mathbb{N}, \hookrightarrow, AP, L)$
Procedure $Bounded\_DFS$(s)
    $Q \leftarrow$ GENERATE_SUCCESSOR_STATES($s$)
    $R \leftarrow \varnothing$                              ▸ The set of states of npts($s$)
    foreach state $s' \in Q$ do
        if IS_PROGRESS_OF_TIME($s'$) then
            $R \leftarrow R \cup s'$
            continue
        else
            if $s' \notin V$ then
                $V \leftarrow V \cup s'$
                recStack($s'$) $\leftarrow true$
                $R \leftarrow R \cup$ Bounded_DFS($s'$)
                recStack($s'$) $\leftarrow false$
            else
                if recStack($s'$) $= true \wedge$ now($s'$) $=$ now($s$) then
                    $hasZeno \leftarrow true$
    return $R$
```

Starting from the initial state s_0, the set of the nearest progress of time states of the initial state ($npts(s_0)$) are generated (in the first iteration of the `while` loop in lines 7 to 14). At the next iteration, for each state of $npts(s_0)$, its set of the nearest progress of time states are found, and so on. As in each iteration only states between consecutive progress-of-time states are generated in a DFS manner, the algorithm is called Bounded-DFS state space generation. Like ordinary DFS, Bounded-DFS is a recursive procedure, defined in lines 19 to 35. In each round, if a progress-of-time state is found, it is put in set R as the return value (line 24).

Otherwise, Bounded-DFS is invoked to explore the successor states of the newly generated state (lines 26 to 34); meanwhile, the existence of a cycle without an elapse of time is checked to detect Zeno behavior (line 33). This way, as each state is visited at most twice (at the generation time and when DFS continues exploration through its successors) and each transition is traversed once (at the generation time), the overall running time of checking for Zeno behavior and generating the FTS is $O(V + E)$.

Note that in Algorithm 5, the function `PROGRESS_OF_TIME` maps its given progress-of-time state to the value of its only outgoing timed transition.

Lemma 3. *The FTS of a given Timed Rebeca model $\mathcal{M}$ can be generated in $O(V + E)$.* □

Corollary 4. *The FTS of a given Timed Rebeca model can be generated and model checked against TCTL property Φ in $O((V \lg V + E) \cdot |\Phi|)$.*

6. Case Studies and Experimental Results

We perform four different case studies in different sizes to illustrate how efficiently the reduction technique works. The host computer of the model checking toolset was a desktop computer with 1 CPU (2 cores) and 8GB of RAM storage, running El Capitan OS X 10.11.5. The selected case studies are a simplified version of a *NoC* system with 16 cores, a simplified version of the *Scheduler of Hadoop*, a *Ticket Service* system, and an application of *Wireless Sensor and Actuator Networks (WSAN)*. The Timed Rebeca source codes of these case studies and the model checking toolset (Afra) are accessible from the Rebeca home page[3]. As shown in Figure 5, the Timed Rebeca source codes and their corresponding TCTL properties are transformed to some C++ files using RMC component and executing the C++ files results in generating state spaces of the models in the XML format. Afra benefits from another component, state space analyzer component, for analyzing the generated state spaces. We developed the proposed TCTL model checking algorithm as a part of this component. We also developed the old TCTL model checking algorithm in this component to be able to measure how efficient is the work of this paper.

For each case study, we provide both an intuitive and a detailed description of the model, and then discuss the gained reduction. We also present the TCTL formula which the model is model checked against it. In the presented TCTL formulas, atomic propositions are defined as boolean expressions based on the values of the state variables of actors. For example, the atomic proposition which shows the equality of the state variable `x` of actor `a` to 3 is shown by $a.x == 3$. We choose the state space size and the model checking time consumptions as the performance metrics of the model checking algorithms. The values of these metrics are compared in a table for each case study. In the tables, ORG is used to refer to the original state spaces and RED is used to refer to the reduced state space (i.e., FTSs). We also use OLD to refer to the old TCTL model checking algorithm and NEW to refer to the proposed TCTL model checking algorithm of this paper. In the case of having both RED and NEW, we address the cases where reduced state spaces are model checked using the TCTL model checking algorithm of this paper. We also reported the spent time for the state space generation. As mentioned before, there is no difference between the spent times for the generation of the original state spaces and the reduced state spaces, so only one number is reported as the spent times.

Note that the simplified source codes of the examples are shown in the figures of this section and many parts of them are eliminated (they replaced by dots). As mentioned before, the complete source codes of the models are in the Rebeca home page.

[3]`http://www.rebeca-lang.org/wiki/pmwiki.php/Examples/Examples`

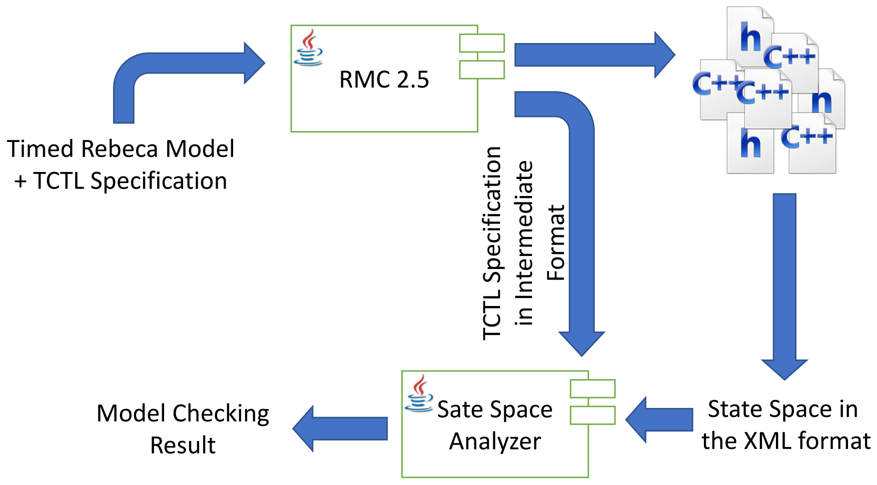


Figure 5: How TCTL model checker of Afra works

6.1. Network on Chip (NoC)

Our first example is a model of a network on chip (NoC), a promising architecture paradigm for many-core systems. In NoC designs, functional verification and performance evaluation in the early stages of the design process are suggested as ways to reduce the fabrication cost. As an example of a NoC, we modeled and analyzed ASPIN (Asynchronous Scalable Packet switching Integrated Network), which is a fully asynchronous two-dimensional NoC design [32]. In a two-dimensional NoC design, each core is placed in a 2D mesh and has four adjacent cores and four internal buffers for storing the incoming packets (one for each direction). Different routing algorithms have been proposed for the two-dimensional NoC deisign, including XY, OE, and DYAD routing algorithms. In the following example, we consider the XY routing algorithm. Using the XY routing algorithm, packets are moving along the X direction first, and then along the Y direction, to reach their destination cores. In ASPIN, packets are transferred through channels, using a four-phase handshake communication protocol. The protocol uses two signals, namely *Req* and *Ack*, to implement this four-phase handshaking protocol. This way, to transfer a packet, first the sender sends a request by raising the *Req* signal, and waits for an acknowledgment which is the raising of the *Ack* signal by the receiver. In the third phase, the data is sent. Finally, after a successful communication all of the signals return to zero.

The timed version of ASPIN was investigated in [33] using simulation and model checking against *deadlock freedom* and *schedulability* properties. In addition to the functional correctness, the Afra toolset was used for estimating the maximum end-to-end latency of the model.

Timed Rebeca Model. The simplified version of the Timed Rebeca model of ASPIN is shown in Figure 6, which contains two different reactive classes: `Manager` and `Router`. The `Manager` does not exist in real NoC systems. Here, it is used as the starter of the model. It sends the combination of `inReq` and `inReqMinus` messages to a router to ask for packet generation. This way, different traffic scenarios are generated by modifying the code of `Manager`. In the example of Figure 6, one packet is generated in the router `r00` which must be routed to the router `r11` (Lines 19 and 20). To make sure successful delivery of this packet, two other messages are sent in lines 21 and 22. Using this pattern, different traffics can be generated easily. `Router` is the model of a core in an ASPIN design. Its specification contains four known rebecs which are its neighbor cores (line 29), a composite id which includes its X-Y position (line 32), buffer variables which show that the buffers are enabled or busy (line 33), a variable which counts the number of received packets (`received` in line 32), and many other control variables. The communication channel between neighbors is modeled by the message passing of Rebeca. Trying for the delivery of a packet is started by sending an `inReq` message to a router. The receiver router accepts the packet if its input buffer is free (line 48).

```
env short maxTime = 28000;
env short rAlg = 1;
env byte writeD = 2;
...
reactiveclass Manager(60){
  knownrebecs{
    Router r00, r10, r20, r30,
         r01, r11, r21, r31,
         r02, r12, r22, r32,
         r03, r13, r23, r33;
  }
  Manager(){
    generate();
  }
  msgsrv reset(){ ... }
  void generate(){
    r01.reStart()after(wholeCycle);
    r11.checkRecieved(2) after(maxTime);
    r00.inReq(4,1,1,1) after (18);
    r00.inReqMinus(4) after (18 + prodD);
    r00.inReq(4,1,1,2) after (110);
    r00.inReqMinus(4) after (110 + prodD);
    ...
  }
}
reactiveclass Router(80) {
  knownrebecs {
    Manager manager;
    Router N, E, S, W;
  }
  statevars {
    byte Xid, Yid, received;
    bboolean[5] inBufFull, outBufFull
    byte[5][2] outPortPtr;
    ...
  }
  Router(byte X, byte Y){
    Xid = X;
    Yid = Y;
    for(byte i=0;i<5;i++){
      waitedOutReq[i] = 5;
      outReqEnable[i] = true;
      outPortPtr [i][0]= -1;
    }
    ...
  }
  msgsrv inReq (byte inPort, byte Xtarget, byte Ytarget,byte id){
    if (inBufFull[inPort] == false ){
      sendInAck((byte)(inPort + 2)% 4, inAD);
      self.process(inPort, Xtarget, Ytarget,id, false, false)after((writeD * inBufSizeTest)+ readD);
      ...
    } else { ... }
  }
  msgsrv process(byte inPort, byte Xtarget, byte Ytarget,byte id, boolean isPushed, boolean justPush) {
    byte routeD;
    ...
    if ((inBufID[inPort][0] == id) || isPushed == true){
      if(passedFlit == 0) {
        if (rAlg == 1) {
          outPort = XYrouting(Xtarget, Ytarget);
          routeD = routeXYD;
        }
        else if (rAlg == 2){ ... }
        else if (rAlg == 3){ ... }
      } else { ... }

      if(outReqEnable[inPort] == true){
        waitedOutReq[inPort] = outPort;
        self.portSchedule(outPort, inPort) after(routeD + schdD + outRD);
      }
    }
  }
  byte XYrouting(byte Xtarget, byte Ytarget) {
    byte outPort = 0;
    if(Xtarget > Xid)  outPort = 1;
    else if(Xtarget < Xid) outPort = 3;
    else if(Ytarget > Yid) outPort = 2;
    else if(Ytarget < Yid) outPort = 0;
    else outPort = 4;
    return outPort;
  }
  ...
}
main {
  Manager m(r00, r10, ..., r33):();
  Router r00(m,r03,r10,r01,r30):(0,0);
  Router r10(m,r13,r20,r11,r00):(1,0);
  Router r20(m,r23,r30,r21,r10):(2,0);
  Router r30(m,r33,r00,r31,r20):(3,0);

  Router r01(m,r00,r11,r02,r31):(0,1);
  Router r11(m,r10,r21,r12,r01):(1,1);
  Router r21(m,r20,r31,r22,r11):(2,1);
  Router r31(m,r30,r01,r32,r21):(3,1);

  Router r02(m,r01,r12,r03,r32):(0,2);
  Router r12(m,r11,r22,r13,r02):(1,2);
  Router r22(m,r21,r32,r23,r12):(2,2);
  Router r32(m,r31,r02,r33,r22):(3,2);

  Router r03(m,r02,r13,r00,r33):(0,3);
  Router r13(m,r12,r23,r10,r03):(1,3);
  Router r23(m,r22,r33,r20,r13):(2,3);
  Router r33(m,r32,r03,r30,r23):(3,3);
}
```

Figure 6: The model of an ASPIN NoC

Configuration	State Space Generation				Model Checking Time		
	states ORG	states RED	Gain	Time	ORG, OLD	ORG, NEW	RED, NEW
3 Packets	442	68	84%	1s	<1s	<1s	<1s
4 Packets	1,239	122	91%	2s	6s	<1s	<1s
5 Packets	3,117	126	96%	7s	2.8m	<1s	<1s
6 Packets	9,907	129	98%	35s	40m	1s	<1s
7 Packets	35,746	102	99%	6.8m	>5h†	5s	<1s
8 Packets	136,666	117	99%	1.4h	>5h†	16s	<1s

Table 1: The size of the state spaces and the gained reductions in the NoC example in different scenarios. The † sign on the reported times shows that the model checking passed the time limit (5 hours).

Upon accepting a packet, an acknowledgment is sent to its sender and an internal message is scheduled to process this packet (lines 49 and 50). Processing of a packet takes place in message server `process`. If there is a packet for processing (line 58), one of the routing algorithms is selected to send the packet to the appropriate neighbor (lines 59 to 64). As shown the details of routing by XY algorithm in line 60, the output port of a packet is computer by the private method `XYrouting`. As shown in lines 75 to 79 the destination port of a packet is computed based on the value of X and Y of both the source router and the destination router. The 2D mesh of this model is formed in the `main` block of the model by setting known rebecs based on the locations of the routers.

Gained Reduction. We model checked this model against $\mathbf{E}(r11.\mathit{received} <= 2)\,\mathbf{U}^{\leqslant 250}(r11.\mathit{received} > 2)$ formula. This formula makes sure that before passing 250 time units more than two packets are received by the router `r11`. As shown in Table 1, sending 7 or 8 packets results in passing the time limit of the model checking (we set it to 5 hours) in case of using the old model checking algorithm. However, the new algorithm computes the results in a reasonable time.

The effect of applying the reduction technique is shown in Table 1 too. In the NoC model, increasing the number of sent packets results in a light increment in the gained reduction, which is because of the increment of the concurrency level of the model. In other word, increasing the number of packets results in interleaving of transitions which correspond to routing the packets. The interleaving of these transitions are omitted in the FTS of the model and as there is no conflict between the routes of the packets (it is because of the traffic pattern we have chosen for this model), eliminating the effect of the interleaving of transitions results in FTSs which have approximately the same sizes.

Table 1 also shows that using the new model checking algorithm together with the FTS reduction technique results in the model checking of the models in less than a second.

6.2. Hadoop YARN Scheduler

Hadoop [34] is a framework for MapReduce, a programming model for generating and processing large data sets [35]. MapReduce has undergone a complete overhaul in its latest release, called MapReduce 2.0 (MRv2) or YARN [36]. The fundamental idea of YARN is to split up the major functionalities of the framework into two modules, a global ResourceManager (RM) and per-application ApplicationMaster (AM). RM arbitrates resources among all of the applications in the system. AM negotiates with RM for the resources to manage the life cycle of its running applications. So, on a Hadoop cluster, there is a single RM and for every job there is a single AM. It is possible to set different policies in YARN for dispatching jobs and resources to AMs based on the deadlines, the jobs priorities, the arrival times of jobs, etc.

Timed Rebeca Model. In the Timed Rebeca model of Figure 7, the YARN system is modeled using two different reactive classes: `ResourceManager`, `ApplicationMaster`. Message server `checkQueue` models the main behavior of RM by looking for a free AM and assigning a job to it. Lines 34 to 43 of `checkQueue` illustrate how a job is assigned to `am1` (the first Application Master) if the status of `am1` is `FREE`. The

Configuration	State Space Generation				Model Checking Time		
	states ORG	states RED	Gain	Time	ORG, OLD	ORG, NEW	RED, NEW
1 AMs	180	56	69%	<1s	<1s	<1s	<1s
2 AMs	5,506	1,283	77%	1s	16s	<1s	<1s
3 AMs	177,989	24,639	86%	14.5m	>5h†	18.8m	<1s

Table 2: The size of the state spaces and the gained reductions in the Hadoop Yarn example with default configuration. The † sign on the reported times shows that the model checking passed the time limit (5 hours).

specification of the job which is sent to `am1` is in the head of the queue of jobs (line 9). After sending the specification, the job is removed from the queue of jobs (lines 38 to 41) and another job is generated and added to the queue of jobs to model the arrival of a new job (line 42). The same behavior is implemented for the other AMs. In `ResourceManager`, state variable `fifoQueue`, as the queue of jobs, keeps track of the deadlines of jobs. In lines 48 to 58 of `checkQueue`, the deadlines of jobs are decreased by one unit to model the time elapse for waiting jobs.

In this model we simplified the behavior of application masters to perform their assigned jobs successfully. This takes place by setting 2 as the completion time of all jobs (line 90). Setting this value to more than the value of `dline` results in missing the deadline and non-successful termination of the job. As shown in line 98, each application master keeps the number of the performed jobs. To avoid state space explosion, the value of this counter is set to 0 after performing 5 successful jobs (line 99).

Gained Reduction. We used $\mathbf{E}(am2.doneJobs <= 4)\,\mathbf{U}^{\leqslant 10}(am2.doneJobs > 4)$ formula for the model checking of the Yarn model. This formula makes sure that before passing 10 time units the second application master finishes five jobs (the same property can be checked for the other application masters). As shown in Table 2, having 3 application master results in passing the time limit of the model checking in case of using the old model checking algorithm. However, the new algorithm terminates in 18 minutes. Although the model checking time of the new algorithm is reasonable even for the case of three application masters, the time is reduced to less than one second when the new algorithm and FTS technique work together, as shown in Table 2.

The same as the model of NoC applying FTS technique reduces the size of the state spaces significantly. Also, increasing the number of the application masters increases the gained reduction. It is because of the fact that the application masters are working in parallel and the interleaving of their parallel activities is eliminated by FTS.

6.3. Ticket Service

Our third example is the model of a *Ticket Service* system. The overview of this example is presented in Section 5. We created the extended version of this model and varying in the number of customers.

Timed Rebeca Model. The Timed Rebeca model of this system for the case of five customers, shown in Figure 8, contains three different reactive classes: `Customer`, `Agent`, and `TicketService`. Customers periodically ask for tickets by sending `requestTicket` to the agent in message server `try` (line 13). Upon sending `requestTicket`, the customer sets its state variable `sent` to true to show that it sends a ticket request. This variable will be used in a TCTL formula which measures the service time of the system. `Agent` forwards the received requests immediately to `TicketService`. As specified by the `deadline` primitive (line 24), the forwarded request must be served before the passage of 24 units of time. The ticket service system issues a ticket and informs `Agent` about the issued ticket (line 38). This process takes 2 units of time, which is specified in line 37. `Agent` sends the issued ticket to its corresponding customer (line 27) and the customer unsets its state variable `sent`.

```
reactiveclass ResourceManager(5) {
  knownrebecs {
    AppMaster am1, am2, am3;
  }
  statevars {
    int FREE, BUSY;
    int appMaster1, appMaster2, appMaster3;
    int m_queue_misses, m_update_miss,
         m_job_complete, DEFAULT_DEADLINE,
         QUEUE_SIZE;
    int[4] fifo_queue;
  }

  ResourceManager() {
    FREE = 1;
    BUSY = 0;
    appMaster1 = FREE;
    appMaster2 = FREE;
    appMaster3 = FREE;
    m_queue_misses = 0;
    m_update_miss = 0;
    m_job_complete = 0;
    DEFAULT_DEADLINE = 3;
    fifo_queue[0] = DEFAULT_DEADLINE;
    fifo_queue[1] = DEFAULT_DEADLINE;
    fifo_queue[2] = DEFAULT_DEADLINE;
    fifo_queue[3] = DEFAULT_DEADLINE;
    QUEUE_SIZE = 4;
    self.checkQueue();
  }
  msgsrv checkQueue() {
    m_queue_misses = 0;
    m_update_miss = 0;
    m_job_complete = 0;
    int I = 0;
    if(appMaster1 == FREE) {
      appMaster1 = BUSY;
      am1.runJob(fifo_queue[0]);
      I = 0;
      while(I < QUEUE_SIZE - 1) {
        fifo_queue[I] = fifo_queue[I + 1];
        I++;
      }
      fifo_queue[QUEUE_SIZE - 1] =
           DEFAULT_DEADLINE;
    }
    if(appMaster2 == FREE) { ... }
    if(appMaster3 == FREE) { ... }
    I = 0;
    int J = 0;
    while(I < QUEUE_SIZE) {
      fifo_queue[I]--;
      if(fifo_queue[I] == 0) {
        m_queue_misses++;
        J = I;
        while(J < QUEUE_SIZE - 1) {
          fifo_queue[J] = fifo_queue[J + 1];
          J++;
        }
        fifo_queue[QUEUE_SIZE - 1] =
             DEFAULT_DEADLINE;
      }
      I++;
    }
    self.checkQueue() after(1);
  }
  msgsrv update(boolean deadline_miss) {
    m_queue_misses = 0;
    m_update_miss = 0;
    m_job_complete = 0;
    if(deadline_miss == true) {
      m_update_miss = 1;
    } else {
      m_job_complete = 1;
    }
    if(sender == am1) {
      appMaster1 = FREE;
    } else if(sender == am2) {
      appMaster2 = FREE;
    } else if(sender == am3) {
      appMaster3 = FREE;
    }
  }
}

reactiveclass AppMaster(5) {
  knownrebecs {
    ResourceManager rm;
  }
  statevars { int doneJobs; }

  AppMaster() { doneJobs = 0; }
  msgsrv runJob(int dline) {
    int completion = 2;
    boolean deadline_miss;
    if(completion > dline) {
      deadline_miss = true;
      rm.update(deadline_miss) after(dline);
    } else {
      deadline_miss = false;
      rm.update(deadline_miss) after(completion);
      doneJobs++;
      if (doneJobs > 5) doneJobs = 1;
    }
  }
}
main {
  ResourceManager rm(am1, am2, am3):();
  AppMaster am1(rm):();
  AppMaster am2(rm):();
  AppMaster am3(rm):();
}
```

Figure 7: The model of a Hadoop YARN system with three application masters

```
reactiveclass Customer(3) {
    knownrebecs { Agent a; }
    statevars {
        byte id;
        boolean sent;
    }
    Customer(byte myId) {
        id = myId;
        sent = false;
        self.try();
    }
    msgsrv try() {
        a.requestTicket();
        sent = true;
    }
    msgsrv ticketIssued() {
        sent = false;
        self.try() after(30);
    }
}
reactiveclass Agent(10) {
    knownrebecs { TicketService ts; }
    msgsrv requestTicket() {
        ts.requestTicket((Customer)sender)
             deadline(24);
    }
    msgsrv ticketIssued(Customer customer) {
        customer.ticketIssued();
    }
}
reactiveclass TicketService(10) {
    knownrebecs { Agent a; }
    statevars { int issueDelay; }
    TicketService(int myIssueDelay) {
        issueDelay = myIssueDelay;
    }
    msgsrv requestTicket(Customer customer) {
        delay(issueDelay);
        a.ticketIssued(customer);
    }
}
main {
    Agent a(ts):();
    TicketService ts(a):(2);
    Customer c1(a):(1);
    Customer c2(a):(2);
    Customer c3(a):(3);
    Customer c4(a):(4);
    Customer c5(a):(5);
}
```

Figure 8: The model of a ticket service system with five customers

Configuration	State Space Generation				Model Checking Time		
	states ORG	states RED	Gain	Time	ORG, OLD	ORG, NEW	RED, NEW
2 customers	77	10	87%	<1s	<1s	<1s	<1s
3 customers	360	40	89%	<1s	<1s	<1s	<1s
4 customers	1,825	184	90%	<1s	1s	1s	<1s
5 customers	10,708	1,047	90%	6s	2s	1s	<1s
6 customers	73,461	6,997	91%	3.4m	2.2m	1.7m	1s

Table 3: The size of the state spaces and the gained reductions in the Ticket Service example with different numbers of customers.

Gained Reduction. Making sure about the upper bound of the response time to the customers' requests is the property we checked for this model. We have to make sure that in all states, the time elapse between sending a request and receiving a ticket is less than a specific number. In the following formula, we ensure that in case of five customers, there is an upper bound of 16 time units for the response time of the system.

$$\mathbf{AG}^{\leqslant 50}((c1.\mathit{sent} \rightarrow \mathbf{AF}^{\leqslant 16}\neg c1.\mathit{sent}) \wedge \cdots \wedge (c5.\mathit{sent} \rightarrow \mathbf{AF}^{\leqslant 16}\neg c5.\mathit{sent}))$$

Note than this formula has to be transformed into the base form which only contains existential until modalities using $\mathbf{AG}^{\leqslant c}\phi \equiv \neg\,\mathbf{EF}^{\leqslant c}\neg\phi \equiv \mathbf{E}\;\mathit{true}\,\mathbf{U}^{\leqslant c}\neg\phi$ and $\mathbf{AF}^{\leqslant c}\phi \equiv \neg\,\mathbf{E}\neg\phi\,\mathbf{U}^{\geqslant c}\mathit{true} \wedge \neg\,\mathbf{E}\neg\phi\,\mathbf{U}P_{scc^0(\neg\phi)}$. As the state spaces are checked to be Zeno free prior to start the TCTL model checking, $\mathbf{E}\neg\phi\,\mathbf{U}P_{scc^0(\neg\phi)}$ is empty and there is $\mathbf{AF}^{\leqslant c}\phi \equiv \neg\,\mathbf{E}\neg\phi\,\mathbf{U}^{\geqslant c}\,\mathit{true}$.

The numbers of Table 3 shows that both of the algorithms perform model checking in a reasonable time. However, the algorithm of this paper is less than two times better than the old one. The gained performance of the new TCTL model checking algorithm in this example is not as significant as the aforementioned two examples because of the fact that a limited number of states pass the first phase of the old algorithm. Therefore, there are few states which have to pass the second phase of the algorithm, which is a costly algorithm. In the previous examples, all of the states pass the first phase, result in executing the second phase algorithm over all of the states. Table 3 shows that combining the new algorithm and FTS improves the performance of the model checking. The same as the previous examples, applying FTS technique reduces the size of the state spaces significantly and increasing the number of the customers increases the gained reduction.

6.4. WSAN Applications

As the fourth example, we present a real-time data acquisition system for structural health monitoring and control (SHMC) of civil infrastructures [37]. This system has been implemented on top of the Imote2 [38] wireless sensor platform, and has been deployed for long-term monitoring of several highway and railroad bridges. The SHMC application development has proven to be particularly challenging: it has the complexity of a large-scale distributed system with real-time requirements, while having the resource limitations of low-power embedded WSAN platforms. Ensuring the safe execution of a SHMC requires modeling the interactions between the components of the data acquisition nodes, which are CPU, sensor, and radio transmission components, as well as interactions between the nodes. In this application, all periodic tasks (sample acquisition, data processing, and radio transmission) are required to be completed before the start of their next period. In addition, each node has to send its processed data to a central station. To handle the communication between the nodes and the central station, a communication protocol is required. The schedulability of the models of this application using Timed Rebeca is investigated in [39]. Here, we showed that how other properties can be model checked using the TCTL model checking of Timed Rebeca.

Timed Rebeca Model. The simplified version of the Timed Rebeca model of WSAN, shown in Figure 9, contains five different reactive classes: `Sensor`, `CPU`, `Misc` (for miscellaneous tasks unrelated to sensing or communication), `CommunicationDevice`, and `WirelessMedium`. The model of a WSAN node concerns the data acquisition, processing, and radio transmission primarily. Having `Sensor`, `CPU`, and `CommunicationDevice` for a WSAN node, the developed Timed Rebeca model closely mimics the structure of the real application. The configuration of this model is specified by the values of the environment variables in lines 1 to 7. Based on these values, there are six nodes in the environment (line 2) and the sampling rate of the nodes is 25 samples per 1000 units of time (line 1). Each node packs two acquired data elements in one packet (line 3). The time spent for the internal activities of a node is specified in lines 4 to 6.

The main activity of this model is started by executing `sensorLoop` of `Sensor`. In this loop, based on the specified sampling rate, data is acquired by `Sensor` and it is sent to `CPU` (lines 17-21). There is the same behavior in `Misc`. These two actors send messages to `CPU`, which are handled by the `sensorEvent` and `miscEvent` message servers respectively (lines 33-35 and line 46). The message server `sensorEvent` starts the processing of the acquired data by sending a `sensorTask` message. In `sensorTask`, the schedulability of the processing of the acquired data is checked (lines 37 and 38), it is packed into one packet (line 40), and the packed data is sent by the communication device of this node if it reaches the limit which is specified by `bufferSize`. The communication protocol between nodes is implemented in the method `send` of `Communication Device` (We developed TDAM and MACB communication protocols in [39]). In the current implementation, before sending data, the freedom of the communication device is checked (line 64) then the needed messages are scheduled for sending data (line 68). To model the effect of Ether is the wireless communication and transmission conflict, we developed `WirelessMedium`. Communication devices send `broadcast` messages to the wireless medium to send data to other communication devices and the receivers of broadcast data send `broadcastingIsCompleted` to inform it received the data successfully.

Gained Reduction. Checking for utilizing the communication channel in each 50 units of time is the property we used for the model checking of this example with different configurations. This property is shown by $\mathbf{AG}^{\leqslant 50}(\,\mathbf{A}(\mathit{freeChannel})\,\mathbf{U}^{\leqslant 50}(\neg \mathit{freeChannel})\,)$ which has to be transformed to the base forms, as we did in the previous example. We verified the WSAN application in different configurations, varying the value of the sampling rate, the number of nodes, the packet size, and the sensor task delay. The results of

```
env int samplingRate = 25;
env int numberOfNodes = 6;
env int bufferSize = 2;
env int sensorTaskDelay = 2;
env int OnePacketTransmissionTime = 7;
env int miscTaskDelay = 10;
env int tmdaSlotSize = 10;
env int miscPeriod = 120;
env int packetMaximumSize = 112;

reactiveclass Sensor(10) {
    knownrebecs { CPU cpu; }
    Sensor() { self.sensorFirst(); }
    msgsrv sensorFirst() {
        self.sensorLoop() after(?(10, 20, 30));
    }
    msgsrv sensorLoop() {
        int period = 1000 / samplingRate;
        cpu.sensorEvent(period);
        self.sensorLoop() after(period);
    }
}

reactiveclass Misc(10) { ... }

reactiveclass CPU(10) {
    knownrebecs {
        CommunicationDevice senderDevice,
             receiverDevice;
        Sensor sensor;
    }
    statevars { int collectedSamplesCounter; }
    CPU() { collectedSamplesCounter = 0; }
    msgsrv sensorEvent(int period) {
        self.sensorTask(period,
             currentMessageWaitingTime);
    }
    msgsrv sensorTask(int period, int lag) {
        int tmp = period - lag -
             currentMessageWaitingTime;
        assertion(tmp >= 0);
        delay(sensorTaskDelay);
        collectedSamplesCounter += 1;
        if (collectedSamplesCounter == bufferSize){
            senderDevice.send(receiverDevice, 0, 1);
            collectedSamplesCounter = 0;
        }
    }
    msgsrv miscEvent() { delay(miscTaskDelay); }
}

reactiveclass CommunicationDevice (10) {
    knownrebecs { WirelessMedium medium; }
    statevars {
        byte id;
        int sendingData;
        int sendingPacketsNumber;
        CommunicationDevice receiverDevice;
    }
    CommunicationDevice(byte myId) {
        id = myId;
        sendingData = 0;
        sendingPacketsNumber = 0;
        receiverDevice = null;
    }
    msgsrv send(CommunicationDevice receiver, int
         data, int packetsNumber) {
        assertion(receiverDevice == null);
        sendingPacketsNumber = packetsNumber;
        receiverDevice = receiver;
        sendingData = data;
        medium.getStatus();
    }
    msgsrv receiveStatus(boolean result) { ... }
    msgsrv receiveResult(boolean result) { ... }
    msgsrv receiveData(CommunicationDevice
         receiver, int data, int
         receivingPacketsNumber) { ... }
}

reactiveclass WirelessMedium(5) {
    statevars {
        CommunicationDevice senderDevice;
        CommunicationDevice receiverDevice;
        int maxTraffic;
    }
    WirelessMedium() {
        senderDevice = null;
        receiverDevice = null;
        maxTraffic = (125 * 1024) / 8;
    }
    msgsrv getStatus() { ... }
    msgsrv broadcast(CommunicationDevice receiver,
         int data, int packetsNumber) { ... }
    msgsrv broadcastingIsCompleted() {
        senderDevice = null;
        receiverDevice = null;
    }
}

main {
  WirelessMedium medium():();
    CPU cpu (sensorNodeSenderDevice, receiver,
         sensor):();
    Sensor sensor(cpu):();
    Misc misc(cpu):();
    CommunicationDevice
         sensorNodeSenderDevice(medium):((byte)1);
  CommunicationDevice receiver(medium):((byte)0);
}
```

Figure 9: The model of a WSAN application.

Configuration	State Space Generation				Model Checking Time		
	states ORG	states RED	Gain	Time	ORG, OLD	ORG, NEW	RED, NEW
25-5-3-10	1,741	402	77%	<1s	<1s	<1s	<1s
33-6-4-2	1,934	451	77%	<1s	<1s	<1s	<1s
25-5-4-10	3,718	945	75%	1s	<1s	<1s	<1s
30-6-4-2	9,353	2,774	71%	1s	<1s	<1s	<1s
25-6-4-2	34,503	10,368	70%	2s	<1s	<1s	<1s
20-6-4-2	57,621	17,714	69%	3s	<1s	<1s	<1s

Table 4: The size of the state spaces and the gained reductions in WSAN example with different configuration.

these experiments are depicted in Table 4. In each row, the configuration (the numbers which are separated by a dash) is a combination of the sampling rate, the number of nodes, the packet size, and the sensor task delay of the experiment, respectively. As shown in Table 4, the time consumption of the model checking is less than one second for all cases and changing the configuration of the model does not end in large state spaces. However, the effectiveness of the reduction technique is reduced in configurations which result in bigger state spaces. This is because of the fact that changing the configuration of WSAN in this way does not increase the number of messages which are sent at the same time. So, the chance of finding transient transitions is decreased as there is no increment in the number of simultaneously executing instantaneous transitions.

7. Related Work

The model checking against TCTL properties The decidability of model-checking TCTL has been shown by Alur, Courcoubetis, and Dill [40], using clock equivalence and region transition systems. They proposed the first model checking algorithm for timed automata against TCTL properties. A variation of this algorithm is used in many model checking algorithms, including UPPAAL [4]. To over come the inefficiencies of this model checking algorithm, some reduction techniques are proposed for verification of timed automata models, instead of proposing a new model checking algorithm. Bengtsson et al. in [41] proved that allowing clock of timed automata to increase independently and synchronize clocks when two timed automata want to communicate, is an effective way for improving the time consumption of the TCTL model checking. This idea is close to the inter-process atomic blocks of SPIN. This work is continued by Minea in [42] by applying the proposed reduction technique in model checking of timed extension of LTL. In comparison to these works, our algorithm performs model checking in polynomial; however, it supports a limited subsystem of the real-time systems, i.e. discrete time systems.

In another attempt, Campos et al. in [8] addressed discrete-time systems and introduced timed transition graph (TTG) as the underlying semantics of this type of systems. TTGs are transition graphs in which an interval is associated with each transition. Passing such a transition results in progress in time with a value which is chosen nondeterministically from the associated interval. They proposed a polynomial time symbolic model checking algorithm for TTGs in [8]. Later, Laroussinie et al. in [43] addressed a subset of timed automata which can be model checked easier. They gave a polynomial time algorithm for the model checking of $\text{TCTL}_{\leqslant,\geqslant}$ over the class of timed automata with one or two clocks. They showed that the model checking of full TCTL over one clock timed automata is PSPACE-complete. In comparison to the work of this paper, our TCTL model checking algorithm outperforms all of the aforementioned works regarding to the algorithm complexity point of view, without need of any limitation on the number or types of clocks.

Model checking of timed actors As one of the earliest attempts for model checking timed actors, a tool is developed for model checking of Timed Rebeca models using a transformation from Timed Rebeca to timed automata. The resulting timed automata are model checked against TCTL properties using the UPPAAL toolset. Using this transformation, the most efficient network of timed automata is generated for Timed Rebeca models (having as much as possible committed states and as few as possible number of clocks). But, because of the inefficiency of modeling asynchronous communication among actors by synchronized communication of timed automata, model checking results in state space explosion even for middle-sized case studies [44]. A similar approach of transforming timed actor models into timed automata is taken by de Boer et al. in [15], where timed actor models in Creol language are analyzed for schedulability. This work also suffers from a lack of scalability for the same reason.

In other work, Floating Time Transition System (FTTS) is introduced as a natural semantics of timed actors in [31]. Focusing on the analysis of timed actors based on the key features of actors, being event-driven and isolated, results in a significant amount of state space reduction in FTTSs. Actors in a state of a FTTS can be in different local times, so, there is no unique value for the time of a state. Such time skew among actors is only admissible where we are not interested in the state of all the actors at a specific point of time, e.g. checking for deadlock freedom and schedulability, or any other event-based property. As a result, although FTTS works efficiently for deadlock freedom and schedulability analysis of timed actors, it cannot be used for the model checking of timed actors against TCTL properties.

Another work on model checking of timed actors is based on mapping timed actors to Real-Time Maude. This enables a formal model-based methodology which combines the convenience of intuitive modeling in timed actors with formal verification of Real-Time Maude. Real-Time Maude is supported by a high-performance toolset providing a spectrum of analysis methods, including simulation through timed rewriting, reachability analysis, and (untimed) linear temporal logic (LTL) model checking as well as timed CTL model checking. As described in [45], all the possible reduction techniques are applied to the generated Real-Time Maude models to avoid state space explosion. Mainly, a number of statements (which are related to the instantaneous statements of Timed Rebeca except sending messages) are grouped together to be executed in one atomic rewrite step. The experimental results, reported in [45], show that the generate state spaces using Real-time Maude is significantly bigger than the state spaces which are generated by the fine-grained semantics of Timed Rebeca. So, although Real-time Maude provides us with a wide range of analysis tools, using transition systems which are generated based on the fine-grained semantics together with the algorithm of this paper outperforms it.

There is also an analysis toolset for simulating Timed Rebeca models. In [46], the simulation engine of Erlang [47] is used to generate a number of traces and verify them. Using this approach, state space explosion is avoided; however, it does not guarantee the correctness of models.

8. Summary and Conclusion

In this paper, we proposed techniques for improving the model checking of discrete time actors. At the first step, we introduced a new model checking algorithm, which is an optimal $\mathrm{TCTL}_{\leqslant,\geqslant}$ model checking algorithm for discrete time actors with dense transition systems. So, discrete time actors can be model checked faster than before. In addition to this improvement, we have proposed a reduction technique which works based on the fact that the instantaneous transitions take no time to execute; so, the system cannot stay in the states whose outgoing transitions are all instantaneous. These states are not observable to the verifier so they can be eliminated from the transition systems. Beside reducing the size of the transition system, applying the reduction technique enables efficient $\mathrm{TCTL}_{=}$ model checking of timed actors. Formerly, timed actor models had to be transformed to Real-Time Maude for TCTL model checking. However, using the work presented in this paper, we apply the new efficient model checking algorithm on a reduced transition system, which outperforms time and memory consumption of TCTL model checking in comparison to the previously proposed algorithm. This way, model checking of bigger transition systems is possible. Although we have used Timed Rebeca to illustrate the techniques presented in this paper, our results are not limited to this language, and can be applied to any modeling formalism with a discrete notion of time.

Experimental evidence supports our theoretical observation that the new model checking algorithm works efficiently and the reduction technique results in smaller transition systems in general. In the case of models with many concurrently executing actors, the time consumption of the model checking increased rapidly for the old algorithm; however, using the new algorithm avoids it. Although the new algorithm works more efficient in comparison with the old one, its time consumption is high for big transition system. For these cases, using the FTS reduction technique results in up to 95% reduction in the size of the transition systems.

Therefore, we can efficiently model check more complicated models against complete TCTL properties under certain conditions.

The work reported in this paper paves the way to several interesting avenues for the future works. In particular, we have already started defining a special kind of DTG for the continuous time which conforms the requirements of dense time actors and can be model checked in polynomial time, using the same algorithm. It is also possible to work on the categorization of TCTL properties to illustrate that which category of TCTL properties benefits more from the provided efficiency of the proposed algorithm.

References

[1] T. A. Henzinger, Z. Manna, A. Pnueli, Timed transition systems, in: J. W. de Bakker, C. Huizing, W. P. de Roever, G. Rozenberg (Eds.), Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings, Vol. 600 of Lecture Notes in Computer Science, Springer, 1991, pp. 226–251. doi:10.1007/BFb0031995.
URL http://dx.doi.org/10.1007/BFb0031995

[2] R. Alur, D. L. Dill, A Theory of Timed Automata, Theoretical Computer Science 126 (2) (1994) 183–235.

[3] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.

[4] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems, in: R. Alur, T. A. Henzinger, E. D. Sontag (Eds.), Hybrid Systems, Vol. 1066 of Lecture Notes in Computer Science, Springer, 1995, pp. 232–243.

[5] E. A. Emerson, A. K. Mok, A. P. Sistla, J. Srinivasan, Quantitative temporal reasoning., Vol. 4, 1992, pp. 331–352.

[6] S. V. A. Campos, E. M. Clarke, W. R. Marrero, M. Minea, H. Hiraishi, Computing quantitative characteristics of finite-state real-time systems, in: RTSS, IEEE Computer Society, 1994, pp. 266–270.

[7] F. Laroussinie, P. Schnoebelen, M. Turuani, On the expressivity and complexity of quantitative branching-time temporal logics, Theoretical Computer Science 297 (1-3) (2003) 297–315.

[8] S. V. Campos, E. M. Clarke, Theories and experiences for real-time system development, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1994, Ch. Real-time Symbolic Model Checking for Discrete Time Models, pp. 129–145.
URL http://dl.acm.org/citation.cfm?id=207907.207912

[9] F. Laroussinie, N. Markey, P. Schnoebelen, Efficient timed model checking for discrete-time systems, Theoretical Computer Science 353 (1-3) (2006) 249–271.

[10] C. Hewitt, Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT (Apr. 1972).

[11] G. Agha, C. Hewitt, Actors: A conceptual foundation for concurrent object-oriented programming, in: Research Directions in Object-Oriented Programming, 1987, pp. 49–74.

[12] G. A. Agha, ACTORS - a model of concurrent computation in distributed systems, MIT Press series in artificial intelligence, MIT Press, 1990.

[13] I. A. Mason, C. L. Talcott, Actor languages their syntax, semantics, translation, and equivalence, Theor. Comput. Sci. 220 (2) (1999) 409–467. doi:10.1016/S0304-3975(99)00009-2.
URL http://dx.doi.org/10.1016/S0304-3975(99)00009-2

[14] S. Ren, G. Agha, RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems, in: R. Gerber, T. J. Marlowe (Eds.), Workshop on Languages, Compilers, & Tools for Real-Time Systems, ACM, 1995, pp. 50–59.

[15] F. S. de Boer, T. Chothia, M. M. Jaghoori, Modular schedulability analysis of concurrent objects in creol, in: F. Arbab, M. Sirjani (Eds.), Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers, Vol. 5961 of Lecture Notes in Computer Science, Springer, 2009, pp. 212–227. doi:10.1007/978-3-642-11623-0_12.
URL http://dx.doi.org/10.1007/978-3-642-11623-0_12

[16] A. H. Reynisson, M. Sirjani, L. Aceto, M. Cimini, A. Jafari, A. Ingólfsdóttir, S. H. Sigurdarson, Modelling and simulation of asynchronous real-time systems using timed rebeca, Sci. Comput. Program. 89 (2014) 41–68. doi:10.1016/j.scico.2014.01.008.
URL http://dx.doi.org/10.1016/j.scico.2014.01.008

[17] M. Geilen, S. Tripakis, M. Wiggers, The earlier the better: a theory of timed actor interfaces, in: M. Caccamo, E. Frazzoli, R. Grosu (Eds.), HSCC, ACM, 2011, pp. 23–32.

[18] F. S. de Boer, M. M. Jaghoori, C. Laneve, G. Zavattaro, Decidability problems for actor systems, in: M. Koutny, I. Ulidowski (Eds.), CONCUR, Vol. 7454 of Lecture Notes in Computer Science, Springer, 2012, pp. 562–577.

[19] L. Aceto, M. Cimini, A. Ingólfsdóttir, A. H. Reynisson, S. H. Sigurdarson, M. Sirjani, Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca, in: M. R. Mousavi, A. Ravara (Eds.), FOCLASA, Vol. 58 of EPTCS, 2011, pp. 1–19.

[20] M. M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, A. Movaghar, Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca, Acta Inf. 47 (1) (2010) 33–66.

[21] M. Sirjani, A. Movaghar, A. Shali, F. S. de Boer, Model checking, automated abstraction, and compositional verification of rebeca models, J. UCS 11 (6) (2005) 1054–1082. doi:10.3217/jucs-011-06-1054.
URL http://dx.doi.org/10.3217/jucs-011-06-1054

[22] Rebeca Home Page, http://www.rebeca-lang.org.

[23] E. Khamespanah, R. Khosravi, M. Sirjani, Efficient TCTL model checking algorithm for timed actors, in: E. G. Boix, P. Haller, A. Ricci, C. Varela (Eds.), Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014, ACM, 2014, pp. 55–66. `doi:10.1145/2687357.2687366`.
URL `http://doi.acm.org/10.1145/2687357.2687366`

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms (3. ed.), MIT Press, 2009.
URL `http://mitpress.mit.edu/books/introduction-algorithms`

[25] M. Nykänen, E. Ukkonen, The exact path length problem, J. Algorithms 42 (1) (2002) 41–53.

[26] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.

[27] M. Sirjani, A. Movaghar, A. Shali, F. S. de Boer, Modeling and Verification of Reactive Systems using Rebeca, Fundam. Inform. 63 (4) (2004) 385–410.

[28] M. Sirjani, M. M. Jaghoori, Ten Years of Analyzing Actors: Rebeca Experience, in: G. Agha, O. Danvy, J. Meseguer (Eds.), Formal Modeling: Actors, Open Systems, Biological Systems, Vol. 7000 of Lecture Notes in Computer Science, Springer, 2011, pp. 20–56.

[29] M. Sirjani, F. S. de Boer, A. Movaghar-Rahimabadi, Modular verification of a component-based actor language, J. UCS 11 (10) (2005) 1695–1717. `doi:10.3217/jucs-011-10-1695`.
URL `http://dx.doi.org/10.3217/jucs-011-10-1695`

[30] E. Khamespanah, M. Sirjani, M. Viswanathan, R. Khosravi, Floating Time Transition System: More Efficient Analysis of Timed Actors, in: C. Braga, P. C. Ölveczky (Eds.), Formal Aspects of Component Software - 12th International Symposium, FACS 2015, Rio de Janeiro, Brazil, October 14-16, 2015, Lecture Notes in Computer Science, Springer, 2016.

[31] E. Khamespanah, M. Sirjani, Z. Sabahi-Kaviani, R. Khosravi, M. Izadi, Timed rebeca schedulability and deadlock freedom analysis using bounded floating time transition system, Sci. Comput. Program. 98 (2015) 184–204. `doi:10.1016/j.scico.2014.07.005`.
URL `http://dx.doi.org/10.1016/j.scico.2014.07.005`

[32] A. Sheibanyrad, A. Greiner, I. M. Panades, Multisynchronous and fully asynchronous nocs for GALS architectures, IEEE Design & Test of Computers 25 (6) (2008) 572–580. `doi:10.1109/MDT.2008.167`.
URL `http://dx.doi.org/10.1109/MDT.2008.167`

[33] Z. Sharifi, M. Mosaffa, S. Mohammadi, M. Sirjani, Functional and performance analysis of network-on-chips using actor-based modeling and formal verification, Vol. 66, 2013.

[34] Apache Hadoop Home Page, http://hadoop.apache.org.

[35] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. `doi:10.1145/1327452.1327492`.
URL `http://doi.acm.org/10.1145/1327452.1327492`

[36] T. White, Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated), O'Reilly, 2012.
URL `http://www.oreilly.de/catalog/9781449311520/index.html`

[37] L. E. Linderman, K. Mechitov, B. F. Spencer, TinyOS-based Real-Time Wireless Data Acquisition Framework for Structural Health Monitoring and Control, Vol. 20, 2013, p. 10071020. `doi:http://dx.doi.org/10.1002/stc.1514`.

[38] L. Nachman, R. Kling, R. Adler, J. Huang, V. Hummel, The intel mote platform: a bluetooth-based sensor network for industrial monitoring, in: Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks, IPSN 2005, April 25-27, 2005, UCLA, Los Angeles, California, USA, IEEE, 2005, pp. 437–442. `doi:10.1109/IPSN.2005.1440968`.
URL `http://dx.doi.org/10.1109/IPSN.2005.1440968`

[39] E. Khamespanah, K. Mechitov, M. Sirjani, G. A. Agha, Schedulability analysis of distributed real-time sensor network applications using actor-based model checking, in: D. Bosnacki, A. Wijs (Eds.), Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings, Vol. 9641 of Lecture Notes in Computer Science, Springer, 2016, pp. 165–181. `doi:10.1007/978-3-319-32582-8_11`.
URL `http://dx.doi.org/10.1007/978-3-319-32582-8_11`

[40] R. Alur, C. Courcoubetis, D. L. Dill, Model-checking in dense real-time, Inf. Comput. 104 (1) (1993) 2–34.

[41] J. Bengtsson, B. Jonsson, J. Lilius, W. Yi, Partial order reductions for timed systems, in: D. Sangiorgi, R. de Simone (Eds.), CONCUR, Vol. 1466 of Lecture Notes in Computer Science, Springer, 1998, pp. 485–500.

[42] M. Minea, Partial order reduction for model checking of timed automata, in: J. C. M. Baeten, S. Mauw (Eds.), CONCUR, Vol. 1664 of Lecture Notes in Computer Science, Springer, 1999, pp. 431–446.

[43] F. Laroussinie, N. Markey, P. Schnoebelen, Model checking timed automata with one or two clocks, in: P. Gardner, N. Yoshida (Eds.), CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings, Vol. 3170 of Lecture Notes in Computer Science, Springer, 2004, pp. 387–401. `doi:10.1007/978-3-540-28644-8_25`.
URL `http://dx.doi.org/10.1007/978-3-540-28644-8_25`

[44] E. Khamespanah, Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, M.-J. Izadi, Timed-rebeca schedulability and deadlock-freedom analysis using floating-time transition system, in: G. A. Agha, R. H. Bordini, A. Marron, A. Ricci (Eds.), AGERE!@SPLASH, ACM, 2012, pp. 23–34.

[45] Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, P. C. Ölveczky, E. Khamespanah, Formal semantics and analysis of timed rebeca in real-time maude, in: C. Artho, P. C. Ölveczky (Eds.), FTSCS, Vol. 419 of Communications in Computer and Information Science, Springer, 2013, pp. 178–194.

[46] B. Magnusson, Simulation-Based Analysis of Timed Rebeca Using TeProp and SQL, Master's thesis, Reykjavk University, School of Computer Science, Iceland, http://rebeca.cs.ru.is/files/MasterThesisBrynjarMagnusson2012.pdf (2012).

[47] C. B. Earle, L. Fredlund, Verification of Timed Erlang Programs Using McErlang, in: Proceedings of the 14th joint IFIP WG 6.1 international conference and Proceedings of the 32nd IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems, FMOODS'12/FORTE'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 251–267.