Modeling, Verification, and Analysis of Timed Actor-Based Models

Ehsan Khamespanah

Doctor of Philosophy

June 2018

School of Computer Science

Reykjavík University

Ph.D. Dissertation

Modeling, Verification, and Analysis of Timed Actor-Based Models

by

Ehsan Khamespanah

Dissertation submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

June 2018

Thomas A. Henzinger, Examiner
Professor, Institute of Science and Technology, Austria

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this Dissertation entitled **Modeling, Verification, and Analysis of Timed Actor-Based Models** submitted by **Ehsan Khamespanah** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy (Ph.D.) in Computer Science**

....

date

....

Marjan Sirjani, Supervisor
Professor, Reykjavík University, Iceland
Professor, Mälardalen University, Sweden

....

Ramtin Khosravi, Supervisor
Assistant Professor, University of Tehran, Iran

....

Edward A. Lee,
Professor, University of California at Berkeley, USA

....

Marcel Kyas,
Assistant Professor, Reykjavík University, Iceland

....

Fatemeh Ghassemi,
Assistant Professor, University of Tehran, Iran

....

Thomas A. Henzinger, Examiner
Professor, Institute of Science and Technology, Austria

Modeling, Verification, and Analysis of Timed Actor-Based Models

Ehsan Khamespanah

June 2018

Abstract

In the recent years, formal modeling and verification of realtime systems have become very important. Difficult-to-use modeling languages and inefficient analysis tools are the main obstacles to use formal methods in this domain. Timed actor model is one of the modeling paradigms which is proposed for modeling of realtime systems. It benefits from high-level object-oriented modeling facilities; however, developed analysis techniques for timed actors needs to be improved to make the actor model acceptable for the analysis of real-world applications.

In this thesis, we first tackle the model checking problem of timed actors by proposing the standard semantics of timed actors in terms of fine-grained timed transition system (FGTS) and transforming it to Durational Transition Graph (DTG). This way, while the time complexity of model checking algorithms for TCTL properties in general is non-polynomial, we are able to check $\text{TCTL}_{\leq,\geq}$ properties (a subset of TCTL) using model checking in polynomial time. We also improve the model checking algorithm of $\text{TCTL}_{\leq,\geq}$ properties, obtaining time complexity of $O((V \lg V + E) \cdot |\Phi|)$ instead of $O(V(V + E) \cdot |\Phi|)$ and use it for efficient model checking of timed actors. In addition, we propose a reduction technique which safely eliminates instantaneous transitions of FGTS. Using the proposed reduction technique, we provide an efficient algorithm for model checking of complete TCTL properties over the reduced transition systems.

In actor-based models, the absence of shared variables and the presence of single-threaded actors along with non-preemptive execution of each message server, ensure that the execution of message servers do not interfere with each other. Based on this observation, we propose Floating Time Transition System (FTTS) as the big-step semantics of timed actors. The big-step semantics exploits actor features for relaxing the synchronization of progress of time among actors, and thereby reducing the number of states in transition systems. Considering an actor-based language, we prove there is an action-based weak bisimulation relation between FTTS and FGTS. As a result, the big-step semantics preserves event-based branching-time properties.

Finally, we show how Timed Rebeca and FTTS are used as the back-end analysis technique of three different independent works to illustrate the applicability of FTTS in practice.

I dedicate this to whom who takes care of us,
my parents, my sisters and brother.

Acknowledgements

Thank you all. This, at the highest level of abstraction, shows my thanks and gratitude to all who made it possible for me to accomplish my PhD. I would like to thank my PhD supervisors Marjan Sirjani and Ramtin Khosravi. Throughout these years, they were sources of inspirations for me and they were more of friends to me than supervisors.

I spent most of my days at Reykjavik University and Tehran University with these people, in addition to Marjan and Ramtin, alphabetically listed: Luca Acceto, Fatemeh Ghassemi, Mohammad-Javad Izadi, Ali Jafari, Mina Mohammadkhani, Mahdi Mosaffa, Mohammadreza Mousavi, Hamidreza Pourvatan, Zeynab Sabahi, Nazanin-Sadat Seyyed-Razi, Sadegh Shafiei, and Zeynab Sharifi. We sometimes had scientific discussions, but I am to thank them mainly because they made my life easy being good friends before anything else.

I traveled a lot, for short visits and presenting papers, all around the world during which I learned scientific collaboration, presentation, discussion, and sightseeing. During all these meetings, I also got to know many other nice people related to the domain of formal modeling and analysis of systems. They all played a role in my way towards this thesis. As the smallest sign of appreciation, I will name them, alphabetically: Gul Agha, Dragan Bosnacki, Holger Hermanns, Joost-Pieter Katoen, Kim Guldstrand Larsen, Kirill Mechitov, and Mahesh Viswanathan. I should thank everyone in Reykjavik University who helped me in different ways, especially Björn Þór Jónsson, Yngvi Björnsson, and Sigrún María Ammendrup. I wish all those who helped and supported me scientifically, and more importantly non-scientifically, the best and success and prosperity in life and afterward.

The work on this dissertation was supported by the project “Self-Adaptive Actors: SEADA” (nr. 163205-051) of the Icelandic Research Fund.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modeling is crucial, both in science and engineering. We build models to be able to do analysis without having to deal with the details of a system's implementation. Edward Lee [1] emphasizes on the difference between engineers and scientists when they build and use a model. Engineers build a model to explore the design space and construct a system based on the model; and scientists build a model of an existing system to be able to analyze it. So, engineers do their best to build the system just like the model, and scientists do their best to build the model similar to the existing system. No matter whether we use a model as an engineer or a scientist, we need to have a faithful model in order to perform a valid analysis and/or design exploration [2].

Besides difficulties of developing faithful models, proposing analysis techniques for these models is a crucial issue. The non-functional properties of different natures are becoming more crucial in correctness of a software system, demanding new models and/or extensions of existing languages. Timing features are no more just performance concerns. In many software systems, nowadays, timing features are part of correctness properties. Ensuring the correctness of these types of properties for realtime systems turned into a very difficult task. Realtime systems are hardware and software systems that are subject to realtime constraints. Such systems are usually used in critical applications. Realtime computations can be said to have failed if they are not completed before their deadline. Therefore, when verifying a model of a realtime system, in addition to its functional correctness, it is necessary to prove that the model meets the specified realtime criteria.

The importance of formal modeling and analysis for ensuring the dependability and correctness of realtime systems has long been acknowledged. However, the lack of a total solution containing an easy to use modeling language and efficient analysis technique has limited the use of formal methods. Timed Rebeca [3] is designed to be a usable and formally analyzable modeling language for realtime systems as a response to the abovementioned requirements. The target application domain of Timed Rebeca is event-driven systems with asynchronous message passing. In this thesis, we present a set of analysis techniques for model checking of Timed Rebeca models. Prior to this work, some other analysis techniques had been proposed (including [4], [5], and [3]) for model checking and analysis of Timed Rebeca models. These approaches had been developed based on transformation from Timed Rebeca models to other models. This way, the analysis toolset of the target models can be used for the analysis of Timed Rebeca models. In comparison to those works, in this thesis we propose direct approaches for model checking of Timed Actor models which results in more efficient analysis of the models. This way, more complicated real-world systems can be modeled and analyzed by Timed Rebeca more efficiently.

1.1 Timed Rebeca as a Modeling Language

We may hear the following question, mostly in more theoretical communities: "why yet another modeling language?" This question is usually asked if you mainly focus on the expressiveness of the modeling languages. But usability and fidelity are also two crucial features of a modeling language, and their importance is very well acknowledged from a more practical point of view. Models need to be able to capture the characteristics of the system which affect the properties of our interest (fidelity), and we need to be able to understand and build a model with the least possible effort (usability). For example, object-oriented approaches were introduced with the philosophy of reducing the semantic gap between the real world problems and the models representing those problems; and their success is undeniable. With the growing need for various software applications, and fast changes in hardware and network infrastructures, the answer to the above question is simple: because we are not there yet. And with "change" being the only constant in our software world, we will possibly never be there [2].

The modeling language Rebeca (*Reactive Objects Language*) [6], [7], is an operational interpretation of the actor model [8], [9] provided with formal semantics and supported by model checking tools [10]. Rebeca is designed to be a usable and analyzable modeling language to bridge the gap between software engineers and the formal methods community. The application domain targeted by Rebeca is where we have event-driven systems, with asynchronous message passing. In Rebeca, we have non-blocking sends, no explicit receive, no shared variables, and non-preemptive method execution.

Timed Rebeca is an extension of Rebeca with time features for modeling and verification of realtime systems. Different approaches have been proposed for modeling and analysis of realtime systems. Timed automata [11], realtime Maude [12], and TCCS [13] are examples of modeling formalisms for design and analysis of realtime systems. Apart from these well-known and general purpose modeling formalisms, high level modeling languages are adopted for the realtime requirements. The actor model as an example of such languages is extended with timing features to address the functional behaviors of actors and the timing constraints on patterns of actor invocations. A realtime actor model, RT-synchronizer, is proposed in [14] as an example of an actor model that enforces realtime relations between events. While RT-synchronizer is an abstraction mechanism for the declarative specification of timing constraints over groups of actors, Timed Rebeca allows us to work at a lower level of abstraction. Using Timed Rebeca, a modeler can easily capture the functional features of a system, together with the timing constraints for both computation and network latencies, and analyze the model from various points of view.

Creol [15] is a concurrent object based language which is designed in parallel with Rebeca. Concurrent objects of Creol can be checked for schedulability using the approach of [15], which is developed based on the same idea presented for Timed Rebeca in [16]. ABS [17] is an extension of Creol in multiple ways. While in Creol and its descendent, ABS, the focus has been on different modeling features, for Rebeca the core of the language is kept simple, avoiding adding any complexity. The focus in Timed Rebeca has been on analysis and formal verification of models.

1.2 Previous Analysis Techniques of Timed Rebeca Models

As one of the earliest attempts for model checking of Timed Rebeca models, a toolset is developed for model checking of Timed Rebeca models using transformation from Timed Rebeca models to networks of 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 many 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 [18]. 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.

Another work on model checking of Timed Rebeca is based on mapping timed actors to Realtime Maude. This enables a formal model-based methodology which combines the convenience of intuitive modeling in timed actors with formal verification of Realtime Maude. Realtime 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 [19], all the possible reduction techniques are applied to the generated Realtime 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 [19], show that the generated state spaces using Realtime Maude are significantly bigger than the state spaces which are generated by the fine-grained semantics of Timed Rebeca.

Translating Timed Rebeca to Erlang [20], [21] has been presented in [22]. The motivation for translating Timed-Rebeca models to Erlang code is to be able to use McErlang [23], [24] to model check them. McErlang has full Erlang data type support, support for general process communication, node semantics, fault detection and other Erlang features which have being used by most modern Erlang programs [23]. The McErlang tool set supporting verification methods ranges from state-based exploration to simulation, with property specifications written as LTL formulas or any other hand-coded piece of code for system runtime verification.

None of the aforementioned approaches did not provide efficient enough model checking facilities which can support the analysis of real-world case studies.

1.3 Thesis Overview and Contributions

The main contributions of this work are on developing more efficient analysis techniques which enable modelers to analyze more complex Timed Rebeca models and real-world case studies. To this end, we presented the standard semantics of Timed Rebeca and developed a tool for directly generating the state spaces of Timed Rebeca models. This semantics is presented in Chapter 2.2. The contributions of this part are the following.

- Proposing the standard semantics of Timed Rebeca in terms of timed transition system with discrete progress of time steps, called Fine-Grained Transition System (FGTS),
- Evaluating the size of transition systems which are generated based on the standard semantics.

At the next step, we propose a new $\text{TCTL}_{\leq\geq}$ model checking algorithm and show that it can be used for the analysis of Timed Rebeca models. This algorithm improves the time complexity of the previously proposed algorithm for the model checking of discrete-time systems, as described in Chapter 3. The contributions of this part are the following.

- Proposing a new $\text{TCTL}_{\leq\geq}$ model checking algorithm with a better time complexity and execution time,
- Proving the fact that the new algorithm can be used for model checking of Timed Rebeca models,
- Evaluating how efficient is the improved algorithm based on the time complexity analysis and real-world execution time.

We also show that ignoring transient states of a model results in a huge reduction in the size of state spaces. This technique, called folding the instantaneous transitions and as shown in Chapter 4, reduces the size of state spaces significantly. The contributions of this part are the following.

- Introducing a reduction technique by folding the instantaneous transitions in object-based modeling languages,
- Making clear in which condition this reduction technique works for Timed Rebeca models,
- Proving the fact that applying this reduction technique results in the ability of complete TCTL model checking of Timed Rebeca models in polynomial time.

To have even smaller state spaces for Timed Rebeca models, a big-step semantics is proposed which results in the generation of Floating Time Transition System (Chapter 5). Using the big-step semantics, the local times of actors in a state can be different, and there is no unique value for time in each state. We prove that this semantics preserves the result of model checking against timed property of actions. Examples of such properties include μ-calculus with weak modalities. The contributions of this part are the following.

- Introducing the notion of Floating Time Transition System,
- Proving the existence of weak-bisimulation relation between the transition systems generated by the big-step semantics and the standard semantics, results in have the same result for model checking against μ-calculus properties with weak modalities in Floating Time Transition Systems and fine-grained transition systems,
- Evaluating how small are the transition systems which are generated based on the big-step semantics in comparison with their corresponding FGTSs.

We extend Afra, the model checking tool of Rebeca, to support the algorithms and techniques of this thesis. To illustrate the applicability of this extension, we use Afra for the analysis of a set of real-world case studies. As presented in Chapter 6, we demonstrate three different case studies which we develop; one of them as a contribution of this thesis and the other two case studies are developed by others as parts of two masters theses. The contributions of this part are the following.

- Enriching Afra to support model checking of Timed Rebeca models, including 1) generating transition systems based on the standard and big-step semantics, 2) folding instantaneous transition, 3) the newly proposed TCTL model checking algorithm,
- Developing a model for WSAN applications and analyze it using Afra.

Chapter 2

Timed Rebeca

2.1 Introduction to Timed Rebeca

Timed Rebeca is an extension on Rebeca with time features for modeling and verification of time-critical systems. Rebeca [6], [10] is an actor-based language, for modeling concurrent and reactive systems with asynchronous message passing. The actor model was originally introduced by Hewitt [8] as an agent-based language, and is a mathematical model of concurrent computation that treats *actors* as the universal primitives of concurrent computation [25]. A Rebeca model is similar to the actor model as it has reactive objects with no shared variables, asynchronous message passing with no blocking send and no explicit receive, and unbounded buffers for messages. Objects in Rebeca are reactive, self-contained, and each of them is called a *rebec* (reactive object). Note that in this thesis we use rebec and actor interchangeably. Communication takes place by message passing among actors. Each actor has an unbounded buffer, called message *queue*, for its arriving messages. Computation is event-driven, meaning that each actor takes a message that can be considered as an event from the top of its message queue and executes the corresponding message server (also called a method). The execution of a message server is atomic which means that there is no way to preempt the execution of a message server of an actor and start executing another message server of that actor.

A Rebeca model consists of a set of reactive classes and the main block (for the syntax of Rebeca see Figure 2.1 and for an example see Figure 2.2). In the main block, actors which are instances of the reactive classes are declared. The body of the reactive class includes the declaration of its known rebecs, state variables, and message servers. Message servers consist of the declaration of local variables and the body of the message server. The statements in the body can be assignments, conditional statements, enumerated loops, non-deterministic assignment, and method calls. Method calls are sending asynchronous messages to other actors (or to itself). A reactive class has an argument of type integer denoting the maximum size of its message queue. Although message queues are unbounded in the semantics of Rebeca, to ensure that the state space is finite, we need a user-specified upper bound for the queue size. The operational semantics of Rebeca has been introduced in [6] in more detail. In comparison with the standard actor model, dynamic creation and dynamic topology are not supported by Rebeca. Also, actors in Rebeca are single-threaded.

We illustrate the Rebeca language with an example. Listing 2.1 shows the Rebeca model of the ticket service system of Figure 2.2. The model consists of three reactive classes: *TicketService*, *Agent*, and *Customer*. *Customer* sends the *requestTicket*

$$
\begin{aligned}
\mathit{Model} &::= \mathit{Class}^{*}\ \mathit{Main}\\
\mathit{Main} &::= \textbf{main}\ \{\ \mathit{InstanceDcl}^{*}\ \}\\
\mathit{InstanceDcl} &::= \mathit{className}\ \mathit{rebecName}(\langle \mathit{rebecName}\rangle^{*}) : (\langle \mathit{literal}\rangle^{*});\\
\mathit{Class} &::= \textbf{reactiveclass}\ \mathit{className}\ \{\ \mathit{KnownRebecs}\ \mathit{Vars}\\
&\qquad \mathit{MsgSrv}^{*}\ \mathit{LocalMethods}^{*}\ \}\\
\mathit{KnownRebecs} &::= \textbf{knownrebecs}\ \{\ \mathit{RebecDcl}^{*}\ \}\\
\mathit{Vars} &::= \textbf{statevars}\ \{\ \mathit{VarDcl}^{*}\ \}\\
\mathit{RebecDcl} &::= \mathit{className}\ \langle v\rangle^{+};\\
\mathit{VarDcl} &::= \mathit{Type}\ \langle v\rangle^{+};\ \mid \mathit{Type}\ [\ \mathit{number}\ ]^{+}\ v\\
\mathit{MsgSrv} &::= \textbf{msgsrv}\ \mathit{msgName}(\langle \mathit{ExtType}\ v\rangle^{*})\ \{\ \mathit{Stmt}^{*}\ \}\\
\mathit{LocalMethods} &::= \mathit{methodName}(\langle \mathit{ExtType}\ v\rangle^{*})\ \{\ \mathit{Stmt}^{*}\ \}\\
\mathit{Stmt} &::= \mathit{Assignment} \mid \mathit{SendMessage} \mid \mathit{MethodCall} \mid\\
&\qquad \mathit{ConditionalStmt} \mid \mathit{LoopStmt} \mid \mathit{LocalVars}\\
\mathit{Assignment} &::= v = \mathit{Exp};\ \mid v =?(\mathit{Exp}\langle, \mathit{Exp}\rangle^{+});\\
\mathit{SendMessage} &::= \mathit{rebecExp}.\mathit{msgName}(\langle \mathit{Exp}\rangle^{*});\\
\mathit{MethodCall} &::= \mathit{methodName}(\langle \mathit{Exp}\rangle^{*});\\
\mathit{ConditionalStmt} &::= \mathit{if}\ (\mathit{Exp})\ \{\ \mathit{Stmt}^{*}\ \}\ [\mathit{else}\ \{\ \mathit{Stmt}^{*}\ \}]\\
\mathit{LoopStmt} &::= \mathit{for}\ (\ \mathit{Exp}\ ;\ \mathit{Exp}\ ;\ \mathit{Exp}\ )\ \{\ \mathit{Stmt}^{*}\ \} \mid \mathit{while}\ (\mathit{Exp})\ \{\ \mathit{Stmt}^{*}\ \}\\
\mathit{LocalVars} &::= \mathit{ExtType}\ \langle v\rangle^{+};\\
\mathit{Exp} &::= e \mid \mathit{rebecExpr}\\
\mathit{rebecExp} &::= \mathit{self} \mid \mathit{rebecTerm} \mid (\mathit{className})\mathit{rebecTerm}\\
\mathit{rebecTerm} &::= \mathit{rebecName} \mid \mathit{sender}\\
\mathit{ExtType} &::= \mathit{Type} \mid \mathit{float} \mid \mathit{double}\\
\mathit{Type} &::= \mathit{boolean} \mid \mathit{int} \mid \mathit{short} \mid \mathit{byte} \mid \mathit{className}
\end{aligned}
$$

Figure 2.1: Abstract syntax of Rebeca(a slightly revised version of the syntax presented in [22]). Angle brackets $\langle ... \rangle$ are used as meta parenthesis, superscript + for repetition at least once, superscript $*$ for repetition zero or more times, whereas using $\langle ... \rangle$ with repetition denotes a comma separated list. Brackets [...] indicates that the text within the brackets is optional. The symbol ? shows non-deterministic choice. Identifiers *className*, *rebecName*, *methodName*, v, *literal*, and *type* denote class name, rebec name, method name, variable, integer number, and type, respectively; and e denotes an (arithmetic, boolean or nondeterministic choice) expression. The parameter t is an expression with natural number result.

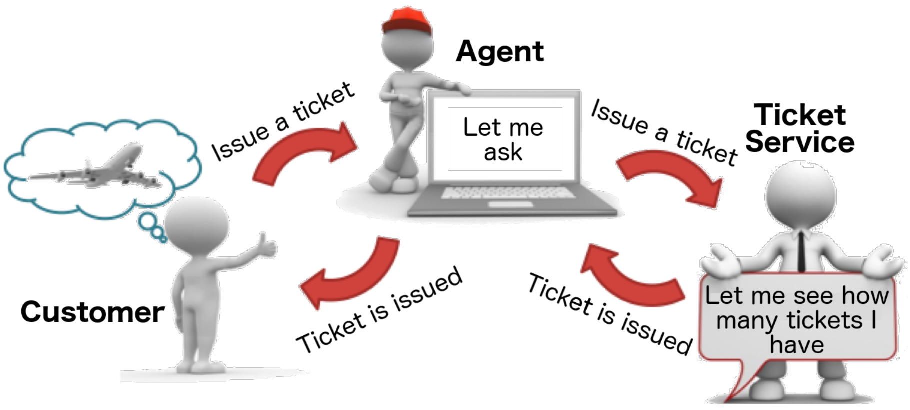


Figure 2.2: The actor model of Ticket Service System

message to *Agent* and *Agent* forwards the message to *TicketService*. *TicketService* replies to *Agent* by sending a *ticketIssued* message and *Agent* responds to *Customer* by sending the issued ticket.

Listing 2.1: The actor model of Ticket Service System

```
reactiveclass TicketService (3) {
    knownrebecs {Agent a;}
    statevars {
        int nextId;
    }
    TicketService() {
        nextId = 0;
    }
    msgsrv requestTicket() {
        a.ticketIssued(nextId);
        nextId = nextId + 1;
    }
}

reactiveclass Agent (3) {
    knownrebecs {
        TicketService ts;
        Customer c;
    }
    msgsrv requestTicket() {
        ts.requestTicket();
    }
    msgsrv ticketIssued(byte id) {
        c.ticketIssued(id);
    }
}

reactiveclass Customer (2) {
    knownrebecs {Agent a;}
    Customer() {
        self.try();
    }
    msgsrv try() {
        a.requestTicket();
    }
    msgsrv ticketIssued(byte id) {
        self.try();
    }
}
main {
    Agent a(ts, c):();
    TicketService ts(a):(3);
    Customer c(a):();
}
```

Timed Rebeca. Timed Rebeca [3] is an extension on Rebeca with time features for modeling and verification of time-critical systems. To this end, three primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events*. In a Timed Rebeca model, each actor has its own local clock and the local clocks evolve uniformly. Methods are still executed atomically, however passing time while executing a method can be modeled. In addition, instead of a queue for messages, there is a bag of messages for each actor.

The timing primitives that are added to the syntax of Rebeca are *delay*, *deadline* and *after*. The *delay* statement models the passing of time for an actor during execution of a message server. The keywords *after* and *deadline* can only be used in conjunction with a method call. The value of the argument of *after* shows how long it takes for the message to be delivered to its receiver. The *deadline* shows the timeout for the message,

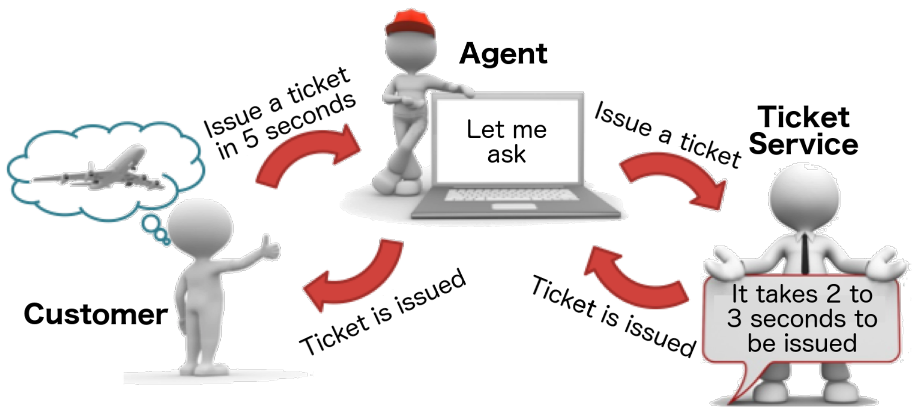


Figure 2.3: The actor model of Ticket Service System with time constraints

i.e., how long it will stay valid. We illustrate the application of these keywords with an example. Listing 2.2 shows the Timed Rebeca model of a ticket service system of Figure 2.3. As shown in line 11 of the model, issuing the ticket takes two or three time units (modeled by a non-deterministic expression). At line 23 the actor instantiated from *Agent* sends a message *requestTicket* to actor *ts* instantiated from *TicketService*, and gives a deadline of five to the receiver to take this message and start serving it. The periodic task of retrying for a new ticket is modeled in line 39 by the customer sending a *try* message to itself and letting the receiver to take it from its bag only after 30 units of time (by stating *after(30)*).

Listing 2.2: The Timed Rebeca model of ticket service system

```
 1 reactiveclass TicketService {
 2     knownrebecs {Agent a;}
 3     statevars {
 4         int issueDelay, nextId;
 5     }
 6     TicketService(int myDelay) {
 7         issueDelay = myDelay;
 8         nextId = 0;
 9     }
10     msgsrv requestTicket() {
11         delay(?(2, 3));
12         a.ticketIssued(nextId);
13         nextId = nextId + 1;
14     }
15 }
16
17 reactiveclass Agent {
18     knownrebecs {
19         TicketService ts;
20         Customer c;
21     }
22     msgsrv requestTicket() {
23         ts.requestTicket() deadline(5);
24     }
25     msgsrv ticketIssued(byte id) {
26         c.ticketIssued(id);
27     }
28 }
29
30 reactiveclass Customer {
31     knownrebecs {Agent a;}
32     Customer() {
33         self.try();
34     }
35     msgsrv try() {
36         a.requestTicket();
37     }
38     msgsrv ticketIssued(byte id) {
39         self.try() after(30);
40     }
41 }
42
43
44 main {
45     Agent a(ts, c):();
46     TicketService ts(a):(3);
47     Customer c(a):();
48 }
```

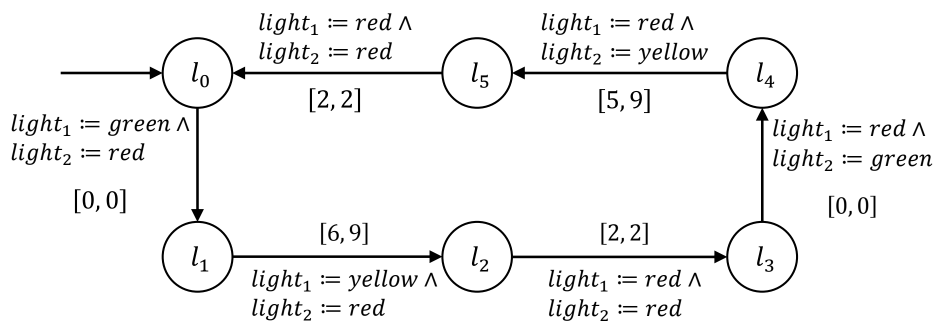


Figure 2.4: A TTS model of two traffic lights at a crossroad

2.2 Standard Semantics of Timed Rebeca[1]

The semantics of realtime systems is often defined assuming an ambient global time that proceeds uniformly for all participants in a distributed system. Even when individual local clocks are assumed to have skews, these skews are modeled relative to this ambient global time. Timed Transition System (TTS), as a basic computational model of realtime systems, generalizes the basic computation model of transition systems by associating an interval with each transition to indicate how long a transition takes [37]. In a TTS, transitions are partitioned into two classes: instantaneous transitions (in which time does not progress), and time ticks when the global clock is incremented. These time ticks happen when all participants "agree" for time elapse. TTS-based semantics is standard and has been defined for a variety of formalisms [30], [38]–[40]. Note that, using TTS is not limited to discrete-time systems. It also has been used to give a semantics for timed languages and formalisms that assume continuous or dense time domains.

Figure 2.4 illustrates how the behavior of a realtime 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 the 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.

In this chapter, we will show how the fine-grained semantics of Timed Rebeca can be directly defined in terms of TTS. To enable the formal description of the semantics of Timed Rebeca, we have to provide an abstract specification for the Syntax of Timed Rebeca models, conforming the details of Figure 2.1.

2.2.1 Abstract Syntax of Timed Rebeca

In the first step, we present the notations used in the rest of the article. Given a set A, the set A^* is the set of all finite sequences over elements of A, the set $\mathcal{P}(A)$ is the power set of A, and the set $\mathcal{P}_{\mathbb{N}}(A)$ is the power multiset of A. For a sequence $a \in A^*$ of length n, the symbol a_i denotes the i^{th} element of the sequence, where $1 \leq i \leq n$. Using this notation, we may also write the sequence a as $\langle a_1, a_2, \cdots, a_n \rangle$. The empty

[1]This chapter is an improvement and extension of the results published in [36] and [26].

sequence is represented by ϵ, and $\langle h|T\rangle$ denotes a sequence whose first element is $h \in A$ and $T \in A^*$ is the sequence comprising the elements in the rest of the sequence. For two sequences σ and σ' over A, the operator $\oplus$ is defined as $\oplus : A^* \times A^* \to A^*$ for the concatenation of two sequences such that $\sigma \oplus \sigma'$ is a sequence obtained by appending σ' to the end of σ.

A Timed Rebeca model consists of a number of reactive class declarations and a main block specifying actors which are instantiated from the reactive classes. A reactive class is defined as an instance of type $RClass = CID \times \mathcal{P}(Mtds) \times \mathcal{P}(Knowns) \times \mathcal{P}(Vars) \times \mathcal{P}(Mtds)$ such that:

- CID is the set of all reactive class identifiers in the model.
- $Mtds$ is the set of all method declarations.
- $Knowns$ is the set of all the identifiers of known actors.
- $Vars$ is the set of all variable names.

A reactive class $(cid, consts, knowns, vars, mtds)$ has the identifier cid, the constructor method $const$, the set of known actors $knowns$, the set of state variables $vars$, and the set of methods $mtds$. Each method (and the constructor method) is defined as the triple $(m, p, b) \in MName \times Var^* \times Stat^*$, where m is the name of the message the method is used to serve, p is the sequence of the names of the formal parameters, and b contains the sequence of statements comprising the body of the method.

The set of statements is defined as $Stat = Assign \cup Cond \cup Delay \cup Send \cup \{skip\}$, where different types of statements are defined as below. The meaning of the below statements is straightforward. $Expr$ denotes the set of integer expressions defined over usual arithmetic operators (with no side effects). $BExpr$ denotes the set of Boolean d defined over usual relational and logic operators. We do not dig into the details of the expressions here.

- $Assign = Var \times Expr$ is the set of assignment statements. We use the notation $var := expr$ as an alternative to $(var, expr)$.
- $Cond = BExpr \times Stat^* \times Stat^*$ is the set of conditional statements. We use the notation $if\, expr\, then\, \sigma\, else \sigma'$ as an alternative to $(expr, \sigma, \sigma')$.
- $Delay = Expr$ is the set of delay statements. We use the notation $delay(expr)$ as an alternative to $(expr)$.
- $Send = (ID \cup \{self\}) \times MName \times Expr^* \times Expr \times Expr$ is the set of send statements. We use the notation $x.m(e)\,\mathbf{after}(e_a)\,\mathbf{deadline}(e_d)$ as alternative to (x, m, e, e_a, e_d) to show that message m is sent from actor x with the set of parameters e after e_a units of time and its serving must be started before e_d units of time from now. Note that *after* and *deadline* specifiers are optional and their default values are zero and infinity, respectively.
- *skip* is a predefined statement that has no effect.

In the main part of a model, actors are defined as instances of reactive classes. The set of actors is defined as $Actor = CID \times AID \times AID^* \times Expr^*$ such that

$(c, a, k, p) \in \mathit{Actor}$ defines an actor instantiated from reactive class c, with identifier a, the set of known actors k, and the set of parameters of its constructor p. Having the above definitions, the set of Timed Rebeca models is specified by $\mathcal{P}(\mathit{RClass}) \cup \mathcal{P}(\mathit{Actor})$, where the first component contains the specification of reactive classes and the second component corresponds to the main block consisting of a sequence of actor instantiations.

Finally, we assumed that the Timed Rebeca models are well-formed. The following rules define the well-formedness of a Timed Rebeca model which is hard to (or cannot be) described in the Timed Rebeca grammar, but maybe statically checked.

- **Unique Identifiers.** The actor identifiers are unique within a Timed Rebeca model.
- **Unique Variables.** The names of the state variables of an actor are unique.
- **Unique Methods.** The names of the methods of an actor are unique.
- **Unique Parameters.** The names of the formal parameters of a method are unique and different from the state variables of the enclosing actor.
- **Type Safety.** The model is well typed, i.e.,
  - the expressions are well-typed,
  - both sides of an assignment are of the same type,
  - the conditions of the conditional statements are of type Boolean, and
  - the receiver of a message has a method with the same name as the message.
- **Well-Formed Arguments.** The list of actual arguments passed to a message send statement conforms to the list of formal parameters of the corresponding method, in both length and type.

2.2.2 Fine-Grained Semantics of Timed Rebeca

In this section, we present the fine-grained semantics of Timed Rebeca based on the work of [36]. Prior to presenting the semantics, we present the notations used in the rest of the article.

For a function $f : X \rightarrow Y$, we use the notation $f[x \mapsto y]$ to denote the function $\{(a, b) \in f | a \neq x\} \cup \{(x, y)\}$. Following this, we use the notation $f[x_1 \mapsto y_1 \wedge x_2 \mapsto y_2 \wedge \cdots \wedge x_n \mapsto y_n]$ to denote the function $\{(a, b) \in f | a \notin \{x_1, x_2, \cdots, x_n\}\} \cup \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$. We also use the notation $x \mapsto y$ as an alternative to (x, y). For $X' \subseteq X$, we write $f|X'$ as the restriction of f to X', i.e., $\{(x, y) \in f | x \in X'\}$. Having two sequences a and b of the same size n, the function $\mathit{map}(a, b)$ returns the mapping of the elements of a into b such that $\mathit{map}(a, b) = \{a_i \mapsto b_i | 1 \leq i \leq n\}$, assuming that the elements of a are distinct.

We also define the following auxiliary functions to be used in defining the formal semantics:

- $\mathit{body} : \mathit{AID} \times \mathit{MName} \rightarrow \mathit{Stat}^*$, in which $\mathit{body}(x, m)$ returns the body of the method m of the reactive class which actor identified by x is instantiated from, appended by the special element endm, which denotes the end of the method.

- $params : AID \times MName \to Var^*$, in which $params(x, m)$ returns the list of formal parameters of the method m of the reactive class which the actor identified by x is instantiated from.

- $svars : AID \to \mathcal{P}(Var)$ which returns the names of the state variables of the reactive class which actor identified by x is instantiated from.

- $eval_v : Expr \to Val$ abstracts away the semantics of expressions by evaluating an expression within the specific context $v : Var \to Val$. Note that Val contains all possible values that can be assigned to the state variables or to be used within the expressions. Here, we have $Val = \mathbb{Z} \cup \{True, False\}$. We assume $eval_v$ is overloaded to evaluate a sequence of expressions: $eval_v(\langle e_1, e_2, \cdots, e_n\rangle) = \langle eval_v(e_1), eval_v(e_2), \cdots, eval_v(e_n)\rangle$. Note that $eval_v(e_1), eval_v(e_2), \cdots, eval_v(e_n)$ are evaluated sequentially not in parallel.

Now, the fine-grained semantics of Timed Rebeca can be defined in terms of transition systems (henceforth fine-grained transition system (FGTS)) as the following. In the following, $Msg = AID \times MName \times (Var \to Val) \times \mathbb{N} \times \mathbb{N}$ is used as the type for the messages which are passed among actors. In a message $(i, m, r, a, d) \in Msg$, i is the identifier of the sender of this message, m is the name of its corresponding method, r is a function mapping argument names to their values, a is its arrival time, and d is its deadline.

Definition 1. *For a given Timed Rebeca model $\mathcal{M}$, $FGTS = (S, s_0, Act, \to, AP, L)$ is its fine-grained semantics where S is the set of states, s_0 is the initial state, Act is the set of actions, $\to \subseteq S \times Act \times S$ is the transition relation, AP is the set of atomic propositions, and $L : S \to 2^{AP}$ is the labeling function, described as the following.*

- *The global state of a Timed Rebeca model is represented by a function $s : AID \to (Var \to Val) \times \mathcal{P}_{\mathbb{N}}(Msg) \times Stat^* \times \mathbb{N} \times \mathbb{N} \cup \{\epsilon\}$, which maps an actor's identifier to the local state of the actor. The local state of an actor is defined by a tuple like (v, q, σ, t, r), where $v : Var \to Val$ gives the values of the state variables of the actor, $q : \mathcal{P}_{\mathbb{N}}(Msg)$ is the message bag of the actor, $\sigma : Stat^*$ contains the sequence of statements the actor is going to execute to finish the service to the message currently being processed, t is the actor local time, and r is the time when the actor resumes executing remained statements. Note that the sequence of statements is put as a part of the states to make the operation semantics easier to understand and more readable not for supporting dynamic statement definition and configuration. Also, as mentioned before, we assume that actors communicate via message passing and put their incoming messages into message bags.*

- *In the initial state of the model, for all of the actors, the values of state variables and content of the actor's message bag is set based on the statements of its constructor method, and the remaining statements is set to ϵ. The local times of the actors are set to zero and their resuming times are set to ϵ.*

- *The set of actions is defined as $Act = MName \cup \mathbb{N} \cup \{\tau\}$.*

- *The transition relation $\to \subseteq S \times Act \times S$ defines the transitions between states that occur as the results of actors' activities including: taking a message from* *the mailbox, continuing the execution of statements, and progress in time. The latter is only enabled when the others are disabled for all of the actors. This rule performs the minimum required progress of time to make one of the other rules enabled. As a result, the model of the progress of time in the fine-grained semantics of Timed Rebeca is deterministic. The following SOS rules define these transitions. Note that we associated a rule name with τ transitions to relate τ transitions to their corresponding rules.*

$$\frac{s(x) = (v, \langle (ac, mg, pr, ar, dl)|T\rangle, \epsilon, t, \epsilon) \wedge ar \leq t \wedge dl \geq t}{s \xrightarrow{mg} s[x \mapsto (v \cup pr \cup \{self \mapsto x \wedge sender \mapsto ac\}, T, body(x, mg), t, t)]} \quad \textit{\textbf{(taking-message)}}$$

$$\frac{s(x) = (v, q, \langle var := expr|\sigma\rangle, t, r) \wedge r = t}{s \xrightarrow{\tau_{assign}} s[x \mapsto (v[var \mapsto eval_v(expr)], q, \sigma, t, r)]} \quad \textit{\textbf{(assignment)}}$$

$$\frac{s(x) = (v, q, \langle \textbf{if } expr \textbf{ then } \sigma \textbf{ else } \sigma'|\sigma''\rangle, t, r) \wedge r = t \wedge eval_v(\text{expt}) = \textbf{True}}{s \xrightarrow{\tau_{cond}} s[x \mapsto (v, q, \sigma \oplus \sigma'', t, r)]} \quad \textit{\textbf{(Conditional}}_{T}\textit{\textbf{)}}$$

$$\frac{s(x) = (v, q, \langle \textbf{if } expr \textbf{ then } \sigma \textbf{ else } \sigma'|\sigma''\rangle, t, r) \wedge r = t \wedge eval_v(\text{expt}) = \textbf{False}}{s \xrightarrow{\tau_{cond}} s[x \mapsto (v, q, \sigma' \oplus \sigma'', t, r)]} \quad \textit{\textbf{(Conditional}}_{F}\textit{\textbf{)}}$$

$$\frac{s(x) = (v, q, \langle var := ?(expr_1, expr_2, \cdots, expr_n)|\sigma\rangle, t, r) \wedge r = t}{s \xrightarrow{\tau_{nondet}} s[x \mapsto (v[var \mapsto eval_v(expr_i)], q, \sigma, t, r)]} 1 \leq i \leq n \quad \textit{\textbf{(nondet-assign)}}$$

$$\frac{s(x) = (v, q, \langle y.m(e_1)\ after(e_2)\ deadline(e_3)|\sigma\rangle, t, r) \wedge r = t \wedge s(y) = (v', q', \sigma', t', r') \wedge p = params(y, m)}{s \xrightarrow{\tau_{send}} s[x \mapsto (v, q, \sigma, t, r)][y \mapsto (v', q' \cup \{(m, (map(p, eval_v(e_1))), e_2, e_3)\}, \sigma', t', r']} \quad \textit{\textbf{(send)}}$$

$$\frac{s(x) = (v, q, \langle \textbf{delay}(e)|\sigma\rangle, t, r) \wedge r = t}{s \xrightarrow{\tau_{delay}} s[x \mapsto (v, q, \sigma, t, r + eval_v(e))]} \quad \textit{\textbf{(delay)}}$$

$$\frac{s(x) = (v, q, \langle \textbf{skip}|\sigma\rangle, t, r) \wedge r = t}{s \xrightarrow{\tau_{skip}} s[x \mapsto (v, q, \sigma, t, r)]} \quad \textit{\textbf{(skip)}}$$

$$\frac{s(x) = (v, q, \langle \textbf{endm}\rangle, t, r)}{s \xrightarrow{\tau_{end}} s[x \mapsto (v|_{svars(x)}, q, \epsilon, t, r)]} \quad \textit{\textbf{(end-method)}}$$

$$\frac{s \overset{mg}{\nrightarrow} \wedge s \overset{\tau}{\nrightarrow} \wedge n_1 = \min_{x\in AID}\{ar | s(x) = (v,q,\sigma,t,r) \cdot \sigma = \epsilon \wedge q = \langle (ac,mg,pr,ar,dl)|T\rangle\} \wedge n_2 = \min_{x\in AID}\{r|s(x) = (v,q,\sigma,t,r)\cdot \sigma \neq \epsilon\} \wedge tp = \min\{n_1,n_2\}}{s \xrightarrow{t} \{(x,(v,q,\sigma,tp,r)) \mid (x,(v,q,\sigma,t,r)) \in s\}} \qquad \textit{\textbf{(time-progress)}}$$

- *AP contains the name of all of atomic propositions.*
- *Function $L : S \rightarrow 2^{AP}$ associates a set of atomic propositions with each state, shown by $L(s)$ for a given state s.*

□

To illustrate how FGTS is created for a Timed Rebeca model, we prepared a very simple model in Listing 2.3, the *ping pong* example. In this example, there are two actors, `pi` and `po`, which send messages to each other periodically. Without loss of generality, we assumed that the actors of this model do not have state variables.

Listing 2.3: The Timed Rebeca model of the ping pong example

```
 1 reactiveclass PingActor(3) {
 2   knownrebecs { PongActor po; }
 3   Ping() {
 4     self.ping();
 5   }
 6   msgsrv ping() {
 7     po.pong() after(1);
 8     delay(2);
 9   }
10 }
11
12 reactiveclass PongActor(3) {
13   knownrebecs { PingActor pi; }
14   msgsrv pong() {
15     pi.ping() after (1);
16     delay(1);
17   }
18 }
19 main {
20   PingActor pi(po):();
21   PongActor po(pi):();
22 }
```

Figure 2.5 shows the beginning part of the FGTS of the ping pong example. The first enabled actor of the model is `pi` (as its corresponding actor `PingActor` has a constructor which puts message `ping` in its bag, line 4), so, the first possible transition is taking message `ping`. As shown in the detailed contents of the second state (the gray block), taking the message `ping` results in setting the values of σ and r for the actor `pi`. The next transition results in executing the first statement of the message server `ping`, results in putting the message `pong` in the bag of the actor `po` with release time 1 (because of the value of `after` in line 7). The deadline for this message is ∞ as no specific value is set as the deadline for this message in line 7. As the next statement of the message server `ping` is a delay statement, `pi` cannot continue the execution. The actor `po` cannot cause a transition too. So, the only possible transition is progress in time which is by 1 unit from the third to the fourth state.

2.2.3 Finite Transition Systems and Zeno Behavior

At the final step of defining the fine-grained semantics of Timed Rebeca we have to make clear that there is no explicit time reset operator in Timed Rebeca; so, the 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 words, they perform periodic behaviors over infinite time. Based on this fact, in [26] we proposed a new notion for

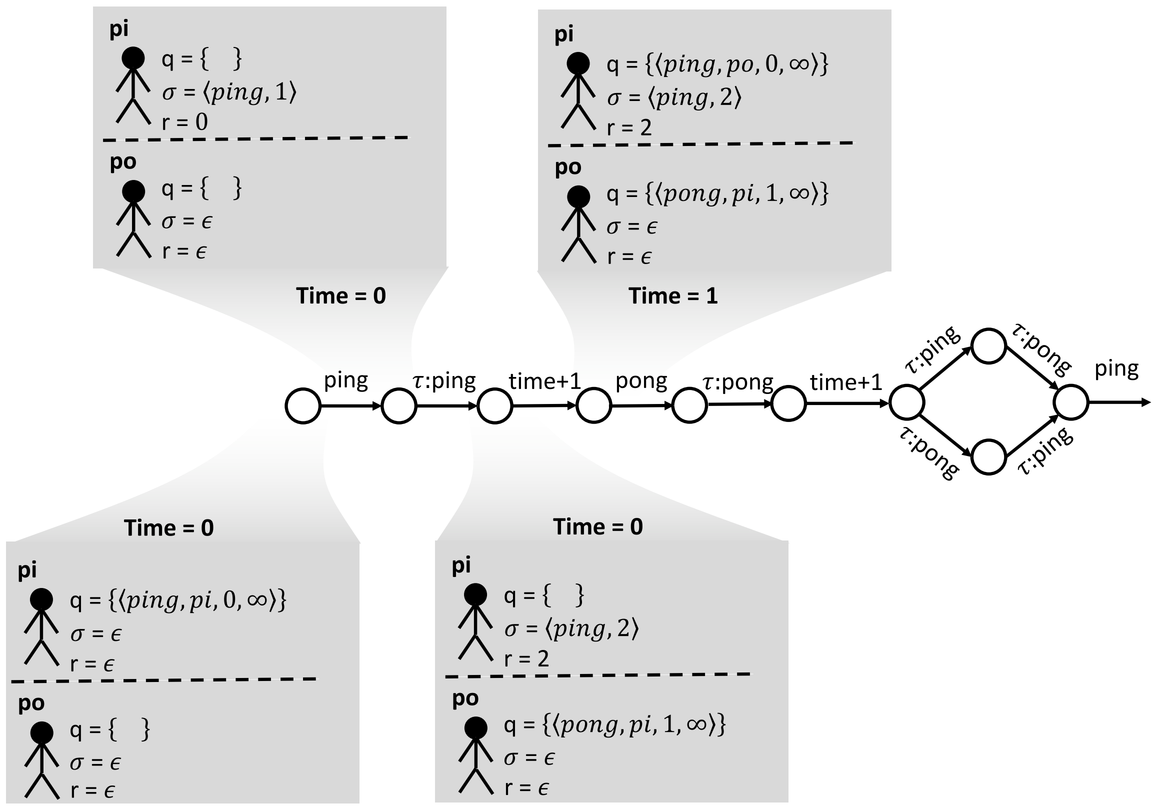


Figure 2.5: The beginning part of the FGTS of the ping pong example

equivalence relation between two states to make the transition systems finite, called *shift equivalence relation*. Intuitively, in shift equivalence relation two states are equivalent if and only if they are the same except for the parts related to the time (values of *now*, resuming times, arrival times and deadlines of messages) and shifting the times of those parts in one state makes it the same as the other one. The formal definition of shift-equivalence relation is depicted in Definition 2.

Definition 2 (Shift-Equivalence Relation between States)**.** *Assume that S is a set of state of a given fine-grained semantics $FGTS = (S, s_0, Act, \rightarrow, AP, L)$. Two states $s, s' \in S$ are in shift-equivalence relation if and only if for all $x \in AID$ where $s(x) = (v_x, q_x, \sigma_x, t_x, r_x)$ and $s'(x) = (v'_x, q'_x, \sigma'_x, t'_x, r'_x)$, there exists $\Delta \in \mathbb{N}$ such that the following conditions hold:*

- $v_x = v'_x$
- $\sigma_x = \sigma'_x$
- $t_x = t'_x + \Delta$
- $r_x = r'_x + \Delta \vee r_x = r'_x = \epsilon$
- *for $\sigma_x = \langle (ac_x, mg_x, pr_x, ar_x, dl_x)|T_x\rangle$ and $\sigma'_x = \langle (ac'_x, mg'_x, pr'_x, ar'_x, dl'_x)|T'_x\rangle$ there are $ac_x = ac'_x$, $mg_x = mg'_x$, $pr_x = pr'_x$, $ar_x = ar'_x + \Delta$, and $dl_x = dl'_x + \Delta$ and this rule is valid for the other elements of T_x and T'_x.*

□

This way, instead of preserving the absolute value of time, only the relative difference of timing parts of states is preserved. As discussed in [26], shift equivalence relation makes transition systems of the majority of Timed Rebeca models finite.

In a state space which is made finite using shift equivalence relation, there is a possibility of having Zeno behavior. 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 elapses of time in Timed Rebeca are 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 1. 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 1: $ZenoFree(s)$ 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      $visited \leftarrow \emptyset$
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 1, 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 $O(V + E)$.

2.3 Experimental Results

To compare the efficiency of the proposed semantics, we have to prepare a set of case studies which are modeled by Timed Rebeca and the size of their state spaces which are generated based on the standard semantics are shown. Note that for each case study we describe an overview of that case, then, present the source code.

Configuration	Standard Semantics	
	#States	Time
1 customer	9	1< sec
2 customers	107	1< sec
3 customers	550	1< sec
4 customers	2.86K	1< sec
5 customers	16.9K	1< sec
6 customers	114K	2 secs
7 customers	884K	3 sec

Table 2.2: Model checking times and size of state spaces for the standard semantics of the ticket service system

2.3.1 Ticket Service System

My first example is the model of a *Ticket Service* system. The overview of this example is presented in Section 2.1. We created the extended version of this model and varying in the number of customers.

The Timed Rebeca model of this system for the case of five customers, shown in Listing 2.4, contains three different reactive classes: `Customer`, `Agent`, and `TicketService`. Customers periodically ask for tickets by sending the message `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 and waits for the response. 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`.

Listing 2.4: The model of a ticket service system with five customers

```
 1 reactiveclass Customer {
 2     knownrebecs { Agent a; }
 3     statevars { byte id; }
 4     Customer(byte myId) {
 5         id = myId;
 6         self.try();
 7     }
 8     msgsrv try() {
 9         a.requestTicket(id);
10     }
11     msgsrv ticketIssued() {
12         self.try() after(30);
13     }
14 }
15 reactiveclass Agent {
16     knownrebecs {
17         TicketService ts;
18         Customer c1, c2, c3, c4, c5;
19     }
20     msgsrv requestTicket(byte id) {
21         ts.requestTicket(id) deadline(24);
22     }
23     msgsrv ticketIssued(byte id) {
24         switch(id) {
25             case 1: c1.ticketIssued();
26             case 2: c2.ticketIssued();
27             case 3: c3.ticketIssued();
28             case 4: c4.ticketIssued();
29             case 5: c5.ticketIssued();
30         }
31     }
32 }
33 reactiveclass TicketService {
34     knownrebecs { Agent a; }
35     statevars { int issueDelay; }
36     TicketService(int myIssueDelay) {
37         issueDelay = myIssueDelay;
38     }
39     msgsrv requestTicket(byte id) {
40         delay(issueDelay);
41         a.ticketIssued(id);
42     }
43 }
44 main {
45     Agent a(ts, c1, c2, c3, c4, c5):();
46     TicketService ts(a):(2);
47     Customer c1(a):(1), c2(a):(2), c3(a):(3), c4(a):(4), c5(a):(5);
48 }
```

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

Configuration	Standard Semantics	
	#States	Time
1 Sensor	75	1< sec
2 Sensors	389	1< sec
3 Sensors	2.15K	1 sec
4 Sensors	12.37K	12 secs

Table 2.3: Model checking times and size of state spaces for the standard semantics of the Gas Sensing system

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`.

The characteristics of the state spaces which are generated for this model are presented in Table 2.2.

2.3.2 A Toxic Gas Sensing and Rescue System

My second example models a lab environment in which the level of a toxic gas changes over time. If this level rises above a certain threshold, the scientist's life is in danger. Sensors in the lab constantly measure the amount of toxicity in the air and send the measurements to a central controller which periodically checks whether the scientist is in danger. If so, it notifies the scientist about the danger. The scientist should acknowledge the alarm; if he fails to do so in a timely manner, the controller notifies a rescue team. When the team reaches the lab it notifies the controller that the scientist has been rescued. If the controller does not receive this notification, it reaches the conclusion that the scientist has lost his life.

Our Timed Rebeca model of this system is shown in Listing 2.5 and contains four reactive classes: `Sensor`, `Controller`, `Scientist`, and `Rescue` in the model. The sensors periodically measure the level of toxic gas in the environment (which is modeled by the nondeterministic assignment in line 12). After sensing, they report the measured data to the only `Controller` instance. Upon receiving the measured data from each `Sensor` (in the `report` message server), `Controller` stores the value in its corresponding `sensorVal` state variable.

Periodically, in the `checkSensors` message server, `Controller` checks if the values are above the normal. If high toxicity is detected, `Controller` alarms `Scientist` (by sending him `abortPlan`), and schedules a task to check for the scientist's acknowledgment (line 47). It also resets the sensor values to normal to prevent sending repeated alarms for the same event. If the controller does not receive an `ack` message from `Scientist`, the rescue team is informed about the situation. Again, the controller sets a time-out for receiving the rescue notification by sending itself a `checkRescue` message (line 56).

In this model, the *network delay* is assumed to be one time unit (line 1). The period of measurements done by the sensors is 10 (line 89), and the period of the controller checking the sensors' data is 15 time units (line 32). The time-outs for receiving acknowledgment from the scientist and the rescue team are 5 and 7 time units (lines 30 and 31), respectively. Finally, the team reaches the lab in 5 time units (line 83).

Listing 2.5: Timed Rebeca model of the toxic gas sensing and rescue system

```
env int netDelay = 1;
reactiveclass Sensor {
  knownrebecs { Controller ctrl; }
  statevars {
    int period, gasLevel;
  }
  Sensor(int myPeriod) {
    period = myPeriod;
    self.doReport();
  }
  msgsrv doReport() {
    gasLevel=?(0,1);
    ctrl.report(gasLevel) after(netDelay);
    self.doReport() after(period);
  }
}

reactiveclass Controller {
  knownrebecs {
    Sensor sensor0, sensor1;
    Scientist scientist;
    Rescue rescue;
  }
  statevars {
    int sensorVal0, sensorVal1;
    boolean sciAck, sciRescued, sciDead;
    int rescueDL, ctrlCheckDelay, sciDL;
  }
  Controller() {
    rescueDL = 7;
    sciDL = 5;
    ctrlCheckDelay = 15;
    self.checkSensors();
  }
  msgsrv report(int value) {
    if (sender == sensor0)
      sensorVal0 = value;
    else
      sensorVal1 = value;
  }
  msgsrv rescueReached() {
    sciRescued = true;
  }
  msgsrv checkSensors() {
    if (sensorVal0 > 0 || sensorVal1 > 0) {
      scientist.abortPlan() after(netDelay);
      self.checkSciAck() after(sciDL);
    }
    self.checkSensors() after(ctrlCheckDelay);
    sensorVal0 = 0;
    sensorVal1 = 0;
  }
  msgsrv checkSciAck() {
    if (!sciAck) {
      rescue.go() after(netDelay);
      self.checkRescue() after(rescueDL);
    }
    sciAck = false;
  }
  msgsrv checkRescue() {
    if (!sciRescued)
      sciDead = true;
    else
      sciRescued = false;
  }
  msgsrv ack() { sciAck = true; }
}

reactiveclass Scientist {
  knownrebecs { Controller ctrl; }
  msgsrv abortPlan() {
    ctrl.ack() after(netDelay);
  }
}

reactiveclass Rescue {
  knownrebecs { Controller ctrl; }
  statevars {
    int reachDelay, rescueDL;
  }
  Rescue() {
    rescueDL = 10;
    reachDelay = 5;
  }
  msgsrv go() { ... }
}

main {
  Sensor sensor0(ctrl):(10);
  Sensor sensor1(ctrl):(10);
  Scientist scientist(ctrl):();
  Rescue rescue(ctrl):();
  Controller ctrl(sensor0, sensor1, scientist, rescue):();
}
```

The characteristics of the state spaces which are generated for this model are presented in Table 2.3.

2.3.3 The IEEE 802.11 RTS/CTS Collision Avoidance Protocol

The next example is the simplified version of IEEE 802.11 RTS/CTS protocol for collision avoidance in wireless networks. Using this protocol, when a node decides to send data to another node, it sends a *Request to Send* (RTS) message to the destination node, which is expected to reply with a *Clear to Send* (CTS) message. Other nodes in the network which receive an RTS or a CTS message wait for a certain amount of time, making the medium free for the two communicating nodes. This mechanism also addresses the *hidden node problem*, which occurs when two nodes want to send data to the same node. The destination node is in the transmission range of both senders, but the senders are out of the transmission ranges of each other (and therefore unaware of each other's decision to send a message). In the protocol, the destination node sends a CTS message to only one of the senders. The other sender waits for a random amount of time, and then sends an RTS message to the destination node again. Furthermore, this protocol solves the *exposed node problem* as well, where two adjacent nodes send data to two different destination nodes, so the interference of data transfer of adjacent senders results in message collision. The problem is solved by preventing the senders from sending data after receiving the CTS message from other sender nodes.

Our Timed Rebeca model of the protocol, shown in Listing 2.6, has two reactive classes: `Node` and `RadioTransfer`. Each `Node` knows a `RadioTransfer` actor, which is responsible for broadcasting its messages to all nodes in the sender's transmission range. We assume that each node has two nodes in its transmission range and that the transmission delay is two units of time.

To transmit data, the sender sends an RTS message to the receiver (through its `RadioTransfer` actor, line 35) and waits for the response. When an RTS message is delivered, the receiver checks whether the network is busy (line 47). If so, it sends an RTS message to itself after a random `backOff` (modeled by a nondeterministic choice among the values $\{4, 5\}$). If the receiver is not the target of the message, it marks the status of the network as busy (line 54). Otherwise, it sends a CTS message to the sender (line 47). When a node receives a CTS message, it checks whether it is the target of the message. If so, it sends its data. If not, it sets the network status to idle (`rcvCTS` message server, lines 59-61).

Listing 2.6: Timed Rebeca model of the IEEE 802.11 collision avoidance protocol

```
 1 reactiveclass RadioTransfer {
 2   knownrebecs { Node node1, node2; }
 3   RadioTransfer() {}
 4   msgsrv passRTS(int sndr,int rcvr) {
 5     delay(2);
 6     node1.rcvRTS(sndr,rcvr);
 7     node2.rcvRTS(sndr,rcvr);
 8   }
 9   msgsrv passCTS(int sndr,int rcvr) {
10     delay(2);
11     node1.rcvCTS(sndr,rcvr);
12     node2.rcvCTS(sndr,rcvr);
13   }
14   msgsrv passData(int sndr,int rcvr) {
15     node1.rcvData(sndr,rcvr);
16   }
17 }
18 reactiveclass Node {
19   knownrebecs { RadioTransfer radioTransfer; }
20   statevars {
21     int dest, id, dataRate;
22     boolean channleIdle;
23   }
24   Node(int myId, int myDest, boolean chIdle, int myRate) {
25     id = myId;
26     dest = myDest;
27     channleIdle = chIdle;
28     dataRate = myRate;
29     self.sendRTS() after(dataRate);
30   }
31   msgsrv sendRTS() {
32     if (channleIdle)
33       radioTransfer.passRTS(id, dest);
34     else
35       self.sendRTS() after(?(4,5));
36   }
37   msgsrv sendData() {
38     radioTransfer.passData(id, dest);
39     self.sendRTS() after(dataRate);
40   }
41   msgsrv rcvRTS(int sndr,int rcvr) {
42     if (rcvr == id) {
43       if (channleIdle) {
44         channleIdle = false;
45         radioTransfer.passCTS(id, sndr);
46       } else {
47         self.rcvRTS(sndr,rcvr) after(?(4,5));
48       }
49     } else {
50       channleIdle = false;
51     }
52   }
53   msgsrv rcvCTS(int sndr,int rcvr) {
54     if(rcvr == id)
55       self.sendData();
56     else
57       channleIdle = true;
58   }
59   msgsrv rcvData(int sndr,int rcvr) {
60     channleIdle = true;
61   }
62 }
63 main {
64   RadioTransfer rt1(n2, n4):(), rt2(n3, n1):(), rt3(n4, n2):(), rt4(n1, n3):();
65   Node n1(rt1):(1,2,true,2), n2(rt2):(2,3,true,5), n3(rt3):(3,4,true,2), n4(rt4):(4,1,true,5);
66 }
```

Configuration	Standard Semantics	
	#States	Time
2 Clients	107	1< sec
3 Clients	550	1< sec
4 Clients	2.86K	1< sec
5 Clients	16.9K	1< sec

Table 2.4: Model checking times and size of state spaces for the standard semantics of the CA protocol

The characteristics of the state spaces which are generated for this model are presented in Table 2.4.

2.3.4 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 [48]. 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 design, 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 [49] 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.

The simplified version of the Timed Rebeca model of ASPIN is shown in Listing 2.7, 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 Listing 2.7, 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).

Listing 2.7: The model of an ASPIN NoC

```
 1 env short maxTime = 28000;
 2 env short rAlg = 1;
 3 env byte writeD = 2;
 4 ...
 5 reactiveclass Manager(60){
 6   knownrebecs{
 7     Router r00, r10, r20, r30,
 8          r01, r11, r21, r31,
 9          r02, r12, r22, r32,
10          r03, r13, r23, r33;
11   }
12   Manager(){
13     generate();
14   }
15   msgsrv reset(){ ... }
16   void generate(){
17     r01.reStart()after(wholeCycle);
18     r11.checkRecieved(2) after(maxTime);
19     r00.inReq(4,1,1,1) after (18);
20     r00.inReqMinus(4) after (18 + prodD);
21     r00.inReq(4,1,1,2) after (110);
22     r00.inReqMinus(4) after (110 + prodD);
23     ...
24   }
25 }
26 reactiveclass Router(80) {
27   knownrebecs {
28     Manager manager;
29     Router N, E, S, W;
30   }
31   statevars {
32     byte Xid, Yid, received;
33     bboolean[5] inBufFull, outBufFull
34     byte[5][2] outPortPtr;
35     ...
36   }
37   Router(byte X, byte Y){
38     Xid = X;
39     Yid = Y;
40     for(byte i=0;i<5;i++){
41       waitedOutReq[i] = 5;
42       outReqEnable[i] = true;
43       outPortPtr [i][0]= -1;
44     }
45     ...
46   }
47   msgsrv inReq (byte inPort, byte Xtarget, ↩
         ↪byte Ytarget,byte id){
48     if (inBufFull[inPort] == false ){
49       sendInAck((byte)(inPort + 2)/ 4, inAD);
50       self.process(inPort, Xtarget, ↩
             ↪Ytarget,id, false, ↩
             ↪false)after((writeD * ↩
             ↪inBufSizeTest)+ readD);
51       ...
52     } else { ... }
53   }
54   msgsrv process(byte inPort, byte Xtarget, ↩
         ↪byte Ytarget,byte id, boolean ↩
         ↪isPushed, boolean justPush) {
55     byte routeD;
56     ...
57     if ((inBufID[inPort][0] == id) || ↩
           ↪isPushed == true){
58       if(passedFlit == 0) {
59         if (rAlg == 1) {
60           outPort = XYrouting(Xtarget, Ytarget);
61           routeD = routeXYD;
62         }
63         else if (rAlg == 2){ ... }
64         else if (rAlg == 3){ ... }
65       } else { ... }
66 
67       if(outReqEnable[inPort] == true){
68         waitedOutReq[inPort] = outPort;
69         self.portSchedule(outPort, inPort) ↩
               ↪after(routeD + schdD + outRD);
70       }
71     }
72   }
73   byte XYrouting(byte Xtarget, byte Ytarget) {
74     byte outPort = 0;
75     if(Xtarget > Xid)  outPort = 1;
76     else if(Xtarget < Xid) outPort = 3;
77     else if(Ytarget > Yid) outPort = 2;
78     else if(Ytarget < Yid) outPort = 0;
79     else outPort = 4;
80     return outPort;
81   }
82   ...
83 }
84 main {
85   Manager m(r00, r10, ..., r33):();
86   Router r00(m,r03,r10,r01,r30):(0,0), ↩
         ↪r10(m,r13,r20,r11,r00):(1,0), ..., ↩
         ↪r23(m,r22,r33,r20,r13):(2,3), ↩
         ↪r33(m,r32,r03,r30,r23):(3,3);
87 }
```

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 the 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

Configuration	Standard Semantics	
	#States	Time
3 Packets	442	1s
4 Packets	1,239	2s
5 Packets	3,117	7s
6 Packets	9,907	35s
7 Packets	35,746	6.8m
8 Packets	136,666	1.4h

Table 2.5: Model checking times and size of state spaces for the standard semantics of the NoC model

neighbor (lines 59 to 64). As shown the details of routing by XY algorithm in line 60, the output port of a packet is computed 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.

The characteristics of the state spaces which are generated for this model are presented in Table 2.5.

2.3.5 Hadoop YARN Scheduler

Hadoop [50] is a framework for MapReduce, a programming model for generating and processing large data sets [51]. MapReduce has undergone a complete overhaul in its latest release, called MapReduce 2.0 (MRv2) or YARN [52]. 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.

In the Timed Rebeca model of Listing 2.8, the YARN system is modeled using two reactive classes: `ResourceManager` and `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 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).

Listing 2.8: The model of a Hadoop YARN system with three application masters

```
  1 reactiveclass ResourceManager(5) {
  2   knownrebecs {
  3     AppMaster am1, am2, am3;
  4   }
  5   statevars {
  6     int FREE, BUSY;
  7     int appMaster1, appMaster2, appMaster3;
  8     int m_queue_misses, m_update_miss, m_job_complete, DEFAULT_DL, QUEUE_SIZE;
  9     int[4] fifo_queue;
 10   }
 11
 12   ResourceManager() {
 13     FREE = 1;
 14     BUSY = 0;
 15     appMaster1 = FREE;
 16     appMaster2 = FREE;
 17     appMaster3 = FREE;
 18     m_queue_misses = 0;
 19     m_update_miss = 0;
 20     m_job_complete = 0;
 21     DEFAULT_DL = 3;
 22     fifo_queue[0] = DEFAULT_DL;
 23     fifo_queue[1] = DEFAULT_DL;
 24     fifo_queue[2] = DEFAULT_DL;
 25     fifo_queue[3] = DEFAULT_DL;
 26     QUEUE_SIZE = 4;
 27     self.checkQueue();
 28   }
 29   msgsrv checkQueue() {
 30     m_queue_misses = 0;
 31     m_update_miss = 0;
 32     m_job_complete = 0;
 33     int I = 0;
 34     if(appMaster1 == FREE) {
 35       appMaster1 = BUSY;
 36       am1.runJob(fifo_queue[0]);
 37       I = 0;
 38       while(I < QUEUE_SIZE - 1) {
 39         fifo_queue[I] = fifo_queue[I + 1];
 40         I++;
 41       }
 42       fifo_queue[QUEUE_SIZE - 1] = DEFAULT_DL;
 43     }
 44     if(appMaster2 == FREE) { ... }
 45     if(appMaster3 == FREE) { ... }
 46     I = 0;
 47     int J = 0;
 48     while(I < QUEUE_SIZE) {
 49       fifo_queue[I]--;
 50       if(fifo_queue[I] == 0) {
 51         m_queue_misses++;
 52         J = I;
 53         while(J < QUEUE_SIZE - 1) {
 54           fifo_queue[J] = fifo_queue[J + 1];
 55           J++;
 56         }
 57         fifo_queue[QUEUE_SIZE - 1] = DEFAULT_DL;
 58       }
 59       I++;
 60     }
 61     self.checkQueue() after(1);
 62   }
 63   msgsrv update(boolean deadline_miss) {
 64     m_queue_misses = 0;
 65     m_update_miss = 0;
 66     m_job_complete = 0;
 67     if(deadline_miss == true) {
 68       m_update_miss = 1;
 69     } else {
 70       m_job_complete = 1;
 71     }
 72     if(sender == am1) {
 73       appMaster1 = FREE;
 74     } else if(sender == am2) {
 75       appMaster2 = FREE;
 76     } else if(sender == am3) {
 77       appMaster3 = FREE;
 78     }
 79   }
 80 }
 81
 82 reactiveclass AppMaster(5) {
 83   knownrebecs {
 84     ResourceManager rm;
 85   }
 86   statevars { int doneJobs; }
 87
 88   AppMaster() { doneJobs = 0; }
 89   msgsrv runJob(int dline) {
 90     int completion = 2;
 91     boolean deadline_miss;
 92     if(completion > dline) {
 93       deadline_miss = true;
 94       rm.update(deadline_miss) after(dline);
 95     } else {
 96       deadline_miss = false;
 97       rm.update(deadline_miss) after(completion);
 98       doneJobs++;
 99       if (doneJobs > 5) doneJobs = 1;
100     }
101   }
102 }
103 main {
104   ResourceManager rm(am1, am2, am3):();
105   AppMaster am1(rm):();
106   AppMaster am2(rm):();
107   AppMaster am3(rm):();
108 }
```

The characteristics of the state spaces which are generated for this model are presented in Table 2.6.

Configuration	Standard Semantics	
	#States	Time
1 AMs	180	<1s
2 AMs	5,506	1s
3 AMs	177,989	14.5m

Table 2.6: Model checking times and size of state spaces for the standard semantics of the Hadoop Yarn model

2.3.6 WSAN Applications

As the fourth example, we present a realtime data acquisition system for structural health monitoring and control (SHMC) of civil infrastructures [53]. This system has been implemented on top of the Imote2 [54] 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 realtime requirements while having the resource limitations of low-power embedded WSAN platforms. Ensuring the safe execution of an 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 [55]. Here, we showed how other properties can be model checked using the TCTL model checking of Timed Rebeca.

The simplified version of the Timed Rebeca model of WSAN, shown in Listing 2.9, 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 B-MAC communication protocols in [55]). 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).

Listing 2.9: The model of a WSAN application

```
  1 env int samplingRate = 25;
  2 env int numberOfNodes = 6;
  3 env int bufferSize = 2;
  4 env int sensorTaskDelay = 2;
  5 env int OnePacketTT = 7;
  6 env int miscTaskDelay = 10;
  7 env int tmdaSlotSize = 10;
  8 env int miscPeriod = 120;
  9 env int packetMaximumSize = 112;
 10
 11 reactiveclass Sensor(10) {
 12     knownrebecs { CPU cpu; }
 13     Sensor() { self.sensorFirst(); }
 14     msgsrv sensorFirst() {
 15         self.sensorLoop() after(?(10, 20, 30));
 16     }
 17     msgsrv sensorLoop() {
 18         int period = 1000 / samplingRate;
 19         cpu.sensorEvent(period);
 20         self.sensorLoop() after(period);
 21     }
 22 }
 23
 24 reactiveclass Misc(10) { ... }
 25
 26 reactiveclass CPU(10) {
 27     knownrebecs {
 28         CommunicationDevice senderDevice, receiverDevice;
 29         Sensor sensor;
 30     }
 31     statevars { int samples; }
 32     CPU() { samples = 0; }
 33     msgsrv sensorEvent(int period) {
 34         self.sensorTask(period, currentMessageWaitingTime);
 35     }
 36     msgsrv sensorTask(int period, int lag) {
 37         int tmp = period - lag - currentMessageWaitingTime;
 38         assertion(tmp >= 0);
 39         delay(sensorTaskDelay);
 40         samples += 1;
 41         if (samples == bufferSize){
 42             senderDevice.send( receiverDevice,0,1);
 43             samples = 0;
 44         }
 45     }
 46     msgsrv miscEvent() { delay(miscTaskDelay); }
 47 }
 48
 49 reactiveclass CommunicationDevice (10) {
 50     knownrebecs { WirelessMedium medium; }
 51     statevars {
 52         byte id;
 53         int sendingData;
 54         int sendingPacketsNumber;
 55         CommunicationDevice receiverDevice;
 56     }
 57     CommunicationDevice(byte myId) {
 58         id = myId;
 59         sendingData = 0;
 60         sendingPacketsNumber = 0;
 61         receiverDevice = null;
 62     }
 63     msgsrv send(CommunicationDevice receiver, int data, int packetsNumber) {
 64         assertion(receiverDevice == null);
 65         sendingPacketsNumber = packetsNumber;
 66         receiverDevice = receiver;
 67         sendingData = data;
 68         medium.getStatus();
 69     }
 70     msgsrv receiveStatus(boolean result) { ... }
 71     msgsrv receiveResult(boolean result) { ... }
 72     msgsrv receiveData(CommunicationDevice receiver, int data, int receivingPacketsNumber) { ... }
 73 }
 74
 75 reactiveclass WirelessMedium(5) {
 76     statevars {
 77         CommunicationDevice senderDevice;
 78         CommunicationDevice receiverDevice;
 79         int maxTraffic;
 80     }
 81     WirelessMedium() {
 82         senderDevice = null;
 83         receiverDevice = null;
 84         maxTraffic = (125 * 1024) / 8;
 85     }
 86     msgsrv getStatus() { ... }
 87     msgsrv broadcast(CommunicationDevice receiver, int data, int packetsNumber) { ... }
 88     msgsrv broadcastingIsCompleted() {
 89         senderDevice = null;
 90         receiverDevice = null;
 91     }
 92 }
 93
 94 main {
 95   WirelessMedium medium():();
 96     CPU cpu (sensorNodeSenderDevice, receiver, sensor):();
 97     Sensor sensor(cpu):();
 98     Misc misc(cpu):();
 99     CommunicationDevice sensorNodeSenderDevice (medium):((byte)1);
100     CommunicationDevice receiver(medium):((byte)0);
101 }
```

Configuration	Standard Semantics	
	#States	Time
25-5-3-10	1,741	<1s
33-6-4-2	1,934	<1s
25-5-4-10	3,718	1s
30-6-4-2	9,353	1s
25-6-4-2	34,503	2s
20-6-4-2	57,621	3s

Table 2.7: Model checking times and size of state spaces for the standard semantics of WSAN model

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.

The characteristics of the state spaces which are generated for this model are presented in Table 2.7.

Chapter 3

TCTL Model Checking for Timed Rebeca[1]

TTSs are expressive enough for modeling the behavior of the majority of realtime distributed systems; however, the formal verification of TTS is PSPACE-complete [37]. Therefore, currently, there is no polynomial time algorithm for the verification of TTSs. The most widely used model checking toolset, e.g. UPPAAL, only supports a limited subset of Timed Computation Tree Logic (TCTL) which can be model checked efficiently [30]. The source of this inefficiency in the analysis of TTS and timed automata is in how the passage of time is modeled. The model of time in TTS 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 [41], [42] by assuming that each transition takes exactly one time unit. Later, a minor extension has been added to this work by allowing the existence of instantaneous transitions (zero time transitions) in [43]. Finally, Timed Transition Graph (TTG) [44] and Durational Transition Graph (DTG) [45] extended the former works by associating discrete time duration with transitions. Although TTG and DTG are less expressive than TTS, 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 TCTL$_{\leq,\geq}$ 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 $O(V \cdot (V + E) \cdot |\Phi|)$ [45]. Here, we are going to use the same approach for the model checking of FGTSs of Timed Rebeca models. To this end, we reviewed the details of these model checking algorithms in Section 3.1. Note that, while TCTL$_{\leq,\geq}$ can be model checked for DTGs in polynomial time, the model checking against TCTL$_{=}$ properties (i.e. TCTL properties with sub-formulas of the form $\Phi\ \mathbf{U}^{=c}\ \Psi$) is an NP-hard problem. We also improved the running time of the algorithm of [45] 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 the model checking of TCTL$_{\leq,\geq}$ properties since its running

[1]This chapter is an improvement and extension of the results published in [36].

time is the same as the tight running time of the CTL model checking algorithm [46]. This algorithm is presented in detail in Section 3.2.

3.1 Timed Model Checking of Discrete Time Systems against TCTL properties

As discussed in [37], timed transition system is expressive enough for modeling the behavior of the majority of realtime systems. However, the verification algorithms of timed transition systems are PSPACE-complete. In practice, it is hard to use timed transition systems for the efficient analysis of real-world systems. The same holds for the verification of realtime systems with dense time presented in other semantics (region transition system, etc.) [46]. For the latter case, there is an efficient algorithm for the verification of a subset of TCTL properties which does not have nested timed quantifiers.

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), is a timed transition system where the duration intervals of transitions are interpreted in the domain of natural numbers [45]. 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 3 (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 ($v \in \Upsilon \wedge v = [n, m] \cdot n, m \in \mathbb{N}$) or right-open infinite ($v \in \Upsilon \wedge v = [n, \infty) \cdot n \in \mathbb{N}$) intervals. □

DTGs can be model checked against Timed CTL (TCTL) properties [46] efficiently. TCTL is a realtime 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 4 (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 $\sim \in \{<, \leq, =, \geq, >\}$. 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 [45], and can be safely *omitted from the syntax and semantics of (T)CTL. However, in this chapter, we use these modalities to make formulas easier to read and understand. Also, note that $|\Phi|$ is defined as the size of the formula Φ, which is the number of modalities' instances in Φ. For example, for a given TCTL formula $\Phi = E(AG^{\geq c_1}\,\Phi_1)\,\mathrm{U}^{\leq c_2}\,E(\Phi_2\,\mathrm{U}^{<c_3}\,\Phi_3)$ the value of $|\Phi|$ is three as there are two EUs and one AG in the formula.* □

In the following, we present the semantics of TCTL properties over DTGs based on the work of [45]. The clauses of Definition 6 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 $s \overset{d_0}{\rightarrow} s_1 \overset{d_1}{\rightarrow} \cdots$, as described below.

Definition 5 (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 $Paths(DTG, s)$ is defined as the set of all valid timed paths of DTG which are started from the state s.* □

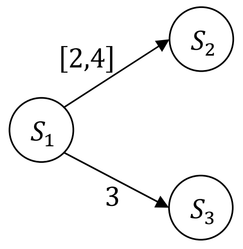
Definition 6 (Semantics of TCTL)**.** *A given TCTL formula Φ holds for state s of $DTG = (S, s_0, \rightarrow, AP, L)$ as described by the following items.*

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

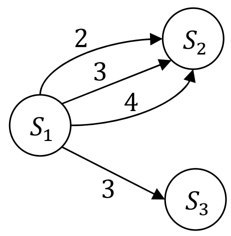
□

Using the above semantics for the model checking of DTGs against TCTL formulas requires resolving the nondeterminism of durations of transitions. The meaning of a duration of a transition between two states can be interpreted in different ways. Here, we introduce two interpretations of durations on a transition, called *jump semantics* and *continuous-early semantics* [45]. In these two timed transition systems, 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.

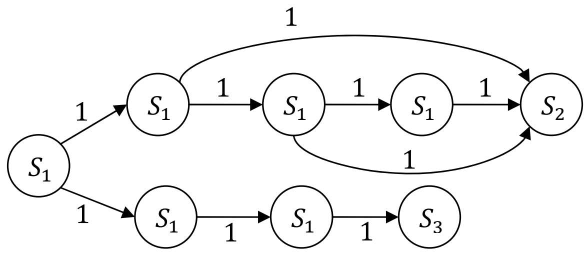
Jump Semantics. In this semantics, moving from the state s to the state s' takes an integer time $d \in [n, m]$. Here, before starting transition from s to s', the value of d is determined, and then the system waits for d units of time and it reaches state s' at time $t + d$. Figure 3.1b shows how this semantics works for the DTG of Figure 3.1a. The idea of this semantics is the same as the semantics of Timed Transition Graph [44] and the semantics of Timed Rebeca as it is described in Section 2.2.2.



(a) A DTG with three states



(b) The timed transition system of the DTG assuming the jump semantics



(c) The timed transition system of the DTG assuming the continuous-early semantics

Figure 3.1: An intuitive representation of the timed transition systems with respect to the jump and continuous-early semantics (it shows the continuous-early semantics as nondeterminism is resolved immediately in the first S_1) [45].

Continuous-Early 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-early 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 3.1c shows how this semantics works for the DTG of Figure 3.1a.

For a given DTG, two timed transition systems generated based on jump semantics and continuous-early semantics are not bisimilar. This can be shown by TCTL formulas which are satisfied by one of them but are violated by the other one. For example, in the DTG of Figure 3.1a, TCTL property $\mathbf{A}(\mathbf{EF}(s_3)\ \mathbf{U}^{\leq 5}\ (s_3 \vee s_2))$ is satisfied in the timed transition system of its jump semantics. As shown in Figure 3.1b 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 3.1c, 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-early semantics, there are polynomial-time model checking algorithms for TCTL$_{\leq,\geq}$ properties; however, model checking of TCTL$_{=}$, TLTL, and TCTL* properties remain PSPACE-complete. In the following, we review the model checking algorithm of DTGs in jump semantics against TCTL$_{\leq,\geq}$ properties according to [45]. As in the 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 $\mathbf{A}(\Phi\mathbf{U^{\sim c}}\Psi)$.

$Sat(\mathbf{E}(\Phi\,\mathbf{U^{\leq c}}\Psi))$**:** 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 the 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 $Sat(\mathbf{E}(\Phi\mathbf{U^{\leq c}}\Psi))$ 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 $O((V+E) + V\cdot(V+E)) = O(V\cdot(V+E))$.

$Sat(\mathbf{E}(\Phi\,\mathbf{U^{\geq c}}\Psi))$**:** 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 [47] 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 $Sat(\mathbf{E}(\Phi\mathbf{U^{\geq c}}\Psi))$ 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 $O((V+E)+V\cdot(V+E)) = O(V\cdot(V+E))$.

$Sat(\mathbf{A}(\Phi\,\mathbf{U^{\leq c}}\Psi))$**:** using the equivalence relations $\mathbf{A}(\Phi\,\mathbf{U^{\leq c}}\Psi) \equiv \mathbf{AF^{\leq c}}\,\Psi \wedge \neg\mathbf{E}((\neg\Psi)\mathbf{U}\,(\neg\Phi \wedge \neg\Psi))$ and $\mathbf{AF^{\leq c}\,\Psi} \equiv \neg\mathbf{E}(\neg\Psi\,\mathbf{U}^{>c}\,\top) \wedge \neg\mathbf{E}(\neg\Psi\,\mathbf{U}\,P_{SCC^{0}(\neg\Psi)})$, 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 $O((V+E)+V\cdot(V+E)) = O(V\cdot(V+E))$.

$Sat(\mathbf{A}(\Phi\,\mathbf{U^{\geq c}}\Psi))$**:** using the equivalence relation $\mathbf{A}(\Phi\,\mathbf{U^{\geq c}}\Psi) \equiv \mathbf{AG}^{<c}\,(\Phi \wedge \mathbf{A}(\Phi\,\mathbf{U}^{>0}\Psi))$ 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 the model checking of DTGs in the continuous-early semantics is out of the scope of this thesis, it is not described here.

3.2 Improving the TCTL$_{\leq,\geq}$ Model Checking Algorithm

In the previous section, we illustrated how DTGs can be model checked against TCTL$_{\leq,\geq}$ properties with running time $O(V \cdot (V + E) \cdot |\Phi|)$. In this section, we show how the two phases of the TCTL$_{\leq,\geq}$ model checking algorithm are combined to develop a new TCTL$_{\leq,\geq}$ 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 $\mathbf{E}(\Phi\, \mathbf{U}^{\leq c}\, \Psi)$ and $\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi)$. As shown in the previous section, other TCTL formulas can be constructed using $\mathbf{EU}^{\leq c}$, $\mathbf{EU}^{\geq c}$, 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 [46]. As shown in Algorithm 2, 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 $s \in S \setminus T$ 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 $\mathbf{E}(\Phi \mathbf{U}^{\leq c} \Psi)$ or $\mathbf{E}(\Phi \mathbf{U}^{\geq c} \Psi)$ the values of $Sat(\Phi)$ and $Sat(\Psi)$ are computed in advance.

Algorithm 2: Enumerative backward search for computing $Sat(\mathbf{E}(\Phi\, \mathbf{U}\, \Psi))$ [46]

```
   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          $Q \leftarrow Q \setminus \{s\}$
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 $\mathbf{EU}^{\leq c}$ and $\mathbf{EU}^{\geq c}$.

3.2.1 Calculating $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leq c}\,\Psi))$

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 the time constraint are started from the goal states, they can be combined together. The details of the new algorithm for calculating $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leq c}\,\Psi))$ are depicted in Algorithm 3. 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 `low_time` $: S \times S \to \mathbb{N}$ is defined to retrieve the lower bound of the associated progress of time with the transition between two given states.

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

```
   Input: A DTG with the set of states $S$ and the TCTL$_{\leq,\geq}$ formula $\mathbf{E}(\Phi\,\mathbf{U}^{\leq c}\,\Psi)$
   Output: $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leq c}\,\Psi)) = \{s \in S \mid s \models \mathbf{E}(\Phi\,\mathbf{U}^{\leq c}\,\Psi)\}$
 1 begin
 2     $T \leftarrow$ Sat($\Psi$)
 3     $Q \leftarrow$ EMPTY_HEAP
 4     foreach state $s \in S \setminus T$ 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                     $\delta_s \leftarrow \min\{\delta_s,$ low_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                 $\delta_{s'} \leftarrow \delta_s +$ low_time($s'$,$s$)
19                 DECREASE_KEY($Q$, $s'$, $\delta_{s'}$)
20     return $T$
```

As shown in Algorithm 3, 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 $s \in S \setminus T$ 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 `low_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.

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 2 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. *Algorithm 3 computes the set of states of a DTG which satisfy a given TCTL$_\leq$ property* $\mathbf{E}(\Phi\, \mathbf{U}^{\leq c}\, \Psi)$.

Proof. Assume that there is a state $s \in S$ which satisfies the formula $\mathbf{E}(\Phi\, \mathbf{U}^{\leq c}\, \Psi)$. As s satisfies $\mathbf{E}(\Phi\, \mathbf{U}^{\leq c}\, \Psi)$; 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, $s \in Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\leq c}\, \Psi))$ which is computed by the new algorithm. The same argument is valid for proving that if the new algorithm puts a state s in $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\leq c}\, \Psi))$, the state s satisfies the formula $\mathbf{E}(\Phi\, \mathbf{U}^{\leq c}\, \Psi)$. $\square$

3.2.2 Calculating $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi))$

As described in Section 3.1, the algorithm of finding $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi))$ is reduced to two cases. A given state $s \in S$ is in $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi))$ 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 $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geq c}\,\Psi))$ is like the approach of calculating $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\leq c}\,\Psi))$. In this case, the enumerative backward search starts from a state which has the maximum level in BFS traverse of DTG, called the *deepest state*. This state is selected because of the fact that before starting the process of a state, all of its successor states must be processed, which is guaranteed by selecting the deepest state. 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 $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geq c}\,\Psi))$.

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 `level` $: S \to \mathbb{N}$ is defined to retrieve the levels of states in BFS traverse of transition systems. Note that the value of the level of states can be associated with states during the generation of transition systems without additional cost or after that by time complexity of $O(V + E)$. 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 4.

Algorithm 4: Enumerative backward search for computing $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geq c}\,\Psi))$

```
Input: A DTG with the set of states $S$, the TCTL$_{\leq,\geq}$ formula $\mathbf{E}(\Phi\,\mathbf{U}^{\geq c}\,\Psi)$, and
       the set of states $SCC$ as the states in cycles of which all members are in
       $Sat(\Phi)$
Output: $Sat(\mathbf{E}(\Phi\,\mathbf{U}^{\geq c}\,\Psi)) = \{s \in S \mid s \models \mathbf{E}(\Phi\,\mathbf{U}^{\geq c}\,\Psi)\}$
1  begin
2      $T \leftarrow \emptyset$
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($Q$, level($s$), $s$)
8      while $Q \neq \emptyset$ do
9          $(level_s, s) \leftarrow$ 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                     $\Delta \leftarrow \Delta_s +$ up_time($s'$,$s$)
16                 $\Delta_{s'} \leftarrow \max\{\Delta, \Delta_{s'}\}$
17                 PUT($Q$, level($s'$), $s'$)
18                 if $\Delta_{s'} \geq c$ then
19                     $T \leftarrow T \cup \{s'\}$
20     return $T$
```

The initialization part of Algorithm 4 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 $(\texttt{level(s)}, s)$. 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 2) 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 the case of 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. In this part of code, the function $\texttt{up\_time} : S \times S \rightarrow \mathbb{N}$ is defined to retrieve the upper bound of the associated progress of time with the transition between two given states. 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. *Algorithm 4 computes the set of states of a DTG which satisfy a given* $TCTL_{\geq}$ *property* $\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi)$.

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 $\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi)$ 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 2, 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 $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi))$. 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 $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi))$, including s.

2. Assume that $s \in S$ satisfies $\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi)$ 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, upon extracting s' from Q, the value of Δ of its predecessors is overwritten as the longest path ends to s' (lines 15 and 16 of Algorithm 4). The same argument is valid for the predecessor of s' and the other predecessors in the path from s to 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 $Sat(\mathbf{E}(\Phi\, \mathbf{U}^{\geq c}\, \Psi))$.

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

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 $TCTL_{\leq,\geq}$ 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 the model checking against $\text{TCTL}_{\leq,\geq}$ 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 [46].

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

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 $\text{TCTL}_{\leq,\geq}$ model checking algorithm in Section 3.2 can be used for the model checking of Timed Rebeca models. This fact is stated in Lemma 1.

Lemma 1. *The fine-grained transition system of a Timed Rebeca model is a DTG.*

Proof. For a given Timed Rebeca model $\mathcal{M}$, $TS_{\mathcal{M}}$ is transformed to its equivalent DTG ($DTG_{\mathcal{M}}$) using mapping of actions. This mapping function associates zero with taking-message and internal transitions and associates an interval with each progress-of-time transition. Note that as one value is associated with each progress-of-time transition of $TS_{\mathcal{M}}$, the time interval which is associated with its corresponding transition in $DTG_{\mathcal{M}}$ has tight bounds which are the same as the value of the progress-of-time transition. $\square$

As a result, for a given $\text{TCTL}_{\leq,\geq}$ 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 2. *There is an $O((V \lg V + E) \cdot |\Phi|)$ algorithm for the model checking of Timed Rebeca models against $TCTL_{\leq,\geq}$ property Φ.* $\square$

3.3 Case Studies and Experimental Results

We perform four different examples in different sizes to illustrate how efficiently the improved algorithm works. The selected examples 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)*. For each example, 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.

Configuration	State Space Generation		Model Checking Time	
	#States	Time	Old Algorithm	Improved Algorithm
3 Packets	442	1s	<1s	<1s
4 Packets	1,239	2s	6s	<1s
5 Packets	3,117	7s	2.8m	<1s
6 Packets	9,907	35s	40m	1s
7 Packets	35,746	6.8m	>5h†	5s
8 Packets	136,666	1.4h	>5h†	16s

Table 3.1: The size of 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).

Configuration	State Space Generation		Model Checking Time	
	#States	Time	Old Algorithm	Improved Algorithm
1 AMs	180	<1s	<1s	<1s
2 AMs	5,506	1s	16s	<1s
3 AMs	177,989	14.5m	>5h†	18.8m

Table 3.2: The size of 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).

3.3.1 Network on Chip (NoC)

Our first example is the model of a network on chip (NoC). The overview and detailed description of this example are presented in Section 2.3.4. We created the extended version of this model and varying in the number of packets.

We model checked this model against the bounded response which is formulated as $\mathbf{E}(\texttt{r11.received} <= 2)\,\mathbf{U}^{\leq 250}(\texttt{r11.received} > 2)$. This formula makes sure that there is a path in which before passing 250 time units more than two packets are received by the router `r11`. As shown in Table 3.1, sending 7 or 8 packets results in passing the time limit of the model checking (we set it to 5 hours) in the case of using the old model checking algorithm. However, the new algorithm computes the results in a reasonable time.

3.3.2 Hadoop YARN Scheduler

Hadoop YARN scheduler is the second example of this section which its overview and detailed description of this example are presented in Section 2.3.5. We created the extended version of this model and varying in the number of application masters.

We used $\mathbf{E}(\texttt{am2.doneJobs} <= 4)\,\mathbf{U}^{\leq 10}(\texttt{am2.doneJobs} > 4)$ formula for the model checking of the Yarn model. This formula makes sure that there is a path in which 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 3.2, 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.

Configuration	State Space Generation		Model Checking Time	
	#States	Time	Old Algorithm	Improved Algorithm
2 Customers	77	<1s	<1s	<1s
3 Customers	360	<1s	<1s	<1s
4 Customers	1,825	<1s	1s	1s
5 Customers	10,708	6s	2s	1s
6 Customers	73,461	3.4m	2.2m	1.7m

Table 3.3: The size of state spaces and the gained reductions in the Ticket Service example with different numbers of customers.

3.3.3 Ticket Service

Our third example is the model of a *Ticket Service* system. The overview and detailed description of this example are presented in Section 2.3.1. We created the extended version of this model and varying in the number of customers.

Making sure about the upper bound of the end-to-end 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.

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

Note that this formula has to be transformed into the base form which only contains existential until modalities using $\mathbf{AG}^{\leq c}\phi \equiv \neg\,\mathbf{EF}^{\leq c}\neg\phi \equiv \mathbf{E}\,\mathit{true}\,\mathbf{U}^{\leq c}\neg\phi$ and $\mathbf{AF}^{\leq c}\phi \equiv \neg\,\mathbf{E}\neg\phi\,\mathbf{U}^{\geq c}\mathit{true} \wedge \neg\,\mathbf{E}\neg\phi\,\mathbf{U}P_{scc^0(\neg\phi)}$. 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 $\mathbf{AF}^{\leq c}\phi \equiv \neg\,\mathbf{E}\neg\phi\,\mathbf{U}^{\geq c}\,\mathit{true}$.

The numbers of Table 3.3 shows that both of the algorithms perform model checking in a reasonable time. However, the newly proposed algorithm 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.

3.3.4 WSAN Applications

As the fourth example, we present the WASN model. The overview and detailed description of this example are presented in Section 2.3.6. We created the extended version of this model and varying in the timing and configuration of the model.

Checking for utilizing the communication channel in every 50 units of time is the property we used for the model checking of this example with different configurations. This property can be formulated in TCTL in the safety like formula $\mathbf{AG}^{\leq 50}(\,\mathbf{A}(\texttt{freeChannel})\,\mathbf{U}^{\leq 50}(\neg\texttt{freeChannel})\,)$, which has to be transformed to the base forms, as we did in the previous example. We verified the WSAN application in

Configuration	State Space Generation		Model Checking Time	
	#States	Time	Old Algorithm	Improved Algorithm
25-5-3-10	1,741	<1s	<1s	<1s
33-6-4-2	1,934	<1s	<1s	<1s
25-5-4-10	3,718	1s	<1s	<1s
30-6-4-2	9,353	1s	<1s	<1s
25-6-4-2	34,503	2s	<1s	<1s
20-6-4-2	57,621	3s	<1s	<1s

Table 3.4: The size of state spaces and the gained reductions in WSAN example with different configuration.

different configurations, varying the value of the sampling rate, the number of nodes, the packet size, and the sensor task delay.

The results of these experiments are depicted in Table 3.4. In each row, the configuration (the numbers which are separated by a dash) is the 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 3.4, the time consumption of the model checking is less than one second for all cases and changing the configuration of the model does not result in the generation of large state spaces.

Chapter 4

State Space Reduction by Folding Transitions[1]

In this chapter, we propose a reduction technique, called "Folding Instantaneous Transitions", to make the model checking of object-based models against TCTL$_{\leq,\geq}$ cheaper. Using this reduction technique, we will propose an approach for the model checking of TCTL$_{=}$ properties in polynomial time. As mentioned before, having instantaneous transitions, the problem of model checking against TCTL$_{=}$ properties is reducible to the Subset Sum problem which is well-known to be NP-complete [45]. By eliminating instantaneous transitions, the resulting transition system 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, the time complexity of model checking is reduced to $O((V \lg V + E) \cdot |\Phi|)$.

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). Applying this technique, a new transition system is created, called folded timed transition system (FTS). The details of eliminating instantaneous transitions is presented in the following section.

4.1 Folding Instantaneous Transitions

Folding instantaneous transitions is a reduction technique that eliminates all instantaneous transitions as well as all transient states from DTGs. Note that in the models of 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*). There is a transition between two states of an FTS if and only if the two states are consecutive progress-of-time states in their corresponding DTG. Figure 4.1 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. As shown in the figure, if the instantaneous transitions branch into a set of paths that have later observable differences, there is a nondeter-

[1]This chapter is an improvement and extension of the results published in [36].

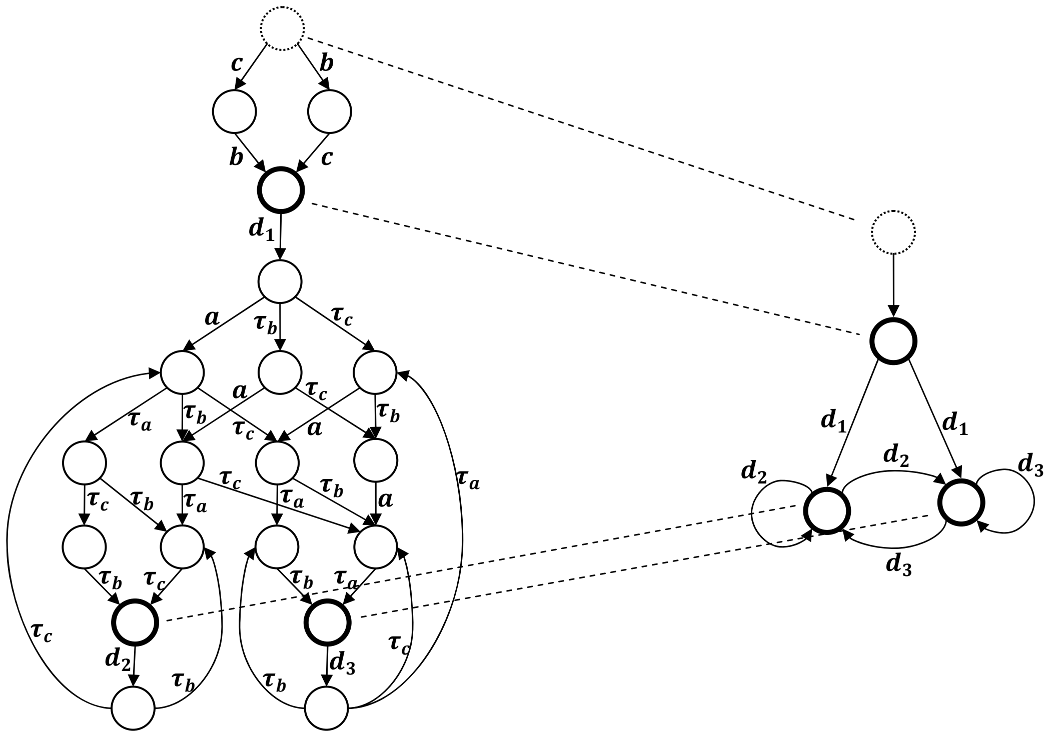


Figure 4.1: Example of how folding instantaneous transitions reduction works

ministic choice in the outgoing observable ancestor of its corresponding FTS (outgoing transitions with d_1 and d_2 labels in the right side figure).

Note that the result of folding instantaneous transitions is not in the bisimulation relation with its corresponding DTG; so, there is no guarantee for preserving the result of model checking against all properties on FTSs. This is because of the fact that this approach eliminates transient states from the transition system regardless of the values of their atomic propositions and branching points. But, in some cases where modelers prefer to model check properties based on the observable behaviors of systems, folding instantaneous transition technique can be used safely. This preference is widely considered in the object-oriented paradigm. Meyer in [56] said that object instances satisfy properties in all "stable" times. Then he defined it as "Stable times are those in which the instance is in an observable state". He mentioned that the time of the instance creation and after/before method calls are observable states of objects. Considering this preference in the verification of Timed Rebeca models, we have to defined stable times and observable states. The definition of observable states in Timed Rebeca is different from Meyer's definition. In Timed Rebeca, observable states are progress-of-time states as they are the only states in which systems are allowed to stay. So, although folding instantaneous transitions eliminates some transient states, it can be used for the analysis of Timed Rebeca models which considers the observable behaviors of actors.

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 7 (Nearest Progress-of-Time States). *For a given model $\mathcal{M}$ and $DTG_{\mathcal{M}} = (S, s_0, \to, AP, L)$ and two states $s, s' \in S$, s' is in $npts(s)$ if and only if s' is a progress-*

of-time state and for all 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. □

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

Definition 8 (Folded Transition System). *For a given $DTG_{\mathcal{M}} = (S, s_0, \dashrightarrow, AP, L)$, its corresponding folded transition system is defined as the tuple $FTS(DTG_{\mathcal{M}}) = (S', s_0, \hookrightarrow, AP, L)$, where:*

- *$S' \subseteq S$ 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 elapsed 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 an FTS can be assumed as the subset of its corresponding DTG, an FTS can be model checked against TCTL$_{\leq,\geq}$ properties using the previously proposed algorithm.

Corollary 3. *The FTS of a given DTG can be model checked against the TCTL$_{\leq,\geq}$ property Φ in $O((V \lg V + E) \cdot |\Phi|)$.*

4.2 Complete TCTL Model Checking of DTGs

In addition to reducing the size of state spaces, here, we show that the approach of [57] can be used for efficient model checking of 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 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) the weight of the path, is an NP-complete problem (using a reduction from finding the EPL between two states to the subset-sum problem [58]); so, there is no known polynomial time algorithm for solving the EPL problem. Based on this fact, the authors in [45] 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 TCTL$_{\leq,\geq}$ subset can be model checked in polynomial time.

On the other hand, as discussed in [57], 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 elapsed 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 2. *There is an $O((V + E) \cdot |\Phi|)$ algorithm for model checking of FTSs 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 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. *An FTS can be model checked against a TCTL property Φ with small constant time quantifiers in the time complexity order of $O((V \lg V + E) \cdot |\Phi|)$.*

Proof. This follows directly from Corollary 3 and Lemma 2. □

4.3 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 how the FTSs of Timed Rebeca models are generated without a significant run-time 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.

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

Algorithm 5: The state space is generated for the given Timed Rebeca model or `null` is returned in the case of Zeno behavior in the model.

```
Input: A Timed Rebeca model $\mathcal{M}$, a labling function $L$, a set of atomic
       propositions $AP$
Output: The result FTS or null if the model has Zeno behavior
1  $V \leftarrow \{$GENERATE_INITIAL_STATE($\mathcal{M}$)$\}$        ▶ The set of all of the states
2  $hasZeno \leftarrow false$                          ▶ Flag for Zeno behavior detection
3  begin
4      $S \leftarrow \{$GENERATE_INITIAL_STATE($\mathcal{M}$)$\}$       ▶ The set of the states of
       result FTS
5      $N \leftarrow$ ENQUEUE($s_0$)                     ▶ The set of the next level states
6      $\hookrightarrow \leftarrow \emptyset$                              ▶ The set of the transitions of FTS
7      while HAS_ELEMENTS($N$) $\wedge \neg hasZeno$ do
8          $s \leftarrow$ DEQUEUE($N$)
9          $N' \leftarrow$ Bounded_DFS($s$)
10         foreach state $s' \in N$ do
11             $time \leftarrow$ PROGRESS_OF_TIME($s$)
12             $S \leftarrow S \cup s'$
13             $\hookrightarrow \leftarrow \hookrightarrow \cup (s, time, s')$
14         $N \leftarrow$ ENQUEUE_ALL($N'$)
15     if $hasZeno = true$ then
16         return null
17     else
18         return $(S, s_0, \hookrightarrow, AP, L)$
19 Procedure $Bounded\_DFS$(s)
20     $Q \leftarrow$ GENERATE_SUCCESSOR_STATES($s$)
21     $R \leftarrow \emptyset$                              ▶ The set of states of npts($s$)
22     foreach state $s' \in Q$ do
23         if IS_PROGRESS_OF_TIME($s'$) then
24             $R \leftarrow R \cup s'$
25             continue
26         else
27             if $s' \notin V$ then
28                 $V \leftarrow V \cup s'$
29                 recStack($s'$) $\leftarrow true$
30                 $R \leftarrow R \cup$ Bounded_DFS($s'$)
31                 recStack($s'$) $\leftarrow false$
32             else
33                 if recStack($s'$) $= true \wedge$ now($s'$) $=$ now($s$) then
34                     $hasZeno \leftarrow true$
35     return $R$
```

recursive procedure, defined in lines 19 to 35. In each round, if a progress-of-time state is found, it is put in the 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 $O(V+E)$.* $\square$

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

4.4 Case Studies and Experimental Results

To illustrate the applicability of the proposed reduction technique, we apply it to the examples of Section 3.3. We choose the state space size and the model checking time consumptions as the performance metrics. The values of these metrics are compared in a table for each case study. In the tables, *Original* is used to refer to the original state spaces and *Reduced* is used to refer to the reduced state space (i.e., FTSs). As the reduction technique applied on-the-fly without a significant overhead, we do not report the spent time for the state space generation. Note that the property specifications of the examples are the same as that of in the previous chapter.

4.4.1 Network on Chip (NoC)

Our first example is a model of a network on chip (NoC), a promising architecture paradigm for many-core systems. The overview and detailed description of this example are presented in Section 2.3.4.

The effect of applying the reduction technique is shown in Table 4.1. In the NoC model, increasing the number of sent packets results in a slight increment in the gained reduction, which is because of the increment of the concurrency level of the model. In other words, increasing the number of packets results in the 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 4.1 also shows that TCTL model checking on the reduced transition system results in the model checking of the models in less than a second.

Configuration	State Space Generation			Model Checking Time	
	#States Orig.	#States Red.	Gain	Orig.	Red.
3 Packets	442	68	84%	<1s	<1s
4 Packets	1,239	122	91%	<1s	<1s
5 Packets	3,117	126	96%	<1s	<1s
6 Packets	9,907	129	98%	1s	<1s
7 Packets	35,746	102	99%	5s	<1s
8 Packets	136,666	117	99%	16s	<1s

Table 4.1: The size of state spaces and the gained reductions in the NoC example in different scenarios

Configuration	State Space Generation			Model Checking Time	
	#States Orig.	#States Red.	Gain	Orig.	Red.
1 AMs	180	56	69%	<1s	<1s
2 AMs	5,506	1,283	77%	<1s	<1s
3 AMs	177,989	24,639	86%	18.8m	<1s

Table 4.2: The size of state spaces and the gained reductions in the Hadoop Yarn example with default configuration

4.4.2 Hadoop YARN Scheduler

Hadoop [50] is a framework for MapReduce, a programming model for generating and processing large data sets, selected as the second example. The overview and detailed description of this example are presented in Section 2.3.5.

The same as the model of NoC, applying FTS technique reduces the size of state spaces significantly, as shown in Table 4.2. 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. Table 4.2 also shows that the time is reduced to less than one second when the FTS technique is applied.

4.4.3 Ticket Service

Our third example is the model of a *Ticket Service* system. The overview and detailed description of this example are presented in Section 2.3.1.

Table 4.3 shows that applying FTS technique improves the performance of the model checking. The same as the previous examples, applying FTS technique reduces the size of state spaces significantly and increasing the number of the customers increases the gained reduction.

4.4.4 WSAN Applications

As the fourth example, we present a realtime data acquisition system for structural health monitoring and control (SHMC) of civil infrastructures. The overview and detailed description of this example are presented in Section 2.3.6.

Configuration	State Space Generation			Model Checking Time	
	#States Orig.	#States Red.	Gain	Orig.	Red.
2 Customers	77	10	87%	<1s	<1s
3 Customers	360	40	89%	<1s	<1s
4 Customers	1,825	184	90%	1s	<1s
5 Customers	10,708	1,047	90%	1s	<1s
6 Customers	73,461	6,997	91%	1.7m	1s

Table 4.3: The size of state spaces and the gained reductions in the Ticket Service example with different numbers of customers

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

Table 4.4: The size of state spaces and the gained reductions in WSAN example with different configuration

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 these experiments are depicted in Table 4.4. In each row, the configuration (the numbers which are separated by dashs) 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 4.4, 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.

Chapter 5

Big-Step Semantics of Timed Rebeca[1]

In the previous chapters, we described the standard semantics of Timed Rebeca in the form of timed transition system and showed how it can be used for model checking against TCTL properties. Although this semantics provides TCTL model checking facilities, unfortunately, it suffers from the usual state space explosion problem. The transition system contains arbitrary interleavings of independent actions of the various components of a distributed system, resulting in a large state space. In the presence of a global clock and timing information, this may become even more common.

In this chapter, we propose a very different semantics, called Floating Time Transition System (FTTS), as a big-step semantics of timed actor-based models. States in a FTTS contain the local time of each actor, in addition to values of their state variables and the bag of their received messages. However, the local times of actors in a state can be different, and there is no unique value for time in each state. Such a semantics is reasonable when one is only interested in the order of visible events. FTTS may not be appropriate for analyses that require reasoning about all synchronized global states of a Timed Rebeca model. The key features of Rebeca actors that make FTTS a reasonable semantics are having no shared variables, no blocking send or receive, single-threaded actors, and atomic (non-preemptive) execution of each message server which gives us an isolated message server execution. This means that the execution of a message server of an actor will not interfere with the execution of a message server of another actor. Therefore, we can execute all the statements of a given message server (even delay statements) during a single transition. This makes the transition system significantly smaller because there will be only one kind of action, which is taking a message and executing the corresponding message server entirely.

For timed systems, the norm is to show that there is a timed weak bisimulation relation between two timed transition systems to prove that they preserve the same set of timed branching-time properties (e.g. TCTL). Proving the existence of such a relation is impossible when one of the transition systems does not have progress-of-time transitions, which is the case of the relation between FGTS and FTTS. Here, we will prove that the actions and the execution time of the actions are preserved in FTTS using an innovative approach for defining a relation between the states of a FGTS and its corresponding FTTS. Since the starting time of the execution of actions is also preserved, we can prove the preservation of any timed property of actions that is bisimulation invariant. Examples of such properties include μ-calculus with weak modalities.

[1]This chapter is an improvement and extension of the results published in [59] and [2].

Our bisimulation proof relies on observing that the FTTS semantics exploits key features of the actor model of computation. In such a model there is no shared memory, and sends and receives are non-blocking. Moreover, actors are single-threaded, with message servers being executed non-preemptively. This means that message servers can be executed in an isolated fashion, as is carried out in FTTS, without compromising the semantics of the model. Since our correctness proof of FTTS relies only on certain features of the actor model (rather than something specific to timed Rebeca), it suggests that FTTSs can be used in the analysis of other actor models and languages, and more generally, in other asynchronous event-based models.

5.1 Semantics of Timed Rebeca in FTTS

In this section, we present the big-step semantics of Timed Rebeca in FTTS. The big-step semantics of Timed Rebeca in FTTS can be defined in terms of a transition system, as shown in the following.

Definition 9. *For a given Timed Rebeca model $\mathcal{M}$, FTTS $= (S, s_0, Act', \hookrightarrow, AP, L)$ is its floating time semantics where S is the set of states, s_0 is the initial state, Act' is the set of actions, $\hookrightarrow \subseteq S \times Act' \times S$ is the transition relation, AP is the set of atomic propositions, and $L : S \to 2^{AP}$ is the labeling function, described as the following.*

- *The global state of a Timed Rebeca model $s \in S$ in FTTS is the function $s : AID \to (Var \to Val) \times \mathcal{P}(Msg^*) \times Stat^* \times \mathbb{N} \times \mathbb{N} \cup \{\epsilon\}$, which is the same as the definition of states in the fine-grained transition system of Timed Rebeca. In comparison with the fined-grained transition system, the values of remaining statements and resuming time are set to ϵ for all actors in the floating time transition system. In addition, there is no guarantee for the local times of actors to be the same. As a result, actors in the floating time transition system are in the form of $(v, q, \epsilon, t, \epsilon)$.*

- *In the initial state of the model, for all of the actors, the values of state variables and content of the actor's message bag are set based on the statements of its constructor method, and the remaining statements is set to ϵ. The local times of the actors are set to zero and their resuming times are set to ϵ.*

- *The set of actions is defined as $Act' = MName$.*

- *The transition relation $\hookrightarrow \subseteq S \times Act' \times S$ defines the transitions between states that occur as the results of actors' activities including: taking a message from the message box, executing all of the statements of its corresponding transition system. For proposing the formal definition of $\hookrightarrow$, we have to define the notion of idle actors. An actor in the state (v, q, ϵ, t, r) is called idle, i.e., it is not in the middle of executing a message. A given state s is idle, if $s(x)$ is idle for every actor x. We use the notation $idle(s, x)$ to denote the actor identified by x is idle in state s, and $idle(s)$ to denote s is idle. Using these definitions, two states $s, s' \in S$ are in relation $s \xhookrightarrow{mg} s'$ if and only if the following conditions hold.*

    - $idle(s) \wedge idle(s')$, *and*
    - $\exists s_1, s_2, \cdots, s_n \in S, x \in AID \cdot s \xrightarrow{mg} s_1 \to \cdots \to s_n \to s' \wedge \forall y \in AID/\{x\},\ 1 \le i \le n \cdot \neg idle(s_i, x) \wedge idle(s_i, y)$

$$\frac{s(x) = (v, q, \langle \mathbf{delay}(e)|\sigma\rangle, t, r) \wedge r = t}{s \xrightarrow{\tau} s[x \mapsto (v, q, \sigma, t + eval_v(e), r + eval_v(e))]} \quad \mathbf{(delay)}$$

Note that in the big-step semantics, the delay statements is modified to increase the local time of actors, in addition to their resuming time, shown below.

- *AP contains the name of all of atomic propositions.*
- *Function $L : S \to 2^{AP}$ associates a set of atomic propositions with each state, shown by $L(s)$ for a given state s.*

□

The same as Section 2.2, we illustrate how FTTS is created for the ping pong example of Listing 2.3 in Figure 5.1. As shown in the detailed contents of the first and second states, execution of the message server `ping` results in progress in the local time of `pi` by 2 units and putting the `pong` message in the bag of `po`. Note that the local time of `po` in this state is 0 as it does not execute any messages server. Performing transition from the second state to the third one, the local time of `po` is increased to 1 as the delay statement in the message server `pong` is executed. Also, a message is sent to `pi` which is put in the bag of `pi` with release time for 2.

As there is no difference between the structure of states in FGTS and FTTS, the notion of shift-equivalence relation works for making FTTSs finite.

5.2 An Action-Based Weak Bisimulation between the Two Semantics

As described in Section 5.1, in FTTS representation of a Timed Rebeca model, all the statements of a message server are executed at once during a $\hookrightarrow$ transition. In contrast, the fine-grained semantics executes one statement at a time and interleaves the execution of different message servers. We demonstrate despite these differences, this two semantics are equivalent in some sense. To this end, we define an action-based weak bisimulation (observational equivalence) relation between $FGTS = (S, s_0, Act, \to, AP, L)$ and $FTTS = (S, s_0, Act', \hookrightarrow, AP, L)$ for a given Timed Rebeca model $\mathcal{M}$. Note that this relation exists for Timed Rebeca models which do not have nondeterministic assignments.

Prior to the formal definition of the relation between the states of FTTS and FGTS the following definitions and proposition are required to make the relation easy to understand.

We begin by defining the observable and τ actions in both transitions systems. All actions in FTTSs are observable. In FGTSs, only *taking-message transitions* are observable. Therefore, *time transitions* and *internal transitions* in FGTSs are assumed to be τ transitions. In other words, only taking-message actions are observable in FGTSs and FTTSs. This definition conforms the definition of *events* and *observer primitives* in the actor model which is introduced by Agha et. al. in [60] as a reference actor framework.

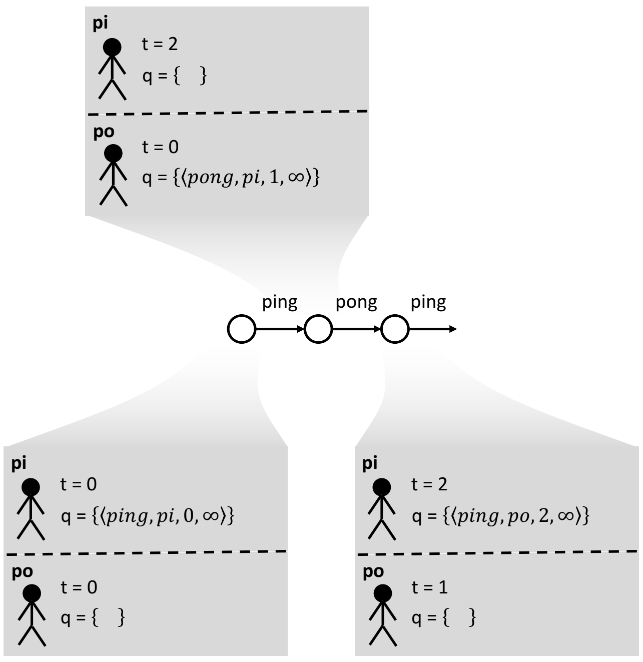


Figure 5.1: The beginning part of the FTTS of the ping pong example

At the second step, we define the function $sent : S \to \mathcal{P}_{\mathbb{N}}(Msg)$ to capture the already sent message from its given actor in a given state, as presented by $sent(s, x) = \{(ac, mg, pr, ar, dl) \mid \exists y \in AID \cdot s(y) = (v_y, q_y, \sigma_y, t_y, r_y) \wedge (ac, mg, pr, ar, dl) \in q_y \wedge ac = x\}$.

Next, we define the notion of a *completing trace* for an actor x in FGTS state s as an execution which results in completing the execution of the message server of x that has already commenced in state s. Note that during a completing trace for x the other actors, may complete their message servers (or not), and may start the execution of new message servers. We begin by first defining an execution trace.

Definition 10 (Execution Trace). *An execution trace from the state s in FGTS is a sequence of transitions from s to one of its reachable states s', shown by $s \xrightarrow{act_1} s_1 \xrightarrow{act_2} \cdots \xrightarrow{act_n} s'$.* □

Definition 11 (Completing Trace for an Actor). *A given execution trace $s \xrightarrow{act_1} s_1 \xrightarrow{act_2} \cdots s_n \xrightarrow{act_n} s'$ from the state $s \in S$ to state $s' \in S$ in FGTS is a completing trace for the actor x if and only if the following conditions hold:*

- *The execution of currently executing message server of x is finished in s', i.e. $s'(x) = (v'_x, q'_x, \epsilon, t'_x, \epsilon)$.*
- *There is no other state in the trace where the execution of currently executing message server of x is finished, i.e. ,$\forall i \in [1, n] \cdot s_i(x) = (v_{i_x}, q_{i_x}, \sigma_{i_x}, t_{i_x}, r_{i_x}) \to \sigma_{i_x} \neq \epsilon$.*

This way, we make sure that x finishes the execution of its currently executing message server and it does not execute any taking-message transition from s to s'.

Here, we also define $CT(s, x)$ as a function which returns one of the completing traces of the actor x from the state s (no matter which one in the case there are more than one completing trace from s for the actor x) and CT as a set of all completing *traces from all of the state of S. In the case of $s(x) = (v_s, q_s, \epsilon, t_s, \epsilon)$, there is $CT(s, x) = \epsilon$ as no more action is needed for completing the execution of a message server of x in s.* □

Note that as there is no preemption in the message server execution and there is no infinite message server body in Timed Rebeca, there is a completing trace for all the actors from all of the states.

We define three functions on the completing traces. The first one returns the valuation function of the state variables of the specific actor at the last state of the trace (the actor that the completing trace is defined for). The second one returns the time of the last state of the trace. The third one returns the bag of messages that are sent by the specific actor during the execution of its currently executing message server. Note that for a give transition $(s, \tau_{send}, s') \in\rightarrow$, we defined the method *sent* which returns the message which is sent by τ_{send} in transition from s to s'.

Definition 12 (Three Functions on completing traces)**.** *Three function statevars : $CT \rightarrow (Var \rightarrow Val)$, now : $CT \rightarrow \mathbb{N}$, sent : $CT \rightarrow Msg$ is defined as the following. In the following we assumed that completing trace $CT(s, x) = s \xrightarrow{act_1} s_1 \xrightarrow{act_2} \cdots \xrightarrow{act_n} s'$ from the state s for the actor x where $s'(x) = (v'_x, q'_x, \epsilon, t'_x, \epsilon)$ is given.*

- *$statevars(CT(s, x))$ returns the valuation function of state variables of x in the target state of the completing trace $CT(s, x)$, i.e., $statevars(CT(s, x)) = v'_x$.*
- *$now(CT(s, x))$ returns the local time of x in the target state of the completing trace $CT(s, x)$, i.e., $now(CT(s, x)) = t'_x$.*
- *$sent(CT(s, x))$ returns a bag of messages which are sent by actions of $CT(s, x)$, i.e., $sent(CT(s, x)) = \{(ac, mg, pr, ar, dl) | \forall j \in [1, n] \cdot act_j = \tau_{send} \wedge sent((s_j, act_j, s_{j+1})) = (ac, mg, pr, ar, dl)\}$.*

□

Based on the isolated execution of actors (there is no shared variable and no preemption in the execution of message servers) we can easily conclude that in case of having more than one completing trace for an actor, any of the completing traces ends in the same values for the state variables, the same local time, and the same bag of sent messages.

Proposition 1 (Completing Traces End in the Same Final Condition)**.** *Assume that there are two different completing traces ct_1 and ct_2 from a given state $s \in S$ and actor x. There are $sent(ct_1) = sent(ct_2)$, $now(ct_1) = now(ct_2)$, and $statevars(ct_1) = statevars(ct_2)$.*

Proof. As mentioned in the semantics of Timed Rebeca, execution of a message server is not interfered with the execution of message servers of the other actors as in Timed Rebeca there is no shared variable or any kind of preemption of execution of message servers. In addition, we assumed that there is no nondeterministic expression in messages servers of actors. Therefore, in all the completing traces from state s, execution of τ transitions which are related to the actor x ends in the same values for state variables and bag of sent messages. On the other hand, as *delay* statements which are related to the execution of the message server of x are the same in two different completing traces, the time at the target states of ct_1 and ct_2 are the same. □

Note that this proposition is valid when there is no nondeterminism in the body of message servers. At the beginning of this section, we made clear that in this work we address Timed Rebeca models which do not have nondeterministic assignments.

Next, we define a projection function for states of FGTS and FTTS. Projection functions extract values of state variables and the collection of messages which are sent by one actor from a given FGTS or FTTS state. Using these projection functions, we get uniform views from states of FGTS and FTTS which are necessary for the definition of the action-based weak bisimulation relation. To this end, as the execution of a message in FGTS is completed in several steps, the projection function in FGTS is defined based on completing traces to be able to have access to the valuation of state variables and bags of sent messages after completing the execution of currently executing messages.

Definition 13 (Projection Function in FGTSs)**.** *For a given FGTS state $s \in S$ and actor x the function $Proj(s,x)$ returns a collection of $statevars(CT(s,x))$, $now(CT(s,x))$, and $sent(s,x) \cup sent(CT(s,x))$.* □

Directly from the above definition, it is concluded that the projection function in FGTSs returns similar data for two consequent states which their transition is not a *taking-message* transition.

Proposition 2 (Only taking-message transitions change the result of the projection functions)**.** *Considering two states $s, u \in S$ such that there is $s \xrightarrow{act} u$ and act is not a taking-message action, there is $\forall x \in AID \cdot Proj(s,x) = Proj(u,x)$.*

Proof. Proof by contradiction. Assume that *act* is not a taking event-action and there is an actor $x \in AID \cdot Proj(s,x) \neq Proj(u,x)$. Also, assume that $s(x) = (v_x, q_x, \sigma_x, t_x, r_x)$. In this case, as transitions of actors do not affect the parts of the other actors which are related to the projection function, *act* belongs to x and consequently $\sigma_x \neq \epsilon$ and $r_x \neq \epsilon$. Note that *act* cannot be a progress-of-time as it does not affect the parts of the other actors which are related to the projection function results in $Proj(s,x) = Proj(u,x)$. Based on the definition completing traces, one of the completing traces of x from s contains u, as x can finish the execution of the statements of σ_x continuing the execution from u. As a result, the valuation of the state variables, the set of sent messages, and the time of actors at the target state of the completing trace of s and u are the same, which is in contradiction with the assumption of $Proj(s,x) \neq Proj(s,u)$. □

In contrast to FGTSs, the execution of a message in FTTSs is completed in one step; therefore, the projection function in FTTSs is defined based on the current content of states.

Definition 14 (Projection Function in FTTSs)**.** *For a given FTTS state $s \in S$ and actor x where $s(x) = (v_x, q_x, \epsilon, t_x, \epsilon)$,the projection function $Proj'(s,x)$ returns a collection of v_x, t_x, and $sent(s,x)$.* □

Using the above definitions, we define the action-based weak bisimulation relation among states of FGTS and FTTS. Two states in FGTS and FTTS are in an action-based weak bisimulation relation if and only if the projections of states according to all actors are the same. This way, we will prove that two states have the same future behavior in Theorem 5. Figure 5.2 shows how states in a FGTS are mapped to their

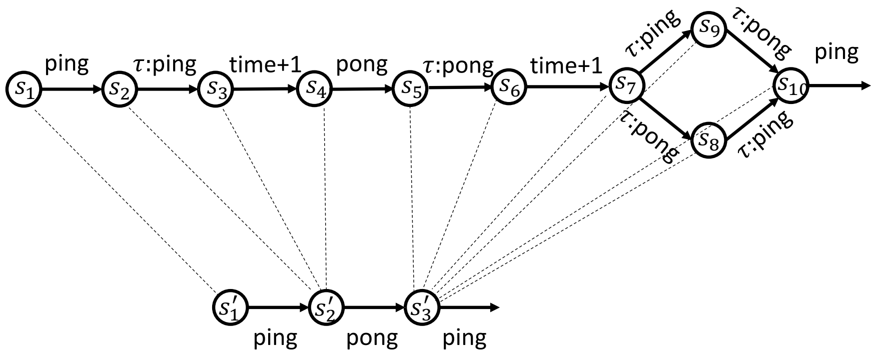


Figure 5.2: How states in FGTS (at the top of the picture) are mapped to their corresponding states in FTTS (at the bottom of the picture) which have the same future behaviors, for the ping pong example of Listing 2.3

corresponding states in its FTTS, for the transition systems of the ping pong example of Listing 2.3. As the observational behavior of s_1 and s'_1 are the same (only the observable action *ping* is enabled), s_1 is mapped to s'_1. The observational behavior of s_2, s_3, and s_4 are the same as the observational behavior of s'_2 as the observable action *pong* will appear after them, and so on.

Definition 15 (Relation among States of FGTS and FTTS)**.** *A FGTS state $s \in S$ and an FTTS state $s' \in S$ are in relation $\mathcal{R} \subseteq S \times S$ if and only if $Proj(s,x) = Proj'(s',x)$ holds for every actor x.* □

Directly from the definition of the relation $\mathcal{R}$ it is concluded that the bag of enabled messages in s and s' are the same, i.e they contain messages with the same signature and the same execution time and their corresponding actors can take them to start to execute (actors are not busy with executing other messages).

Proposition 3 (Relation $\mathcal{R}$ preserves enabled messages)**.** *A FGTS state $s \in S$ and an FTTS state $s' \in S$ which are in the relation $\mathcal{R}$ have the same bag of enabled messages.*

Proof. Based on the fact that $s\ \mathcal{R}\ s'$ it is concluded that $\forall\ x \in AID \cdot Proj(s,x) = Proj'(s',x)$ and $\forall\ x \in AID \cdot sent(s,x) \cup sent(CT(s,x)) = sent(s',x)$, results in $\bigcup_{x\in AID} (sent(s,x) \cup sent(CT(s,x))) = \bigcup_{x\in AID} (sent(s',x))$. Now, considering the message bags of the actors, the formula is rewritten to $\bigcup_{x\in AID} (\sigma_x \cup sent(CT(s,x))) = \bigcup_{x\in AID} (\sigma'_x)$ where σ_x is the bag of messages for actor x in the state s and σ'_x is the bag of messages for actor x in the state s'. As the messages in $\bigcup_{x\in AID} sent(CT(s,x))$ will be send in the future, none of the enabled messages in s are in $\bigcup_{x\in AID} sent(CT(s,x))$. Therefore, enabled messages in $\bigcup_{x\in AID} (\sigma'_x)$ are in $\bigcup_{x\in AID} (\sigma_x)$.

On the other hand, based on the definition of the enabled messages, the completing trace of their corresponding actors are empty traces. Assume that x is the identifier of one of the actors which have enabled messages in s. Also, assume that there are $s(x) = (v_x, q_x, \sigma_x, t_x, r_x)$ and $s'(x) = (v'_x, q'_x, \epsilon, t'_x, \epsilon)$. Having $CT(s,x) = \emptyset$ results in $now(CT(s,x)) = t_x$ which implies $t_x = t'_x$. Therefore, the taking message time of enabled messages for actors in s is the same as that of in s'. Considering this fact, there are the same enabled messages in the bags of actors in s and s' and their taking message times are the same. □

Having the same bag of enabled messages in two given states s and s' where $s\ \mathcal{R}\ s'$, we are able to prove that s and s' have the same future behavior. To this end, we have to prove that $\mathcal{R}$ is an action-based weak bisimulation relation.

Definition 16 (Action-based weak bisimulation relation)**.** *A relation $\mathcal{P}$ over two transition systems $TS_1 = (S_1, s_{1_0}, Act_1, \rightarrow, AP_1, L_1)$ and $TS_2 = (S_2, s_{2_0}, Act_2, \Rightarrow, AP_2, L_2)$ where TS_2 is τ-free transition system, is an action-based weak bisimulation relation if the following conditions hold for states of TS_1 and TS_2.*

1. *$\forall s_1, t_1 \in S_1$ and $s_2 \in S_2$ where $s_1\ \mathcal{P}\ s_2$, in the case of $s_1 \xrightarrow{\alpha} t_1$ where $\alpha \in Act_1$ then $\exists\ t_2 \in S_2$ such that $s_2 \overset{\alpha}{\Longrightarrow} t_2$ and $t_1\ \mathcal{P}\ t_2$ and in the case of $s_1 \xrightarrow{\tau} t_1$ there is $t_1\ \mathcal{P}\ s_2$.*

2. *$\forall s_2, t_2 \in S_2$ and $s_1 \in S_1$ where $s_1\ \mathcal{P}\ s_2$, for a message $\alpha \in Act_2$ such that $s_2 \overset{\alpha}{\Longrightarrow} t_2$ then $\exists\ s', s'', \ldots, s^{(k)}, t_1 \in S_1$ (for $k \geq 0$) such that $s_1 \xrightarrow{\tau} s' \xrightarrow{\tau} s'' \xrightarrow{\tau} \cdots \xrightarrow{\alpha} t_1$ and $t_1\ \mathcal{P}\ t_2$.* □

Theorem 5. *The relation $\mathcal{R}$ is an action-based weak bisimulation relation between states of FGTSs and FTTSs.*

Proof. To prove that the first condition of action-based weak bisimulation relation, we assume that there are two FGTS state $s, u \in S$ and two FTTS state $s', u' \in S$ such that $s\ \mathcal{R}\ s'$. Also, we assume that there is a transition between s and u in the form of $s \xrightarrow{act} u$. Based on the type of *act* the following two cases are possible.

The action act is a taking-message action: Based on the definition of the relation $\mathcal{R}$, in this case, projection function for all of the actors in s and u return the same value except for the sender and receiver of the message which corresponds to *act*. Assume that this message is (ac, mg, pr, ar, dl) and it is in the bag of actor x. For the sender actor ac the difference is in the bag of sent messages, results in $sent(u, ac) = sent(u, ac)/\{(ac, mg, pr, ar, dl)\}$.
For the receiver actor, there is a completing trace $CT(u, x)$ such that $Proj(u, x)$ returns the valuation of state variables of x from the target state of $CT(u, x)$ and messages which are sent by x in u in union with messages which are sent during $CT(u, x)$.
As *act* is an enabled message in s', there is $s \xrightarrow{act} s'$. So, the projection function returns valuation of state variables and the sent messages of x after the execution of all of the statements of the message server mg in u' which is the same as what projection function returns in u. Therefore, there is $u\ \mathcal{R}\ u'$ as the results of the projection functions in u and u' are the same for all of the actors.

The action act is not a taking-message action: For this case, based on the Proposition 2, u and s' are in relation $\mathcal{R}$. This way, doing a progress of time transition or internal transition from s results in stuttering in s' as one of the properties of action-based weak bisimulation relations.

To prove the second condition, as all the transitions in FTTS are taking-message transitions, *act* must be a taking-message transition. To this end, the argument which is presented above for the case of *action act is a taking-message action* can be used.

Finally, we have to show that the initial states of the transitions systems are in the relation $\mathcal{R}$. As the program counter of all of the actors in s_0 is set to ϵ, the completing traces started from s_0 are ϵ. Therefore, the valuation of the state variables, sent messages, and time of the actors is computed based on the values in s_0 which is the same as their corresponding values in s'_0, results in $\forall x \in AID \cdot Proj(s_0, x) = Proj'(s'_0, x)$. $\square$

We discussed in Section 5.1 that in actor systems we are interested in relations among actions of systems and the time where they are triggered (messages are taken from bags). So, we have to find the most expressive action-based logic which is preserved in action-based weak bisimulation relation. As mentioned in [61], weak bisimulation relation preserves properties in form of modal μ-calculus with weak modalities. Weak-bisimulation relation does not preserve complete modal μ-calculus. Weak modal μ-calculus has the same syntax as modal μ-calculus, where we assume that the diamond ($\langle a \rangle \varphi$) and box ($[a]\varphi$) modalities are restricted to observable transitions, i.e., action a must be a taking-message transition. The semantics of this logic is identical to that of μ-calculus, except for the semantics of the diamond and box operators — a state s satisfies $\langle a \rangle \varphi$ if there is an execution starting from state s to t, such that a is the only visible action, and t satisfies (inductively) φ. The semantics of box is defined dually.

Corollary 5. *Transition systems of Timed Rebeca models in FGTS and FTTS are equivalent with respect to all formulas that can be expressed in modal μ-calculus with weak modalities where the actions are taking messages from bags.* $\square$

5.3 Comparing to the Other Reduction Technique

Here, we give an overview of the reduction techniques which are proposed for the realtime systems and illustrate differences between FTTS and those techniques.

Partial Order Reduction. The reduction from FGTS to FTTS has aspects that are similar to partial order reduction (POR). In fact, the relationship between POR and FTTS is subtle. FTTS is unaware of any independence relation, persistence/ample sets for timed actor systems that will result in POR techniques producing FTTS as the reduced transition system. Moreover, not only is the formal relationship between FTTS and POR nontrivial, POR techniques for timed systems were empirically compared against the FTTS semantics and found that the FTTS results in smaller transition systems.

Timed Automata. Modeling of realtime distributed systems with asynchronous message passing between components using synchronous communication of automata increases the number of states dramatically (because of many synchronizations among automata for model asynchronous behavior).

We can apply some techniques, like using *committed states*, to reduce the number of states of the resulting region transition system; although, we still need points of synchronization. During the parallel composition of the network of timed automata, related to each component, different automata need to synchronize on the following four actions.

1. A message is sent,
2. A message is taken from the message bag to start to execute,
3. A transition modeling passage of time,
4. It is the time for a sent message to be delivered to its receiver.

Therefore, messages of each component cannot be processed in one step. The execution of each message must be divided into some parts when one of the following synchronizations are required.

To reduce the number of states some reduction techniques are proposed for verification of timed automata models. In [32] authors proved that instead of a global clock synchronization among all timed automata in a network of timed automata, the synchronization of clocks is only required in communication between two timed automata. Therefore, they allowed clocks to increase independently and they only synchronize the clocks when two timed automata want to communicate. This way, the third item of the synchronization actions (see above) is omitted and other three synchronization points are considered in the parallel composition of timed automata. This work is continued in [33] by applying the proposed reduction technique in model checking of timed extension of LTL. In FTTS, instead of four synchronization points which are required for generating region transition system (or three synchronization points in case of using reductions of [32] and [33]), one synchronization point is required. This synchronization point is on the release time of messages. Therefore, using FTTS requires less number of synchronizations, which results in executing each message in one step.

5.4 Experimental Results

We extended Afra to support model checking of Timed Rebeca models based on the semantics of FTTS[2]. The current version of the model checking toolset supports schedulability and deadlock-freedom analysis and assertion-based verification of Timed Rebeca models using FTTS. We provide four case studies of different sizes to illustrate the reduction in state space size, number of transitions, and time consumption of the model checking using FTTS in comparison with FGTS. The selected case studies are the models of a *Wireless Sensor and Actuator Networks (WSAN)*, the simplified version of *Scheduler of Hadoop*, a *Ticket Service* system, and simplified version of *802.11 Wireless Protocol.*

5.4.1 Hadoop YARN Scheduler

Hadoop [50] is a framework for MapReduce, a programming model for generating and processing large data sets, selected as the second example. The overview and detailed description of this example are presented in Section 2.3.5. Here, we used two versions of Hadoop model which differ in the behavior of task queues. The first model has a normal task queue and the second one has a priority task queue.

As shown in Tables 5.2 and 5.1, the gained reduction in this example is less than 40 percent. This limited gain is because of the fact that there is few number of delay statements in the Hadoop model, so, there is a poor chance of having split transitions in its FGTS.

[2]The latest version of the toolset is accessible from `http://rebeca-lang.org/alltools/Afra`

Config	FTTS			FGTS			Reduction	
	States	Trans.	Time	States	Trans.	Time	States	Trans.
1 AM	375	707	1< sec	581	913	< 1 sec	35%	24%
2 AMs	1.40K	2.90K	1 sec	2.10K	3.83K	1 sec	33%	24%
3 AMs	5.10K	13.3K	2 secs	7.02K	16.7K	2 secs	28%	20%
4 AMs	28.19K	92.87K	70 secs	37.82K	116.7K	78 secs	25%	20%

Table 5.1: Comparing the number of states and transitions in FGTS and FTTS of YARN example with priority queue

Config	FTTS			FGTS			Reduction	
	States	Trans.	Time	States	Trans.	Time	States	Trans.
1 AM	44	62	1< sec	69	87	1< sec	36%	29%
2 AMs	351	614	1< sec	557	848	1< sec	37%	28%
3 AMs	1.91K	4.23K	1 sec	2.73K	5.29K	1 sec	30%	20%
4 AMs	15.41K	42.06K	18 secs	20.61K	50.73K	18 secs	25%	17%
5 AMs	113.6K	378.1K	25 mins	27.4K	453.3K	27 mins	23%	17%

Table 5.2: Comparing the number of states and transitions in FGTS and FTTS of YARN example with normal queue

Config	FTTS			FGTS			Reduction	
	States	Trans.	Time	States	Trans.	Time	States	Trans.
33-6-4-2	977	1.5K	1< sec	1.92K	2.52K	1< sec	49%	41%
25-5-4-10	1.85K	2.54K	1< sec	3.72K	4.55K	1< sec	50%	44%
30-6-4-2	4.75K	5.78K	1< sec	9.35K	10.46K	2 secs	50%	45%
25-6-4-2	17.02K	20K	5 secs	34.5K	37.85K	24 secs	51%	47%
20-6-4-2	28.19K	32.19K	16 secs	57.62K	62.21K	64 secs	51%	48%

Table 5.3: Comparing the number of states and transitions in FGTS and FTTS of WSAN example

5.4.2 WSAN Applications

As the second example, we present a realtime data acquisition system for structural health monitoring and control (SHMC) of civil infrastructures. The overview and detailed description of this example are presented in Section 2.3.6.

In the case of the *WSAN* model, in each row, the size (the numbers which are separated by dashes) is a combination of the sampling rate, the number of nodes, the packet size, and the sensor task delay of the model, respectively. As the complexity of these examples is greater than the *Yarn* model, the reduction is about 50%. It is because of the fact that the majority of message servers in the WSAN model shows timed behavior, results in increasing the chance of having split transitions in its FGTS.

Config	FTTS			FGTS			Reduction	
	States	Trans.	Time	States	Trans.	Time	States	Trans.
1 Customer	5	6	1< sec	8	9	1< secs	38%	33%
2 Customers	51	77	1< sec	77	107	1< sec	34%	28%
3 Customers	252	418	1< sec	360	550	1< sec	30%	24%
4 Customers	1.29K	2.21K	1< sec	1.82K	2.89K	1< sec	30%	24%
5 Customers	7.53K	12.8K	1< sec	10.7K	16.9K	1< sec	30%	24%
6 Customers	51.6K	84.7K	2 secs	73.5K	114K	2 secs	30%	26%
7 Customers	408K	650K	18 secs	582K	884K	24 secs	30%	26%

Table 5.4: Comparing the number of states and transitions in FGTS and FTTS of Ticket Service example

Config	FTTS			FGTS			Reduction	
	States	Trans.	Time	States	Trans.	Time	States	Trans.
2 Interfaces	1.12K	2.09K	2 secs	1.92K	2.62K	2 secs	10%	4%
3 Interfaces	59K	196K	122 secs	61K	198K	153 secs	3%	1%

Table 5.5: Comparing the number of states and transitions in FGTS and FTTS of CA protocol

5.4.3 Ticket Service

Our third example is the model of a *Ticket Service* system. The overview and detailed description of this example are presented in Section 2.3.1.

The trend of changes in the case of Ticket Service example is the same as that of in the WSAN model. Increasing the number of customers results in increasing requests for issuing a ticket. Issuing a ticket is served by a message server which contains a delay statement. So, the execution of this message server is split into two parts. As a result, having more customers results in having to split message server executions and increasing the efficiency of using FTTS in comparison to FGTS.

5.4.4 The IEEE 802.11 RTS/CTS Collision Avoidance Protocol

The fourth example is the model of a *Collision Avoidance* protocol. The overview and detailed description of this example are presented in Section 2.3.3. There are some exceptional models in which the state space size and the number of transitions in FGTS and FTTS are close to each other. The model of the *Collision Avoidance* protocol is one of them. As there is no delay statement in the body of the message servers of *Collision Avoidance* protocol, the execution of the message servers takes place without the progress of time. Therefore, atomic execution of message servers in FTTS and the rather fine-grain execution of message servers in FGTS results in state spaces with comparable sizes. The effectiveness of FTTS is reduced in this kind of models. Table 5.5 also shows that using FTTS reduces the model checking time consumption (even in the case of the *Collision Avoidance* protocol). It is because of the simplicity of the generated state space in FTTS, using the atomic execution of message servers.

Chapter 6

Case Studies

In this section, we introduce three different case studies, analyzed using the technique and the toolset of this thesis, which are "Analyzing Wireless Sensor and Actuator Networks", "Analyzing Different Scheduling Policies in YARN", and "Functional and Performance Analysis of NoCs". The first case study is developed by the author of this thesis as a part of the contributions of the thesis. The second and third case studies are developed in two independent master theses by Helgi Leifsson in [62] and Zeynab Sharifi in [63], respectively; so, they cannot be counted as the contributions of this thesis.

6.1 Analyzing Wireless Sensor and Actuator Networks[1]

Wireless sensor and actuator networks (WSANs) can provide low-cost continuous monitoring. Building WSAN applications is particularly challenging because of the complexity of concurrent and distributed programming, networking, realtime requirements, and power constraints. As a result, it can be hard to find a configuration that satisfies these constraints while optimizing the resource usage. A common approach to address this problem is to perform an informal analysis based on conservative worst-case assumptions and empirical measurements. This can lead to poor utilization of resources. For example, a workload consisting of two periodic tasks would be guaranteed to be safe only if the sum of the two worst-case execution times (WCET) were less than the period of both of the two tasks. Whereas, it is possible in practice to have many safe schedules violating this restriction.

Another approach for this problem is a trial and error. For example, in [53], an empirical test-and-measure approach based on binary search is used to find configuration parameters: worst-case task runtimes, timeslot length of the communication protocols, etc. Trial and error is a laborious process, which nevertheless fails to provide any safety guarantees for the resulting configuration.

A third possibility is to extend scheduling techniques that have been developed for realtime systems [65] so that they can be used in WSAN environments. Unfortunately, this turns out to be difficult in practice. Many WSAN platforms rely on highly efficient event-driven operating systems such as TinyOS [66]. Unlike a realtime operating system (RTOS), event-driven operating systems generally do not provide realtime

[1]This chapter is an improvement and extension of the results published in [55] and [64].

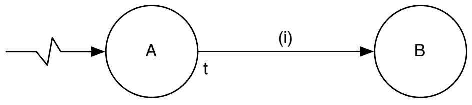


Figure 6.1: An example of an event graph

scheduling guarantees, priority-based scheduling, or resource reservation functionality. Without such support, many schedulability analysis techniques cannot be effectively employed. For example, in the absence of task preemption and priority-based scheduling, unnecessarily conservative assumptions must be used to guarantee correctness in the general case.

We propose an actor-based modeling approach that allows WSAN application programmers to assess the performance and functional behavior of their developed codes throughout the design and implementation phases. The developed models are analyzed using model checking to determine the parameter values resulting in the highest resource utilization. A WSAN application is a distributed system with multiple sensor nodes, each comprised of the independent concurrent entities: CPU, sensor, radio system, and bridged together via a wireless communication medium which uses a transmission control protocol. Interactions between entities, both within a node and across nodes, are concurrent and asynchronous. Moreover, WSAN applications are sensitive to timing, with soft deadlines at each step of the process that is required to ensure correct and efficient operation.

Due to the performance requirements and latencies of operations on sensor nodes, coordination among sensing, data processing, and communication activities is required. In particular, once a sample is acquired from a sensor, its corresponding radio transmission activities must be performed. At the same time, data processing tasks must be executed (for example, because of the environmental changes in the temperature, a kind of data compensation must be applied on sensor data to adjust the acquired values). Moreover, the timing of radio transmissions from different nodes must be coordinated using a communication protocol.

6.1.1 Preliminaries: Event Graphs

At the first step of modeling WSAN applications, we need to introduce *event graphs* in which a highly abstracted view of scheduling events can be depicted. Event graphs have a single type of node and two types of edges, i.e. jagged and ordinary edges. The nodes represent events in a system. Edges correspond to the scheduling of other events [67]. In this graph, the initial event is shown by jagged edges. Edges can optionally be associated with an enabling guard, i.e. a boolean condition, for scheduling an event and/or a time delay which means that an event will be scheduled after the delay. Figure 6.1 shows an example of an event graph where the event B is scheduled by the event A if its associated guard *(i)* is evaluated to true, at t units of time later than the current time.

Event graphs are widely used in the engineering community for the simulation and analysis of complex systems. More specifically, they are used to graphically represent discrete-event simulation models.

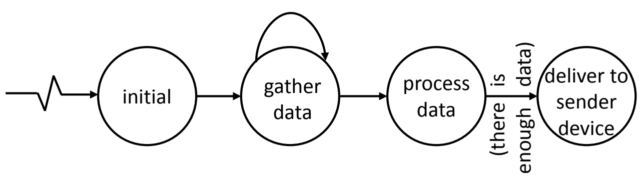


Figure 6.2: The event graph of a WSAN sensor behavior

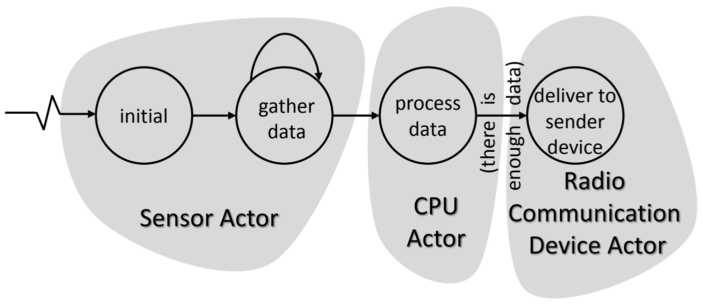


Figure 6.3: How events of a WSAN sensor are associated with actors

6.1.2 The Actor Model of WSAN Applications

The characteristics of Timed Rebeca make it useful for modeling WSAN applications: many concurrent processes and interdependent realtime deadlines. Observe that common tasks such as sample acquisition, sample processing, and radio transmission are periodic and have well-known or easily measurable periods. This makes the analysis of worst-case execution times feasible. However, because of the event-triggered nature of applications, initial offsets between the tasks are variable.

At the first step of proposing an actor model for the WSAN applications, we need to have a look into the interaction of the components and the events which are triggered and served by them. Based on the specification of WSAN applications, there are many nodes which have the role of data acquisition and data transmission. For data acquisition, a node has a set of sensors which periodically acquire data from the environment and send the data to the processing unit of the node. The processing unit is responsible for validating the data and storing it in an internal buffer. Upon receiving enough data, the processor unit sends the data to the radio communication unit. The radio communication unit tries to send data via a wireless medium, considering a predefined communication protocol. The event graph of this model is depicted in Figure 6.2. The majority of WSAN applications can be modeled using this graph; although, minor modifications may be needed.

We split up the event handlers of the events of Figure 6.2 into three different actors, depicted in Figure 6.3 and add one additional actor for carrying out miscellaneous tasks unrelated to sensing or communication. This additional actor is necessary for making the model close to its real configuration. The three actors are called `Sensor` (for the data acquisition), `CPU` (processing unit), and `RCD` (a radio communication device) together with the additional actor is called `Misc`. `Sensor` collects data and send it to `CPU` for further data processing. Meanwhile, `CPU` may respond to messages from `Misc` by carrying out other computations. The processed data is sent to `RCD` to forward it to a data collector node actor.

Composing the collection of sensor nodes to develop a complete WSAN application requires that the wireless communication medium is specified and a communication

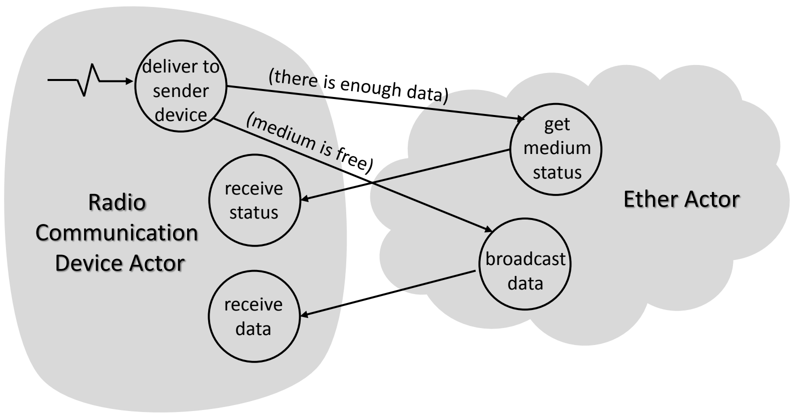


Figure 6.4: How events of wireless communication mechanism are associated with actors

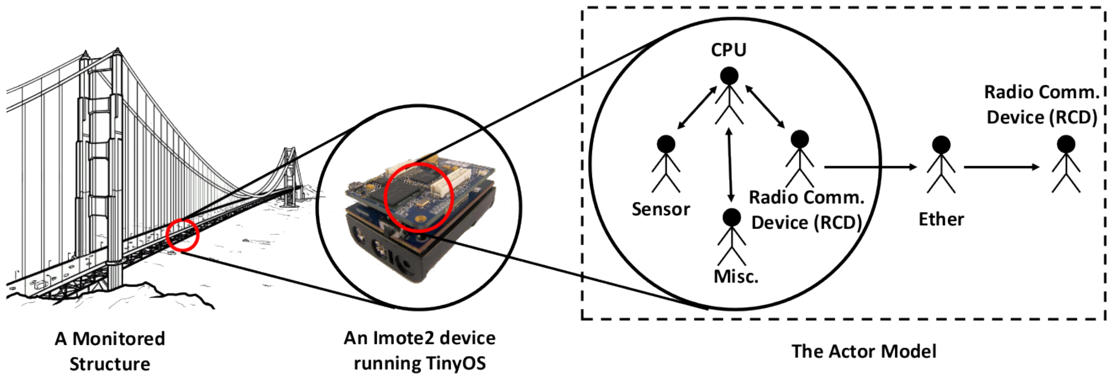


Figure 6.5: Modeling the behavior of a WSAN application in its real-world installation in the actor model

protocol is implemented in radio communication devices. Note that the process of sending a packet is controlled by a wireless network communication protocol. We model the communication medium as an actor (`Ether`) and the receiver node also by the actor `RCD`. Using the actor `Ether` facilitates modularity: specifically, implementation of the Media Access Control (MAC) level details of communication protocols is localized. As a result, different implementations of communication protocols can be replaced without significantly impacting the remainder of the model. As shown in Figure 6.4, `Ether` serves events for receiving the status of the medium and broadcasting data. For the development of different communication protocols, different combinations of these two events can be triggered to model the behavior of the protocols.

During the application design phase, different components, services, and protocols may be considered. For example, TDMA [68] as a MAC-level communication protocol may be replaced by B-MAC [69] with minimal changes. In a nutshell, using the mentioned association of events with actors, a given WSAN application is modeled by actors as shown in Figure 6.5.

6.1.3 Schedulability Analysis of a Stand-Alone Node

We now illustrate our approach using a node-level Timed Rebeca model of a WSAN application to check for possible deadline violations. Specifically, by changing the timing parameters of our model, we find the maximum safe sampling rate in the presence of other (miscellaneous) tasks in the node. Then, we show how the specification of a node-level model can be naturally extended to a network-wide specification. Note that the values which are used in this model (e.g. radio transmission time, sensor task delay, etc) are come from the real implementation of a WSAN application in the domain of structural health monitoring.

Following the mapping of Figure 6.5, the Timed Rebeca model for the four different *reactive classes* is presented in Listing 6.1 through Listing 6.3. As shown in Listing 6.1, the maximum capacity of the message bag of `Sensor` is set to 10, the only actor which `Sensor` knows about is of type `CPU` (line 4), and `Sensor` does not have any state variables. The behavior of `Sensor` is to acquire data and send it to `CPU` periodically. This behavior is implemented using `sensorLoop` (lines 10-14) which sends the acquired data to `CPU` (line 12). The sent data must be served before the start time of the next period, specified by the value of `period` as the parameter of `deadline`.

Recall that there is a nondeterministic initial offset after which the data acquisition becomes a periodic task. To represent this property, `Sensor` which sends a `sendLoop` message to itself; the message is nondeterministically delivered after one of 10, 20, and 30 (line 8). After this random offset, the sensor's periodic behavior is initiated. Note that in line 1, the sampling rate is defined as a constant. A similar approach is used in the implementation of the `Misc` reactive class.

Listing 6.1: The Timed Rebeca implementation of `Sensor` reactive class

```
 1 env int samplingRate = 25; // Hz
 2
 3 reactiveclass Sensor(10) {
 4   knownrebecs { CPU cpu; }
 5
 6   Sensor() { self.sensorFirst(); }
 7   msgsrv sensorFirst() {
 8     self.sensorLoop() after(?(10, 20, 30)); // ms
 9   }
10   msgsrv sensorLoop() {
11     int period = 1000 / samplingRate;
12     cpu.sensorEvent() deadline(period);
13     self.sensorLoop() after(period);
14   }
15 }
```

The behavior of `CPU` as the target of `Sensor` and `Misc` events is more complicated (Listing 6.2). Upon receiving a `miscEvent`, `CPU` waits for `miscTaskDelay` units of time; this represents computation cycles consumed by miscellaneous tasks. Similarly, after receiving the `sensorEvent` message from `Sensor`, `CPU` waits for `sensorTaskDelay` units of time; this represents cycles required for the intra-node data processing. Data must be packed in a packet of a pre-specified `bufferSize` (line 16) and when the threshold is reached (line 17), `CPU` asks `senderDevice`, to send the collected data in one packet (line 18).

Listing 6.2: The Timed Rebeca implementation of `CPU` reactive class

```
1 env int sensorTaskDelay = 2; // ms
2 env int miscTaskDelay = 10; // ms
3 env int bufferSize = 3; // samples
4
5 reactiveclass CPU(10) {
6   knownrebecs {RCD senderDevice, receiverDevice;}
7   statevars { int collectedSamplesCounter; }
8
9   CPU() { collectedSamplesCounter = 0; }
```

```

  msgsrv miscEvent() {
    delay(miscTaskDelay);
  }
  msgsrv sensorEvent() {
    delay(sensorTaskDelay);
    collectedSamplesCounter += 1;
    if (collectedSamplesCounter == bufferSize) {
      senderDevice.send(receiverDevice, 1);
      collectedSamplesCounter = 0;
    }
  }
}
```

As this is a node-level model, communication between nodes is omitted and the behavior of RCD is limited to waiting for some amount of time (line 6 of Listing 6.3); this represents the sending time of a packet.

Listing 6.3: The node-level implementation of RCD

```
1 env int OnePacketTT = 7; // ms(transmission time)
2
3 reactiveclass RCD (2) {
4   RCD() { }
5   msgsrv send(RCD receiver, byte numOfPackets) {
6     delay(OnePacketTT * numOfPackets);
7   }
8 }
```

Note that computation times (`delay`'s) depend on the low-level aspects of the system and are application-independent; they can be measured prior to starting the design of applications. For schedulability analysis, we set the `deadline` for messages in a way that any scheduling violations are caught by the model checker.

6.1.4 Schedulability Analysis of Multi-Node Model with a Distributed Communication Protocol

Composing the models of stand-alone nodes to have a multi-node model requires that the wireless communication medium Ether be specified and a communication protocol is implemented for radio communication devices. Recall that nodes in the multi-node model periodically send their data to an aggregator node (Listing 6.5). The sending process is controlled by a wireless network communication protocol. The reactive class `Ether` (Listing 6.4) has three message servers: which are responsible for sending the status of the medium, broadcasting data, and resetting the condition of the medium after a successful transmission.

Broadcasting data takes place by sending data to an `RCD` which results in setting the values of `senderDevice` and `receiverDevice` to their corresponding actors. So, the status of `Ether` can be easily examined by the value of `receiverDevice` (i.e., using `null` as the value of `receiverDevice` is interpreted as the medium is free, line 13). This way, upon sending data successfully, the value of `receiverDevice` and `senderDevice` must be set to `null` to show that the transmission is completed (lines 30 and 31).

Data broadcasting is the main behavior of `Ether` (lines 16 to 28). Before the start of broadcasting, `Ether` status is checked (line 17) and data-collision error is raised in the case of two simultaneous broadcasts (line 26). With a successful data broadcast, `Ether` sends an acknowledgment to itself (line 20) and the sender (line 22), and informs the receiver of the number of packets sent to it (line 24). In addition to the functional requirements of `Ether`, there may be non-functional requirements. For example, the Imote2 radio offers a theoretical maximum transfer speed of 250 kbps. When considering only the useful data payload (goodput), this is reduced to about 125 kbps.

Listing 6.4: The Timed Rebeca implementation of `Ether` reactive class

```
 1 env int OnePacketTT = 7; // ms(transmission time)
 2
 3 reactiveclass Ether(5) {
 4   statevars {
 5     RCD senderDevice, receiverDevice;
 6   }
 7
 8   Ether() {
 9     senderDevice = null;
10     receiverDevice = null;
11   }
12   msgsrv getStatus() {
13     ((RCD)sender).receiveStatus(
14       receiverDevice != null);
15   }
16   msgsrv broadcast(RCD receiver, int packets) {
17     if(senderDevice == null) {
18       senderDevice = (RCD)sender;
19       receiverDevice = receiver;
20       self.broadcastingIsCompleted()
21         after(packets * OnePacketTT);
22       ((RCD)sender).receiveResult(true)
23         after(packets * OnePacketTT);
24       receiver.receiveData(receiver, packets);
25     } else {
26       ((RCD)sender).receiveResult(false);
27     }
28   }
29   msgsrv broadcastingIsCompleted() {
30     senderDevice = null;
31     receiverDevice = null;
32   }
33 }
```

We now extend `RCD` to support communication protocols. Listing 6.5 shows the Timed Rebeca implementation model of TDMA protocol. TDMA protocol defines a cycle, over which each node in the network has one or more chances to transmit a packet or a series of packets. If a node has data available to transmit during its allotted time slot, it may be sent immediately. Otherwise, packet sending is delayed until its next transmission slot. This way, the packet transmission of one sensor node does not interfere with the other sensor nodes. Having more sensor nodes only results in having shorter time slots, so the presence of sensor nodes can be abstracted and modeled as making time slots shorter. Using this abstraction, compositional verification of WSAN applications against schedulability and deadlock-freedom properties become feasible as only one node which is in communication with the central node has to be considered for networks in any size.

The periodic behavior of TDMA slot is handled by `handleTDMASlot` message server which sets and unsets `inActivePeriod` to show that whether the node is in its allotted time slot. Upon entering into its slot, a device checks for pending data to send (line 32) and schedules `handleTDMASlot` message to leave the slot (line 31). On the other hand, when `CPU` sends a packet (message) to an `RCD`, the message is added to the other pending packets which are waiting for the next allotted time slot. `tdmaSlotSize` is the predefined size of the TDMA slots, and `currentMessageWaitingTime` is the waiting time of this message in the bag of its receiver.

Listing 6.5: The Timed Rebeca implementation of TDMA protocol in `RCD`

```
env int OnePacketTT = 7; ms (transmission time)

reactiveclass RCD (10) {
  knownrebecs { WirelessMedium medium; }
  statevars {
    byte id;
    int slotSize, sendingData;
    boolean busyWithSending, inActivePeriod;
    RCD receiverDevice;
  }

  RCD(byte myId) {
    id = myId;
    inActivePeriod = false;
    sendingData = 0;
    busyWithSending = false;
    receiverDevice = null;
    ...
  }
  msgsrv send(RCD receiver, int data, int packetsNumber) {
    assertion(receiverDevice == null);
    receiverDevice = receiver;
    sendingData = data;
    self.checkPendingData();
  }
  msgsrv handleTDMASlot() {
    inActivePeriod = !inActivePeriod;
    if(inActivePeriod) {
      int remainedTime = tmdaSlotSize - currentMessageWaitingTime;
      assertion(remainedTime > 0);
      self.handleTDMASlot() after(remainedTime);
      self.checkPendingData();
    } else {
      self.handleTDMASlot() after((tmdaSlotSize * (numberOfNodes - 1))- currentMessageWaitingTime);
    }
  }
  msgsrv checkPendingData() { ... }
  msgsrv receiveStatus(boolean result) { ... }
  msgsrv receiveResult(boolean result) { ... }
  msgsrv receiveData(RCD receiver, int data, int receivingPacketsNumber) {
    if (receiver == self) {
        delay(receivingPacketsNumber * OnePacketTransmissionTime);
    }
  }
}
```

For the sake of simplicity, some details of `RCD` are omitted in Listing 6.5. The complete source code (which implements the B-MAC protocol) is available on the Rebeca web page.

B-MAC protocol is designed for low power Ad-Hoc networks in which some sender nodes send data to a receiver. Like the other low power protocols, B-MAC uses periodical sleep/wake-up cycles. During wake-up times, the node listens for incoming data transmissions. If there is no data to receive, the listen state is interrupted and the node moves to sleep state by turning off the radio device. Otherwise, the node stays in the listen state for complete data transmission. The sleep periods of nodes may differ, making B-MAC an asynchronous communication protocol. When a node wants to send, it turns on the radio and starts sending an announcement. This announcement is long enough to make sure that it has overlap with the wake-up time of the data receiver. Afterwards, the sender transmits data to the target address. In order to reduce the amount of needed energy, clear channel assessment is used with the aim of better separation between signals and noise on the channel. B-MAC has an application interface for flexible configuring parameters. A good value for this sometimes depends on the use case, so, this can be adjusted by a higher layer application.

We now extend `RCD` to implement B-MAC protocols, depicted in Listing 6.6. In contrast to TDMA, B-MAC RCD tries to detect free channel status and sends data upon receiving a request from CPU (line 20). In the case of detecting free channel, the data is sent immediately (line 24). This way, collisions may occur; so, RCD has to wait for some amount of time and resend data (line 23). B-MAC protocol does not need complicated and expensive synchronization methods. It also avoids data fragmentation. So, it would be more complicated to coordinate long messages and B-MAC expects short messages, which is common for the size of packets of WSAN nodes.

Based on this fact, the presence of sensor nodes can be abstracted and modeled as the possible number of collisions before a data communication is performed successfully. Using this abstraction, efficient verification of WSAN applications becomes feasible as only one sensor node which is in communication with the central node has to be considered for networks in any size. Any data transmission of this sensor node maybe encounter a collision. The maximum number of the collisions is the number of sensor nodes in the model. So, in the Rebeca code for `RCD`, for each data transmission, we have a non-deterministic choice between a successful transmission or a collision. During model checking, in the case of collision, data transmission with zero, one, ..., up to n collisions are considered where n is the number of sensor nodes. The Timed Rebeca model of this protocol is available on the Rebeca home page[2].

Listing 6.6: The Timed Rebeca implementation of B-MAC protocol in `RCD`

```
 1 env int OnePacketTT = 7; ms (transmission time)
 2
 3 reactiveclass RCD (10) {
 4   knownrebecs { WirelessMedium medium; }
 5   statevars {
 6     byte id;
 7     int sendingData;
 8     RCD receiverDevice;
 9   }
10
11   RCD(byte myId) {
12     id = myId;
13     sendingData = 0;
14     receiverDevice = null;
15   }
16   msgsrv send(RCD receiver, int data, int packetsNumber) {
17     assertion(receiverDevice == null);
18     receiverDevice = receiver;
19     sendingData = data;
20     medium.getStatus();
21   }
22   msgsrv receiveStatus(boolean result) {
23     delay((numberOfNodes/2) * (OnePacketTT + 1));
24     medium.broadcast(receiverDevice,sendingData, packetsNumber);
25     delay(OnePacketTT * packetsNumber);
26   }
27   msgsrv receiveResult(boolean result) { ... }
28   msgsrv receiveData(RCD receiver, int data, int receivingPacketsNumber) { ... }
29 }
```

[2]The latest version of this model is accessible as one of examples which are developed in TARO project from `http://rebeca-lang.org/allprojects/TARO`

Once a complete model of the distributed application has been created, Afra verifies whether the schedulability properties hold in all reachable states of the system. If there are any deadline violations, a counterexample will be produced, indicating the path—sequence of states from an initial configuration—that results in the violation. This information can be helpful with changing the system parameters, such as increasing the TDMA time slot length or reducing the sampling rate, to prevent such situations.

6.1.5 Generalization of the Approach for Any WSAN Application

Here, we summarize the modeling approach and describe the way of extending it to make the approach applicable for the other WSAN applications. It is noteworthy that the actor-based approach is aligned with the structure of WSAN applications with different types of behaviors. Loosely coupled actors as the units of concurrency, with asynchronous message passing, and event-driven computation, are natural candidates for modeling such systems. The semantic gap between the model and the real-world system that has to be modeled is small, this so-called fidelity of the model to the system makes modeling easier and also makes the model easy to understand. Also, the possibility of building an understandable model with the least needed effort illustrates the usability of the model.

There is a natural rule for mapping a WSAN application to the actor model. Each entity in a WSAN application that is running concurrently, and is serving or creating events, has to be mapped into an actor. Therefore, two events which are served concurrently will be served by two different actors. In contrast, if there are two triggered events which are sent to the same entity and are served sequentially, then this entity is mapped to an actor which serves both events.

The simplest scenario is when there is no concurrent activity within each node in a WSAN application. In this scenario, the network communication among nodes are directly mapped to asynchronous communication among actors, and intra-node activities of each node are mapped into the event handlers of the corresponding actor. In a more general case, we recognize five different actors in a WSAN application, Sensor, CPU, Misc, RCD, and Ether. Here, the events created by sensors and miscellaneous activities have to be served by CPU and then sent to other nodes in the network.

One may want to make the model even more general as there may be more than one sensor in each node (e.g. one for humidity and one for temperature measurements). In this case, two different actors are needed to model the concurrent (data acquisition and) event creation of sensors. Also, in the case of using multi-core CPUs inside a node, more than one CPU actor has to be associated with a node and the incoming tasks from sensors and miscellaneous activities have to be dispatched among them. Note that handling a multi-core CPU in a node requires developing a task scheduler which must be run on one of the cores and dispatches the incoming tasks to appropriate CPUs. A modeler may perform these activities using event graphs or associating events with actors directly.

In addition to the mapping of entities to actors, we also need to model different communication protocols among nodes. The current implementation of Ether, explained in Section 6.1.4, serves two events which facilitates modeling of any wireless node-to-node communication protocol. Basically, no modification is needed in this code for supporting other node-to-node communications protocols. However, one may want to develop a WSAN application in which a node broadcasts data to some other nodes. In this case, Ether must be extended to support multi-node data broadcasting, multicasting, or anycasting.

6.1.6 Experimental Results

To illustrate the applicability of this approach, we present a case study involving real-time continuous data acquisition for structural health monitoring and control (SHMC) of civil infrastructure [53]. SHMC is fed with algorithms that control centralized or distributed control elements such as active and semi-active dampers. The control algorithms attempt to minimize vibration and maintain stability in response to excitations from rare events such as earthquakes, or more mundane sources such as wind and traffic. The system we examine has been implemented on the Imote2 wireless sensor platform [53], which features a powerful embedded processor, sufficient memory size, and a high-fidelity sensor suite required to collect data of sufficient quality for SHMC purposes. These nodes run the TinyOS operating system, supported by middleware services of the Illinois SHM Services Toolsuite [70]. They used in several long-term developments of several highway and railroad bridges [71].

SHMC application development has proven to be particularly challenging: it has the complexity of a large-scale distributed system with realtime requirements while having the resource limitations of low-power embedded WSAN platforms. Ensuring safe execution requires modeling the interactions between CPU, sensor, and radio within each node, as well as interactions among the nodes. Moreover, the application tasks are not isolated from other aspects of the system: they execute alongside tasks belonging to other applications, middleware services, and operating system components. In the application we consider, all periodic tasks (sample acquisition, data processing, and radio packet transmission) are required to complete before the next iteration starts. Our results show that a guaranteed safe application configuration can be found using the Afra model checking tool. Moreover, this configuration improves resource utilization compared to the previous informal schedulability analysis used in [53], supporting a higher sampling rate or a larger number of nodes without violating schedulability constraints.

Based on the specification, WSAN nodes in flexible data acquisition system can be configured to support realtime collection of high-frequency, multi-channel sensor data from up to 30 wireless smart sensors at frequencies up to 250 Hz. As it is designed for high-throughput sensing tasks that necessitate larger networks sizes with relatively high sampling rates, it falls into the class of *data-intensive sensor network applications*, where efficient resource utilization is critical since it directly determines the achievable scalability (number of nodes) and fidelity (sampling frequency) of the data acquisition process. Configured on the basis of network size, associated sampling rate, and desired data delivery reliability, it allows for near-realtime acquisition of 108 data channels on up to 30 nodes—where each node may provide multiple sensor channels, such as 3-axis acceleration, temperature, or strain—with minimal data loss. In practice, these limits are determined primarily by the available bandwidth of the IEEE 802.15.4 wireless network and sample acquisition latency of the sensors. The accuracy of estimating safe limits for sampling and data transmission delays directly impacts the system's efficiency.

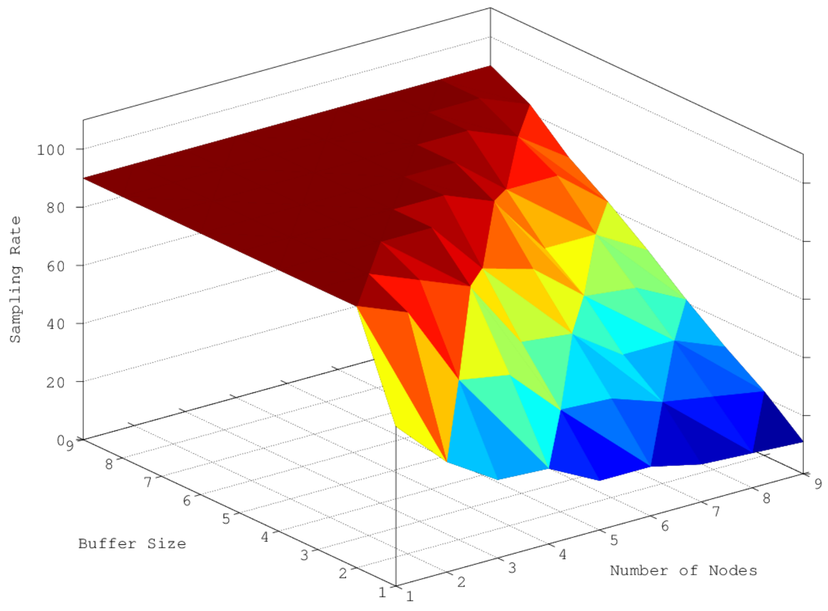


Figure 6.6: Maximum sampling rate in case of using TDMA protocol and setting the value of `sensorTaskDelay` to 2ms

6.1.6.1 Finding the Maximum Sampling Rate

In this part, we considered applications where achieving the highest possible sampling rate that does not result in any missed deadline is desired. This is a very common requirement in WSAN applications in the SHMC domain in particular. We begin by setting the value of `OnePacketTT` to 7ms (i.e., the maximum transmission time of this type of applications) and fixed the value of `sensorTaskDelay`, `miscPeriod`, and `miscTaskDelay` to some predefined values. In addition to the sampling rate, the number of nodes in the network and the packet size remain variable. By assuming different values for the number of nodes and the packet size, different maximum sampling rates are achieved, shown as a 3D surface in Figure 6.6. As shown in the figure, higher sampling rates are possible when the buffer size is set to a larger number (there is more space for data in each packet). Similarly, increasing the number of nodes decreases the sampling rate: in competition among three different parameters of Figure 6.6, the cases with the maximum buffer size (i.e., 9 data points) and minimum number of nodes (i.e., 1 node) results in the highest possible maximum sampling rates. Decreasing the buffer size or increasing the number of nodes, non-linearly reduces the maximum possible sampling rate.

A server with Intel Xeon E5645 @ 2.40GHz CPUs and 50GB of RAM, running Red Hat 4.4.6-4 as the operating system was used as the host of Afra. We changed the size of the state space from less than 500 to more than 140K states, resulting in model checking times ranging from 0 to 6 seconds. Analyzing the specifications of the state spaces, some relations between the size of the state spaces and the configurations of the models are observed. For example, the largest state spaces correspond to configurations where `sensorTaskDelay`, `bufferSize`, and `numberOfNodes` are set to large values.

6.1.6.2 Real-World Applications

Although we showed that how WSAN applications can be modeled by Timed Rebeca and be analyzed by Afra, its usability for the real-world applications has to be discussed. To illustrate how practical is the approach of this paper, we performed two examinations.

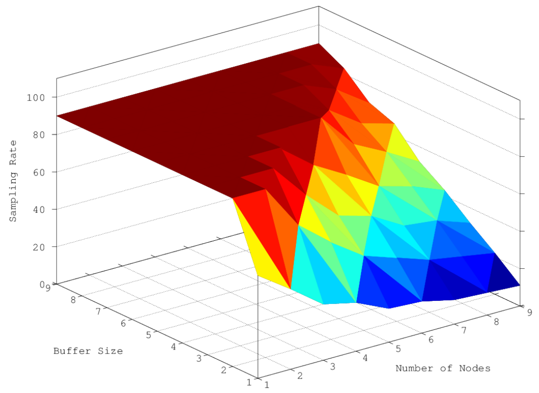
In the first one, we tried to use the approach for the analysis of a real-world installation of an SHMC application. To this end, the parameters of the model are set to values which are determined by a real-world installation of an SHMC application and the communication protocol of the model is set to TDMA. Our results show that the current manually-optimized installation can be tuned to an even more optimized one: by changing the configuration based on the findings of using analysis approach of this work, the performance of the system safely improved by 7% percent.

In the second examination, we tried to figure out the effect of using two different widely used communication protocols, B-MAC and TDMA, on the performance of the system using the approach of this work. B-MAC (Berkeley Media Access Control) is a MAC-level protocol for WSAN applications which uses adaptive preamble sampling scheme. This technique consists of sampling the medium at fixed time periods. Using B-MAC, every node samples the medium at fixed intervals to figure out nodes which are willing to communicate. If there is a node which has a data packet to send, sender senses the medium if it is free, takes a small back-off and then sends the data packet. This way, data packets are sent as soon as the communication medium is free. But, the possibility of communication loss because of transmission collisions is increased. In contrast, TDMA protocol allocates to each node an exclusive time slot for communication and guarantees collision-free media access in that slot. This behavior allows reducing preamble transmissions to save more energy. But, as they afford longer slots with a larger sleeping part, a ready data packet may wait for a longer time to be sent. We also changed the value of `sensorTaskDelay` in the supported maximum sampling rate, considering 648 different configurations. The maximum sampling rate found for each configuration is depicted in Figure 6.7; the figure shows that increasing the value of `sensorTaskDelay` as the representer of intra-node activities, decreases the sampling rate dramatically. Conforming the theoretical expectation, they also show that using B-MAC results in achieving higher sampling rates in comparison with TDMA, as the waiting times of ready to send data packets in B-MAC are smaller than that of TDMA.

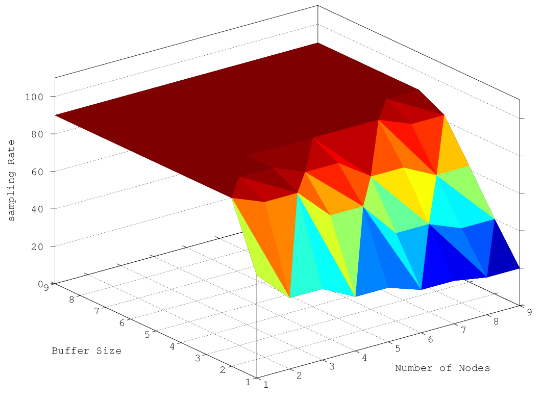
6.2 Analyzing Different Scheduling Policies in YARN

MapReduce is a programming paradigm for generating and processing large data sets [72]. In this paradigm, users have to specify a map function that processes a pair of key/value into a set of intermediate pairs of key/value. Users also specify a reduce function that merges all intermediate values associated with the same intermediate key.

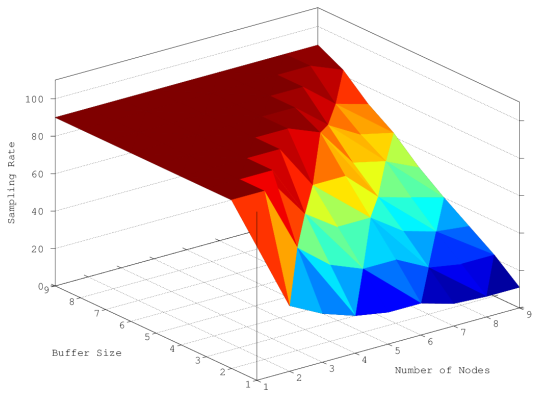
Hadoop is a framework for MapReduce [52], and YARN (Yet Another Resource Negotiator) is a part of Hadoop. YARN leaves the responsibilities of job scheduling and task progress monitoring to a Resource Manager (RM). These activities include doing task bookkeeping, keeping track of tasks, maintaining counter totals, and restarting failed or slow tasks. An Application Master (AM) negotiates with the RM for access to resources. This way, it manages the life-cycle of applications like MapReduce jobs running on a cluster. Considering a cluster, there is one instance of RM and for every



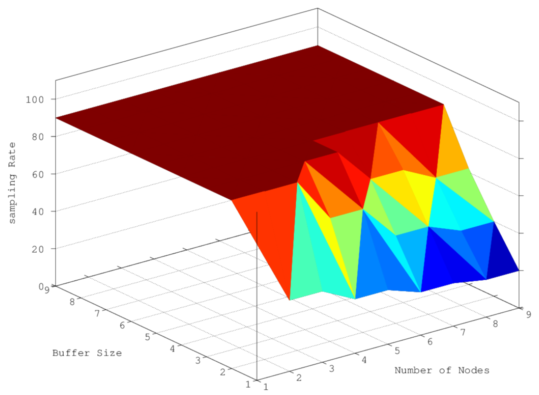
(a) TDMA, Sensor task delay is 5ms



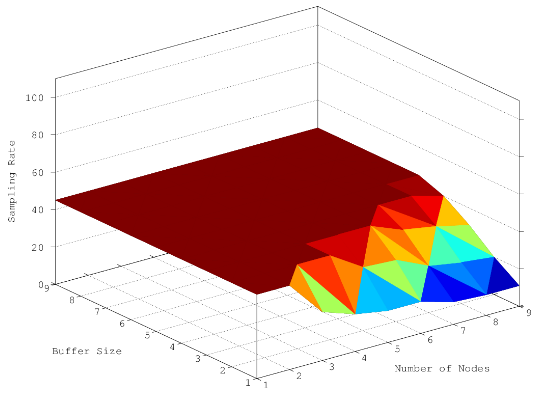
(b) B-MAC, Sensor task delay is 5ms



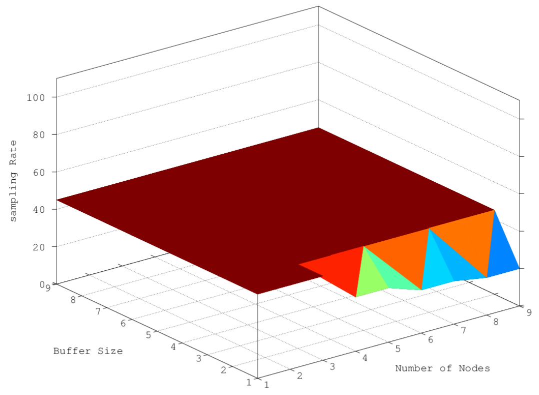
(c) TDMA, Sensor task delay is 10ms



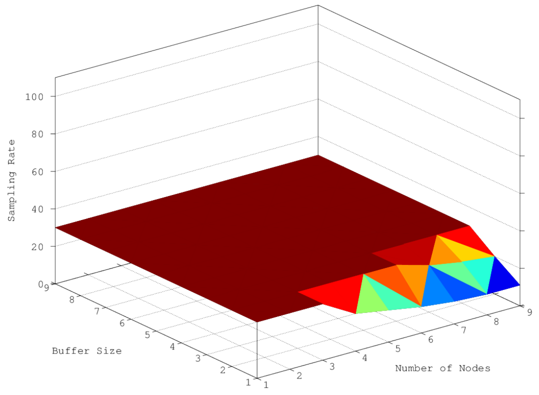
(d) B-MAC, Sensor task delay is 10ms



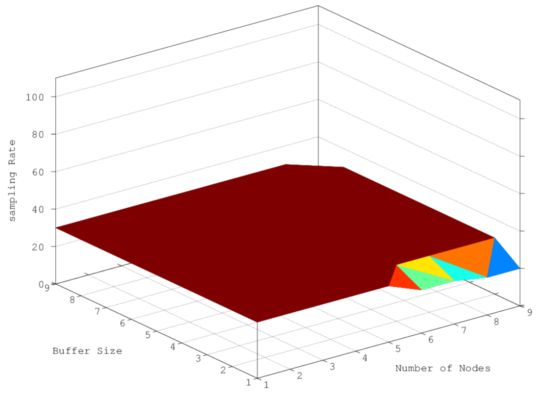
(e) TDMA, Sensor task delay is 20ms



(f) B-MAC, Sensor task delay is 20ms



(g) TDMA, Sensor task delay is 30ms



(h) B-MAC, Sensor task delay is 30ms

Figure 6.7: Maximum possible sampling rate in case of different communication protocols, number of nodes, sensor internal task delays, and radio packet size

job there is one instance of AM, and jobs can be made up of many tasks. YARN can use different policies for dispatching jobs to AMs based on deadlines, priorities, and arrival times of jobs. Note that the current implementation of YARN does not support preemption.

The author in [62] developed ReGen (Rebeca Generator) as a Java application that generates Timed Rebeca codes for Hadoop YARN models with different policy, job arrival patterns, and job length pattern parameters. He also ran the back-end analyzer of Timed Rebeca and gathered results for comparing the efficiency of different dispatch and job eviction policies in YARN clusters. To this end, a Timed Rebeca code is generated using the following structure, for each configuration.

- ResourceManager class
  - Known rebecs
  - State variables
  - Constructor
  - Message server for checking queues every timeunit
    - Count deadline misses in the queues
    - Sort the incoming queues based on deadline
    - Dispatch production jobs to free AMs
    - Preempt research jobs for production jobs if no free AMs Dispatch checkpoints to free AMs
    - Preempt high laxity/deadline jobs for low laxity/deadline checkpoints
    - Dispatch research jobs to free AMs
    - Preempt high laxity/deadline jobs for low laxity/deadline incoming jobs
    - Unlock AM mutexes
    - Determine the number of incoming jobs
    - Determine the length of each incoming job, compute deadline and put job in queue
    - Determine priority of each incoming job, and modify
    - length of high priority jobs if needed
  - Message server for updates from AMs
    - Process whether deadline was missed or job completed
    - Compute the margin between deadline and time remaining
    - Update the RMs information on the AM
  - Message server for checkpoints from AMs
    - Store the deadline and time remaining
    - If not possible, count checkpoint queue overflow
    - Update the RMs information on the AM
- AppMaster class
  - Known rebecs
  - State variables

Policy	Mean Deadline Misses	Job Req. Success Rate	AM Jobs Success Rate
EDF	25.96	70.37%	77.20%
FIFO	31.16	64.68%	86.79%
MDF	30.55	66.52%	99.89%
Priority	25.65	71.07%	81.36%

Table 6.1: Comparing the KPIs of scheduling policies in a configuration in favor of EDF

  - Constructor
  - Message server for running jobs
    * If a research job is running, preempt and send a checkpoint
    * If no job is running, send an update to the RM
    * Start processing by sending a process message to self if receiveing a new job
  - Message server for processing jobs
    * If job completes or deadline runs out, send update to RM
    * Otherwise, continue processing by sending a process message to self

- Main function
  - Create actors

Analyzing the Timed Rebeca model of each configuration, the mean deadline misses, job request success rate, and AM job success rate is computed as KPIs. It is assumed that deadline misses are the sum of jobs that missed their deadlines while running and jobs that missed their deadlines while waiting in the queue. The job request success rate is the ratio of jobs completed that were submitted to the RM. The AM job success rate is the ratio of jobs completed that were submitted from the RM to the AMs. The authors presented a wide range of analysis varying the scheduling policy and model configuration. In Table 6.1 and Figure 6.8 we presented the results of analyzing EDF scheduling policy in the case that configuration is in favor of EDF. In this experiment, the length of submitted jobs is nondeterministically chosen from 1 to 6, and 20 percents of the jobs have high priority.

6.3 Functional and Performance Analysis of NoCs

Through technology shrinkage, multiprocessor systems on chips have emerged as a viable solution to the growing complexity. Network on chip (NoC) is a promising interconnection paradigm for these systems. Globally Asynchronous Locally Synchronous (GALS) approach [73] has gained attention in designing of NOCs. However, GALS-based NoCs encounter two significant challenges: (1) functional verification for making sure that the desired properties are met, and (2) performance evaluation in various stages of the design process for choosing the appropriate design parameters.

As fabrication cost is high, it is desirable to tackle the mentioned challenges prior to the production of the first prototype and even in the early stages of design process. So, it is crucial to have models of the systems with proper details, to perform

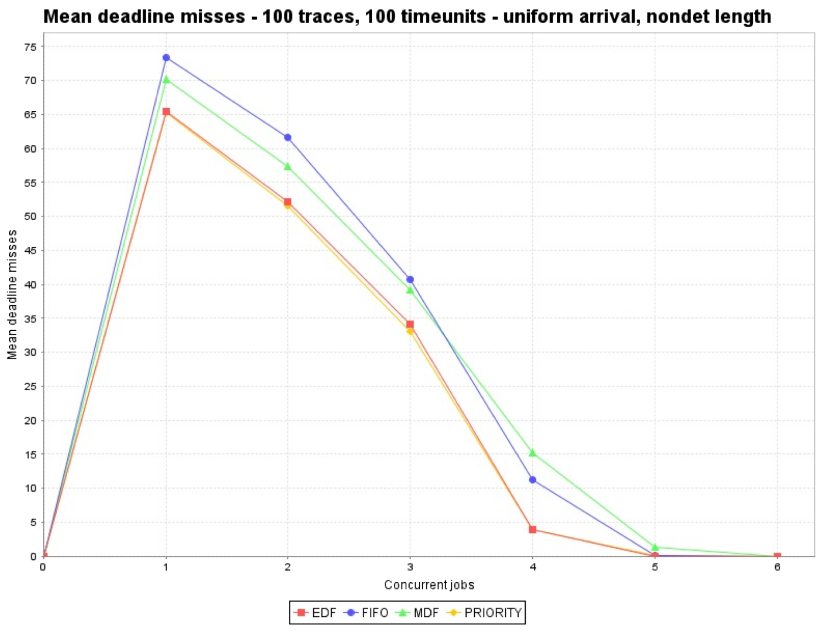


Figure 6.8: Mean deadline misses for a scenario in favor of EDF in the case of changing the number of concurrent jobs

design-time model-based analysis. Prior to this work, there was no model for GALS NoCs containing appropriate details which are needed for the verification against both functional and performance properties. In this work, authors used model checking to confront both challenges simultaneously and realize the use of one model for verifying both functional and performance properties. The proposed model can be used for estimating end-to-end packet latency, as well as checking functional properties, e.g. deadlock freedom and successful delivery of packets to destinations. Functional and timing behaviors, communication protocol, routing and scheduling algorithms are considered in the proposed model.

One of the major issues in asynchronous systems is the lack of a reference clock, results in interleaved executions of processes. Therefore, in GALS NoCs, a sent packet might be delayed by a different number of disrupting packets and may have various end-to-end latencies, in addition to the network congestion based packets disruption. Thus, timing analysis in these systems is required to make valid design decisions for avoiding deadline misses for traveling packets. For analysis of such systems, it is essential to consider all possible behaviors of the systems. However, existing work based on simulation techniques cannot be applied for exhaustive verification. Also, ensuring correctness to a certain degree using simulation is highly time-consuming and infeasible in some cases. Formal methods and more specifically model checking are alternative approaches that can be used for both performance evaluation and correctness checking considering exhaustive search in the behaviors of system [74]. Using Timed Rebeca model with formal verification support using Afra allowed authors to model the asynchronous behavior of GALS NoC naturally. Similarities between the computational model of Timed Rebeca and GALS NoCs, lead authors to a natural and easy to understand model. Due to the asynchronous communication, applying an exhaustive verification on large NoCs may result in the state space explosion. To alleviate this problem they presented a method based on compositional verification. The method computes the maximum end-to-end latency in GALS NoCs with XY routing algorithm in two steps. It breaks the path of a packet to its destination into horizontal and vertical sub-paths and then performs latency estimation in each sub-path separately. Finally, the results for each sub-path are combined to get latency estimation of the whole path. To illustrate the applicability of the proposed approaches, the authors modeled and analyzed ASPIN (Asynchronous Scalable Packet-switching Integrated Network) [48], which is a fully asynchronous two-dimensional GALS NoC design using XY routing algorithm.

6.3.1 GALS NOC Model in Timed Rebeca

Focusing on the above properties the topology of the communication, routing algorithm, buffer status, and communication protocol have to be considered in the Time Rebeca model of GALS NoCs, together with the timing behaviors. Considering these details, the full state space for a specific traffic pattern is produced and analyzed against deadlock-freedom and the successful arrival of each packet properties. In order to estimate maximum end-to-end packet latency, buffer delays and channel delays are also considered in the model. The elements of a GALS NoC can be modeled as the following in Timed Rebeca.

- **Router:** Each router is mapped to an actor which communicates with its neighboring routers through message passing. The processing delay of a router is modeled by “delay” statement. The routing algorithm of a router is implemented as a body of a message server, called `reqSend`. Based on the port which a packet entered to the router, it decides to which port the packet has to be sent. Since ASPIN uses Round Robin scheduling algorithm, the scheduler of Timed Rebeca is used as the scheduler of the model.

- **Buffer:** Router buffers are modeled as an array of elements (packets). In the proposed model, the actors’ queues model buffers, and the number of packets in the buffer is kept track using a counter state variable as the number of elements in buffers. Having this information, adaptive and dynamic routings can be modeled. Delay of writing and reading to/from buffers are modeled by “after” specifiers.

- **Packet:** Abstracting away the details, a packet is modeled by a pair of its identifier and destination.

- **Channel (Link):** Channels are modeled by message passing. Delay of passing through a channel is modeled by “after” specifier.

- **Communication protocol:** The body of message servers can be changed to develop different communication protocols of GALS NoCs.

Listing 6.7 shows the structure of ASPIN in Timed Rebeca. Based on the instantiated actors in the main part of Listing 6.7, a 4x4 NoC is defined. Packets of the network are generated by the `Manager` actor. Packets can be generated at any time and traffic pattern.

Packets are transferred through channels, using four-phase handshake communication protocol. The channel functionalities are modeled by means of message passing in the Timed Rebeca model. Four-Phase handshake protocol is modeled using three message servers: `reqSend`, `giveAck`, and `getAck`. A router calls its `reqSend` message server to send a request to its neighbors; `reqSend` requires a destination address that shows the destination of the packet as a parameter. The function XY-routing selects which neighboring router the packet should be sent to, and then giveAck message server of the selected neighbor router is called.

Listing 6.7: The structure of the ASPIN architecture GALS NoC in Timed Rebeca

```
 1 reactiveclass Manager {
 2   knownrebecs {
 3     Router r00, r10, ..., r33;
 4   }
 5   msgsrv generateTraffic() { ... }
 6 }
 7 reactiveclass Router {
 8   knownrebecs {
 9     Router North, East, South, West;
10   }
11   statevars {
12     byte Xid, Yid;
13     byte[4] bufNum;
14     boolean[4] full, enable, outMutex;
15   }
16   Router(byte X, byte Y) { ... }
17   msgsrv init() { ... }
18   msgsrv getAck() { ... }
19   msgsrv reqSend(byte Xtarget, byte Ytarget) { ... }
20   msgsrv giveAck(byte Xtarget, byte Ytarget) { ... }
21 }
22 main {
23   Manager m(r00,r10, ... ,r33): ();
24   Router r00(r03,r10,r01,r30): (0,0);
25   Router r10(r13,r20,r11,r00): (1,0);
26   ...
27   Router r33(r32,r03,r30,r23): (3,3);
28 }
```

Following four-phase protocol, request signal of the sender is raised until it receives an acknowledgment signal. While waiting for the acknowledgment signal, the router cannot send any packet from that port. We assigned `enable` boolean variable to each output port of the router to model this functionality. Whenever a packet is sent from an output port, the corresponding `enable` becomes false (line 7). For sending the acknowledgment signal the `getAck` message server is invoked. The `giveAck` message server first checks the address of the destination of the packet. If the address is the same as the current router, then the actor will consume the packet (line 14). Otherwise, it checks if the corresponding input buffer has enough capacity to store the packet, if there is enough capacity the packet will be stored and an acknowledgment is sent to the sender by calling its `getAck` message server (line 19). Then, it will be sent to the other neighboring routers calling their `reqSend` message server. If the buffer is full the packet will not be stored in the receiver's buffer and the sender should wait for an acknowledgment from the receiver until the buffer has an empty space.

Listing 6.8: A part of the body of a router of the ASPIN architecture

```
 1 msgsrv reqSend(byte Xtarg, byte Ytarg) {
 2   if(Xtarg > Xid) {
 3     byte leavingDirection = ...;
 4     if(outMutex[leavingDirection]) {
 5       East.giveAck(Xtarg, Ytarg) after(50);
 6       outMutex[leavingDirection] = false;
 7       enable[leavingDirection] = false;
 8     } else
 9       self.reqSend(Xtarg, Ytarg) after(50);
10   } else if(Xtarg < Xid) { ... }
11   ...
12 }
13 msgsrv giveAck(byte Xtarg, byte Ytarg) {
14   if(Xtarg == Xid && Ytarg == Yid) {
15     //Consume the packet
16   } else if(!(Xtarg == Xid && Ytarg == Yid)) {
17     byte enteranceDirection = ...;
18     bufNum[enteranceDirection]++;
19     ((Router)sender).getAck() deadline(50);
20     self.reqSend(Xtarg, Ytarg) after(100);
21   }
22 }
23 msgsrv getAck(int directionS){
24   enable[directionS] = true;
25   bufNum[directionS] = (byte)bufNum[directionS] - 1;
26   full[directionS] = false;
27   if (sender == N) {
28     outMutex[0] = true;
29   } else if (sender == E){
30     outMutex[1] = true;
31   } else if (sender == S){
32     outMutex[2] = true;
33   } else if (sender == W){
34     outMutex[3] = true;
35   }
36 }
```

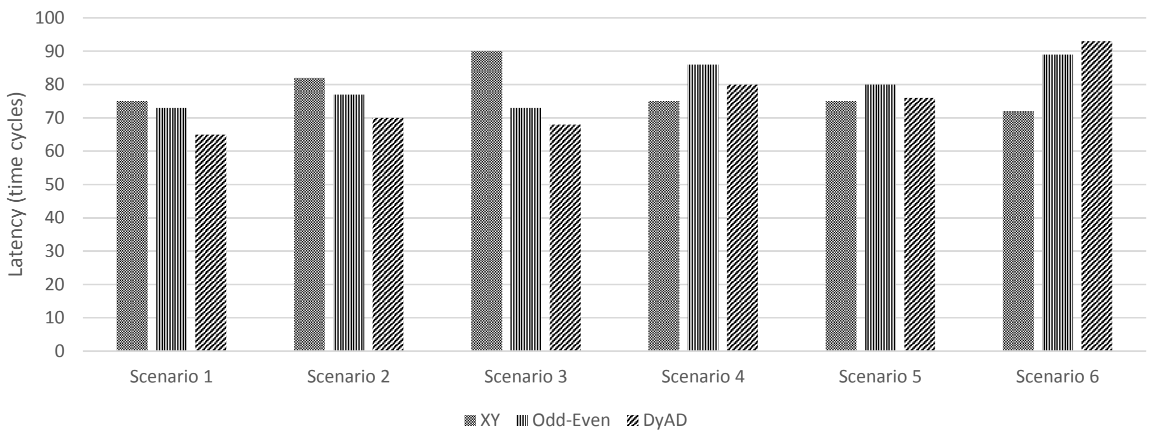


Figure 6.9: Comparing the packet latency in different routing algorithms

6.3.2 Experimental Results

For the case of functional correctness, authors consider deadlock-freedom and correct packet sending properties. The first property automatically checked by Afra. For the second one, they considered two correctness properties. The first property is to check whether a packet sent from a source has reached its destination. For checking this property `packetSent` and `packetReceived` state variables are added to the router model and $\mathbf{G}\,(packetSent \rightarrow \mathbf{F}\, packetReceived)$ LTL property is defined. This way, if a packet is sent then `packetSent` is set to true, and if that packet is reached to its destination, `packetRecieved` of the destination actor is set to true. In addition to this property, it is needed to check whether all packets in the network were sent. To do so, `allSent` state variable is declared and it is set to `true`if the number of sent packets is the same as the number of the existing packets in the NoC. This way, the property is defined by the LTL property $\mathbf{F}\, allSent$.

In addition to the functional correctness, authors performed some performance evaluation on the model. As the first goal, they compared the performance of using *Odd-Even*, *XY*, and *DyAD* routing algorithms in the presence of six typical scenarios. As shown in Figure 6.9 in the first three scenarios, *DyAD* and *Odd-Even* avoid congestion by monitoring their neighbors and thus have less end to end packet latency. Also, *DyAD* has better results than *Odd-Even* because it exploits the low latency of deterministic routings in low traffics. The second three scenarios show distributed traffic in which disrupting packets exist in all possible directions, by which the target packet can get to its destination. These scenarios investigate the impact of low latency of deterministic routings which is the result of their simplicity in contrast to adaptive routing algorithms. As shown in Figure 6.9, *XY* shows the best performance, as stated in [75], because XY has a global and long-term knowledge about the traffic, it exhibits better results than the others.

As the second goal, the authors used HSPICE simulation of ASPIN model to validate the proposed formal and compositional method. The results of the simulation match the results from formal and compositional methods while effort for simulation is much higher. The reason is because of the more details that are considered in the HSPICE model. The similarity of the results shows that in spite of the fact that our methods are not considering the same amount of details they are still eligible for the required measurements. By this comparison we show how using our method in the early stages of design can help the designer to make suitable architectural decisions

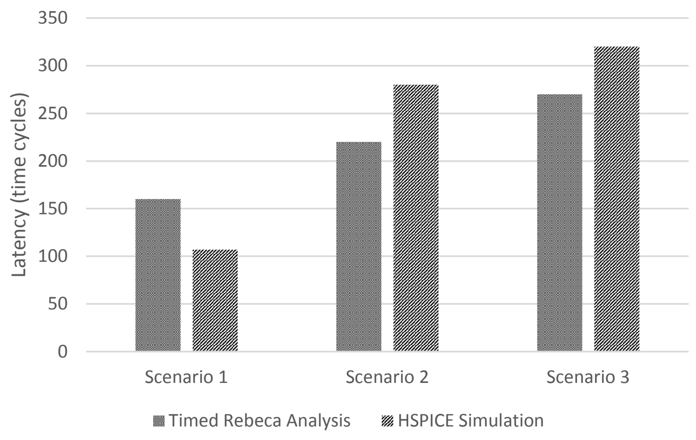


Figure 6.10: Comparing the timing behavior of the Timed Rebeca model with the HSPISE implementation of ASPIN architecture GALS NoC

according to the performance parameters of the system. To compare the results of the formal approach to the simulation results, they extracted buffer read and buffer write delays from HSPICE simulation of ASPIN for 32nm CMOS technology and normalized to C-element Muller gate to convert them from Float to Integer. Delay of read and write operations on a buffer was considered 19 and 6 time units, respectively. Producing and consuming a packet in a core cause delays of 10 and 19 time units, respectively. They considered capacity of two entries for input buffers and one for output buffers. Figure 6.10 shows results for estimating the maximum latency of packets in the specified scenarios compared to results of the simulation and results of estimation using our compositional method.

Chapter 7

Previous Work on Analyzing Timed Rebeca Models

As the earliest attempts for analyzing Timed Rebeca models, an approach is developed for model checking of Timed Rebeca models using transformation from Timed Rebeca models to networks of timed automata. The resulting timed automata are model checked against TCTL properties using the UPPAAL toolset. The details of this approach is presented in Section 7.1. Another work on model checking of Timed Rebeca is based on mapping timed actors to Realtime Maude. This enables a formal model-based methodology which combines the convenience of intuitive modeling in timed actors with formal verification of Realtime Maude. Realtime 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. The details of this approach is presented in Section 7.2.

7.1 Semantics of Timed Rebeca In Timed Automata

Timed automata [11] is one of the most widely used modeling languages for modeling of realtime systems and is supported by UPPAAL toolset[1]. Because of successful results in modeling and verification of different types of realtime systems, including [27]–[29], UPPAAL was chosen as the back-end model checker for verification of Timed Rebeca models. Therefore, a mapping from Timed Rebeca models to networks of timed automata was proposed in [5]. We tried to implement the optimized mapping of this approach to achieve the smallest possible state space of Timed Rebeca models. Our experiments showed that modeling of asynchronous message passing between actors using synchronous communication of timed automata results in large state spaces, even for small case studies.

In the proposed mapping, each actor is mapped into two timed automata, called the *rebec-behavior* automaton and *rebec-bag* automaton. Additionally, one timed automaton, called the *after-handler* automaton, is defined to handle the behavior of *after* primitive for all actors. To reduce the number of clocks in the model, one global clock pool is defined. This clock pool contains a predefined number of clocks. When a timed automaton requires a clock, it picks a clock from the pool using *selectClock*

[1]http://www.uppaal.org

function. To illustrate the mapping, the timed automata for the ticket service model of Figure 2.3 is described as the running example in the rest of this section.

7.1.1 Rebec-Behavior Automaton

The *rebec-behavior* automaton models the behavior of an actor according to the statements of its message servers and valuations of state variables. The state variables of each actor are mapped into variables of its corresponding *rebec-behavior* automaton. After receiving a message in *rebec-behavior* automaton of each actor, the first transition checks the received message and based on that the behavior of the corresponding message server is modeled in the succeeding transitions. To model the behavior of a message server, its statements are mapped to transitions of timed automata as described in the following. In the following items, we assume that each statement is modeled by some outgoing transition from state S. The label of the outgoing transitions are in form of tuple of (a, g, c) in which a is the action, g is the guard, and c is the set of clocks that are reset during the transition.

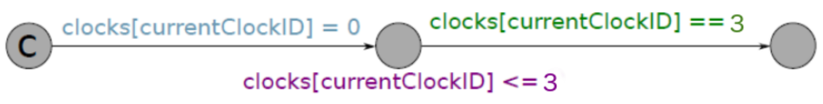
- Conditional statement $if(cond)\{\cdots\}$: is mapped to transition t from S to S'. The label of t is set to $(-, g, -)$ in which g is the same as the *cond* expression of the conditional statement.

- Assignment statement $var = exp$: is mapped to transition t from S to S'. The label of t is set to $(a, -, -)$ in which a assigns the value of *exp* to the variable *var*.

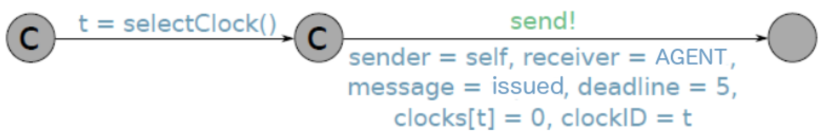
- Non-Deterministic assignment statement $var =?(exp_1, \cdots, exp_n)$: is mapped to transitions $t_1, \cdots t_n$ from S to $S'_1 \cdots S'_n$. The label of t_i is set to $(a_i, -, -)$ in which a_i assigns the value of exp_i to the variable *var*.

- Delay statement $delay(d)$: in mapping of a delay statement one clock and one additional state is required. In this case, the mapping results in transition t_1 from S to S' and transition t_2 from S' to S''. The label of t_1 is set to $(a, -, -)$ in which a is the action of selecting a clock from the global clock pool. The clock is selected from pool of clocks using *selectClock* function. Assume that cl is the selected clock. The guard of state S' is set to $cl \leq d$ and the label of t_2 is set to $(-, g, -)$ in which g equals to $cl = d$. Based on this mapping, the active state of the timed automaton is forced to stay in S' for d units of time. Mapping of delay statement in line 12 of Listing 2.2 is shown in Figure 7.1a.

- Sending message statements: For message sending, one clock is attached to each message to show its sent time. This clock is used for checking the release time and deadline of messages. The clock is returned to the pool when the message is delivered to the *rebec-behavior* automaton for execution. The channel *send* is used if the message is sent immediately and the channel *after*, if the sent message is associated with a value for *after* primitive. Messages which are sent via the *send* channel are directly put in the *rebec-bag* of their receivers. Messages which are sent via the *after* channel are put in an internal buffer in the *after-handler* automaton. Mapping of sending message in line 13 of Listing 2.2 is shown in Figure 7.1b.


clocks[currentClockID] = 0
clocks[currentClockID] == 3
clocks[currentClockID] <= 3
C

(a) Mapping from delay statement of line 11 of Listing 2.2 to timed automata in case of having delay of 3 units of time


t = selectClock()
send!
sender = self, receiver = AGENT,
message = issued, deadline = 5,
clocks[t] = 0, clockID = t
C
C

(b) Mapping from sending message statement of line 12 of Listing 2.2 to timed automata

Figure 7.1: Implementation of delay and sending message statements of Listing 2.2 in timed automata.

Upon completion of the execution of a message server, a *rebec-behavior* automaton will back to its initial state and the outgoing transition is requesting for the next message. To illustrate the mapping of reactive classes to *rebec-behavior* automata, the *rebec-behavior* automaton of `Customer` reactive class of Listing 2.2 is shown in Figures 7.2. For the transition from *S0* to *S1* the action of the transition is set to receiving a message from the *rebec-bag* automaton. Based on the message name, which is stored in *currentMessage*, the execution point is directed from *S1* to one of *S2* or *S3*. In this timed automaton, the execution of the *ticketIssued* message server is started from *S2* and the execution of the *try* message server is started from *S3*. The *rebec-behavior* automata of `Agent` and `TicketService` are shown in Figures 7.3 and 7.4 respectively.

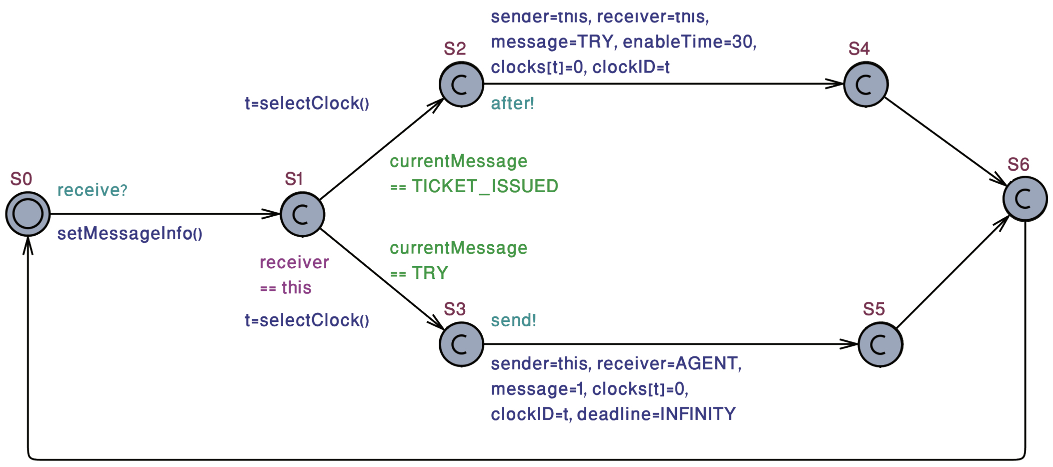


Figure 7.2: The *rebec-behavior* automaton of Customer reactive class of Listing 2.2

7.1.2 Rebec-Bag Automaton

The *rebec-bag* automaton handles the behavior of the message bag of an action using an internal buffer, called *messageQ* as shown in Figure 7.5. The messages which are sent to an actor are received by *rebec-bag*, regardless of the state of the corresponding *rebec-behavior* automaton. Then, *rebec-bag* automaton delivers received messages upon the requests of its corresponding *rebec-behavior* automaton. Additionally, the *rebec-*

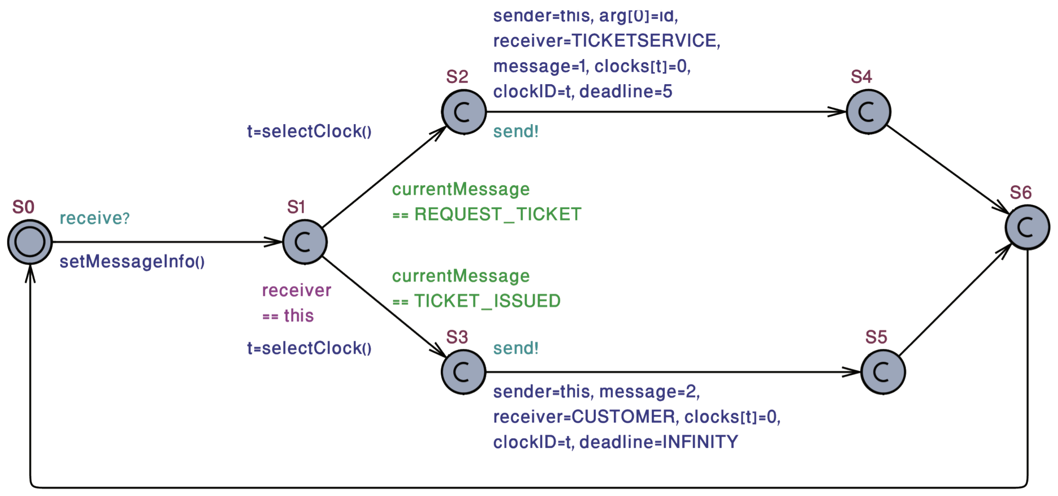


Figure 7.3: The *rebec-behavior* timed automaton of Agent reactive class of Listing 2.2

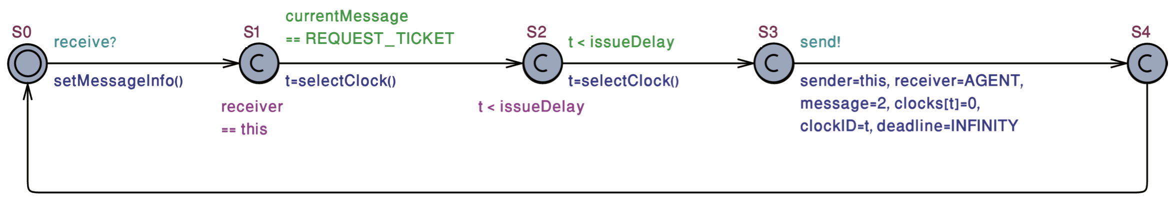


Figure 7.4: The *rebec-behavior* timed automaton of TicketService reactive class of Listing 2.2

bag automaton is responsible for handling message deadlines. Figure 7.5 shows the timed automaton of *rebec-bag*. As shown in Figure 7.5, *rebec-bag* inserts the incoming messages into its buffer (transition from *S1* to *S3*), discards messages with passed deadlines from its buffer (self loop transition in *S1*), and extracts the messages from its buffer and delivers them (transition from *S1* to *S2*). Extracting a message from the buffer is done by *shift* function which is used as the update function of the transition from *S2* to *S1*.

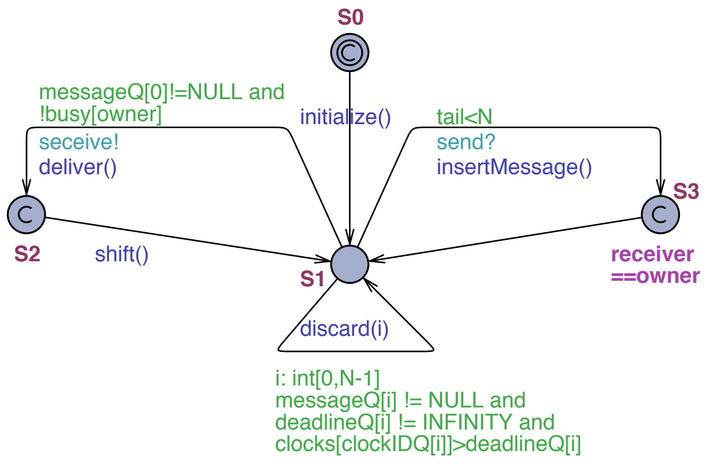


Figure 7.5: Timed automaton of *rebec-bag*

7.1.3 After-Handler Automaton

The *after-handler* automaton handles the messages which should be delivered to *rebec-bag* in the future (messages which are sent by *after* primitive). The *after-handler* automaton accepts messages and put them into its buffer until the release time of them. When a message in the buffer of *after-handler* is released, it is sent to its corresponding *rebec-bag* automaton. Figure 7.6 shows the timed automaton of *after-handler*. As shown in Figure 7.6, the incoming messages are inserted into its buffer by transition from *S1* to *S2*. The messages are extracted from the buffer of *after-handler* and are delivered in their release times by self loop transition on *S1*.

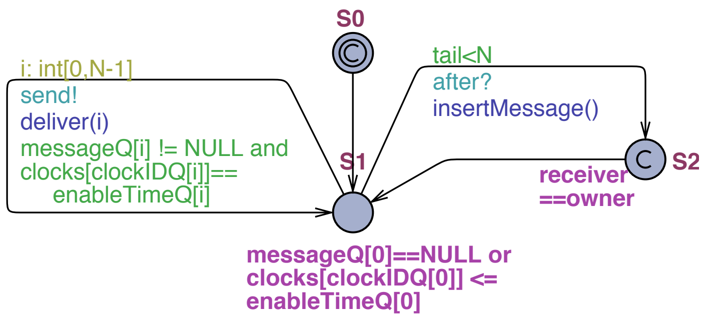


Figure 7.6: Timed automaton of after-handler

7.1.4 Analysis of Network of Timed Automata

The parallel composition of the resulting timed automata and the schedulability analysis of the model is done using UPPAAL [30].

Modeling of asynchronous message passing between actors using synchronous communication of automata increases the number of states dramatically [31]. However, some techniques can be applied, like using *committed states*, to reduce the number of states of the resulting region transition system. In the parallel composition of the network of timed automata, related to an actor model, different automata need to synchronize on the following four actions.

1. A message is sent (on *after* or *send* channels).
2. A message is taken from the message bag to start to execute.
3. A transition modeling a *delay* statement is reached in an automaton.
4. It is the time for a sent message to be delivered to its receiver.

This will increase the number of states and is the main reason that using a network of timed automata is not an ideal approach for models in which elements communicate by asynchronous message passing. Additionally, the time consumption of the analysis of the models is increased dramatically by increasing the number of actors of the model, as the number of clocks grows linearly with the number of actors.

To reduce the number of states, some reduction techniques are proposed for verification of timed automata models. In [32] authors proved that instead of global clock synchronization among all timed automata in a network of timed automata, synchronization of clocks is only required between two timed automata when they want to communicate. Therefore, they allowed clocks to increase independently and they only being synchronized when two timed automata want to communicate. This way, the third item of the synchronization actions (see above) is omitted and other three synchronization points need to be considered in the parallel composition of timed automata. This work is continued in [33] by applying the proposed reduction technique in the model checking of timed extension of LTL. Later in [34] a new approach is suggested for partial order reduction in component-based systems. This approach is only applicable for systems where components work in three phases: reading from their input ports, performing their internal operations, and writing the result of the internal operations to their output ports. Based on these phases, the progress of time and inter-component timing complexity of phase two can be modeled using a single clock. This reduction technique is not applicable for timed automata derived from Timed Rebeca models. As phases two and three cannot be separated in Timed Rebeca models. It is because of the fact that because message passings and internal actions of actors are interleaved in Timed Rebeca.

7.2 Semantics of Timed Rebeca in Realtime Maude

In this section, we explain how the semantics of Timed Rebeca have been formalized in Realtime Maude in an object-oriented style as presented in [4]. Note that only an overview of this semantics is presented in this section and the detail description can be found in [4].

In the Realtime Maude semantics of Timed Rebeca, each actor is represented by an object instance of class `Rebec`, and each undelivered message is represented by a ("delayed") message. As a result, a state of an actor system consists of the actors and a set of undelivered messages.

When defining the semantics of a language in rewriting logic, each event/action—such as sending a message or executing an assignment statement—is typically formalized by a rewrite step (see, e.g., [35]). However, that approach leads to many interleavings that may render exhaustive state space analysis unfeasible. One novel contribution of this semantics is in considering the fact that the execution of a message server cannot be preempted in Timed Rebeca, which allows "grouping together" all consecutive deterministic instantaneous actions and executing them in a single rewrite step.

Section 7.2.1 explains the statics: how a Timed Rebeca state is represented in Realtime Maude, and how a Timed Rebeca model is translated into Realtime Maude. Section 7.2.2 formalizes the instantaneous dynamic behaviors of Timed Rebeca in Realtime Maude, and Section 7.2.3 formalizes the timed behaviors of Timed Rebeca. The entire executable Realtime Maude semantics of Timed Rebeca is available at Rebeca homepage [2].

7.2.1 Representing Timed Rebeca Models in Realtime Maude

In the Realtime Maude representation of a Timed Rebeca model the following information must be kept track: (i) the actors in their current states; (ii) the *declarations* of the (message servers of the) reactive classes; and (iii) the set of as-yet undelivered messages.

[2]http://rebeca-lang.org/allprojects/TR2RTMaude

Representing Rebecs An actor is modeled in Realtime Maude by an object instance of the following class `Rebec`:

```
class Rebec | classId : ClassName,
              stateVars : Valuation,
              queue : MsgList,
              toExecute : Statements .
```

The attribute `classId` denotes the name of the reactive class of the actor. The attribute `stateVars` is a valuation that maps the state variables of the actor to their current values and also maps the name of known rebec to their corresponding identifiers. To avoid name clashes, the class name together with '`-`' is added to the beginning of the state variable name. The message queue of an actor is stored in the `queue` attribute as a '`::`'-separated list of messages. Finally, the attribute `toExecute` denotes the remaining statements that the actor has to execute during the execution of the current message server.

7.2.1.1 Valuations

Timed Rebeca actors may have state variables of three types: Integer, Boolean, and state variables that refer to other actors. In the Realtime Maude model, a sort `Rid` for actor names/identifiers is assumed as the name of the third type:

State variables of sort Integer in the Timed Rebeca model are represented by a constant of sort `IntVar` in Realtime Maude; Boolean state variables and variables pointing to other actors are modeled as constants of sort `BoolVar` and `RidVar`, respectively.

The sort `Valuation` represents mappings from variables to their current values as a set of terms of the form *var-name* `|->` *value*, as the following:

```
sort Valuation .
op _|->_ : IntVar Int -> Valuation [ctor] .
op _|->_ : BoolVar Bool -> Valuation [ctor] .
op _|->_ : RidVar Rid -> Valuation [ctor] .
op _ _ : Valuation Valuation -> Valuation [ctor assoc comm id:
             emptyValuation] .
op emptyValuation : -> Valuation [ctor] .
```

7.2.1.2 Messages and Message Lists

Communication between actors takes place when an actor sends a message to another actor (or to itself). The message is put into the multiset of undelivered messages until its message delay ends. It is then delivered to the receiver's message queue. *Delivered messages* are declared by:

```
msg _with_from_to_deadline_ : MsgHeader Valuation Rid Rid TimeInf -> Msg .
```

So, a delivered message

msgHeader `with` *parameters* `from` *snd* `to` *rcvr* `deadline` *relativeDeadline*

contains a header (the message name), its arguments, the identity of the sender actor, the identity of the receiver, and the time remaining until the expiration of the message. A message queue is a `::`-separated *list* of delivered messages:

```
sort MsgList .
subsort Msg < MsgList .
op nil : -> MsgList [ctor] .
op _::_ : MsgList MsgList -> MsgList [ctor assoc id: nil] .
```

Program Statements Since the definitions and behaviors of program statements of Timed Rebeca are very similar to their equivalent statements in the ordinary programming languages, we omitted their definition from this part. The detailed representation of Timed Rebeca statements in Realtime Maude is presented in [4].

7.2.1.3 Representing Message Servers

A Timed Rebeca model defines reactive classes with their message servers. Since the message servers do not change dynamically, they are *not* include in the state. Instead, the function

```
op msgServer : ClassName MsgHeader -> StatementList .
```

which defines the message server for each reactive class and message header. The message names are prefixed by the class name to prevent naming conflicts. This function also gives the body of the constructors by treating them as messages with the header `constructor`. This way, `msgServer(Heater, constructor)` equals

```
(on := false) ;
(send run with noArg to self deadline INF after 0)
```

7.2.1.4 Messages in Transit

Messages in transit have the form `dly`(m, t), where m is a message as described in Section 7.2.1.2 and t is the remaining delay of the message. Such messages are declared as the following.

```
sort DlyMsg .
subsort Msg < DlyMsg < NEMsgConfiguration .
op dly : Msg Time -> DlyMsg [ctor right id: 0] .
```

When the remaining delay of a delayed message becomes 0, the message becomes "undelayed".

The Realtime Maude state representing a Timed Rebeca state during the execution of a Timed Rebeca model, therefore, contains a message `dly`(m, d) for each undelivered message m with *remaining* messaging delay ("after") d.

7.2.2 Instantaneous Dynamics

This section presents the semantics of the "instantaneous actions" of Timed Rebeca. As mentioned, all consecutive instantaneous deterministic actions are performed together "atomically" in one rewrite step.

Receiving a Message The following rewrite rule causes an *undelayed* message to be delivered; i.e., removed from the set of undelivered messages in the state and appended to the message queue of the receiving actor:

```
rl [readMessage] :
   (M with ARGS from O to O' deadline DL)
   < O' : Rebec | queue : MSGLIST >
  =>
   < O' : Rebec | queue : MSGLIST :: (M with ARGS from O deadline DL) > .
```

Serving a Message In the following rewrite rule, an *idle* actor (`toExecute` is `noStatement`) takes the first message from its queue and starts executing the statements in the corresponding message server by putting them into its `toExecute` attribute. In addition, the actual arguments in the message to be "served" are included in the execution environment `stateVars`. To clean up at the end of the execution, a new statement `removeVars` is appended to the end of the statements in the message server:

```
rl [takeMessage] :
   < O : Rebec | stateVars : SVARS,
                 queue : (M with VAL from O' deadline DL) :: MSGLIST,
                 classId : CN,
                 toExecute : noStatement >
  =>
   < O : Rebec | stateVars : SVARS VAL (sender |-> O'),
                 queue : MSGLIST,
                 toExecute : msgServer(CN,M) ;
                 removeVars(VAL (sender |-> O')) > .
```

Note that the mapping `SVARS VAL (sender |-> O')` contains the valuation of the state variables of the receiving actor (`SVARS`), together with the valuation of actual parameters (`VAL`) and the mapping of the keyword `sender` to the sender of the message (`sender |-> O'`).

7.2.2.1 Executing Immediate Statements

As already mentioned, when executing a message server, it is possible to execute as many statements as possible together instead of executing them one-by-one, avoiding a lot of unnecessary interleavings. However, it may not be possible to execute the entire body of a message server atomically, because a delay statement forces the execution to stop. Furthermore, since nondeterministic assignment leads to multiple successor states, it would make this semantic model too complex. As a result, these two statements are considered as special cases that cannot be grouped by the other statements.

The basic idea is to define three functions that act on a sequence of statements, the first of which is immediate, and specify the result of executing the whole sequence as (i) the new state variables mapping, (ii) new messages to be sent, and (iii) the "leftover" statements that could not be executed as a group. The parameters to these functions are: (1) the identifier of the executing actor, (2) the state variables and the known rebecs of the actor, (3) the list of statements to execute in a group, (4) the name of the executing class, and (5) a set of undelivered messages. The last parameter is used especially by the `newMsgs` function to append the new messages to the current configuration.

The rewrite rule for executing as many consecutive immediate statements as possible in one step can be executed only if the `toExecute` of the actor starts with an immediate statement. The actor then applies the accumulated changes to its local state at once and changes its `toExecute` to what remains after executing as many immediate statements as possible. Some new messages may have been generated that are added to the configuration. The computation performed by these functions goes on as long as possible; the functions "return" when there are no more statements to be performed or when a "non-immediate" statement must be performed.

7.2.2.2 Delay

When the executing actor encounters a delay statement, it evaluates the delay expression in the current valuation. Then it leaves the delay statement at the beginning of its `toExecute` *until* the remaining delay becomes 0, when the actor just continues with the next statement. Decreasing the remaining delay is done by the tick rule explained in Section 7.2.3:

```
crl [evaluateDelayExpression] :
    < O : Rebec | stateVars : SVARS, toExecute : delay(IE) ; SL >
   =>
    < O : Rebec | toExecute :  delay(evalIntExp(IE, SVARS)) ; SL  >
       if not (IE :: Int) .
```

7.2.3 Timed Behavior

The following "standard" object-oriented tick rule [12] is used to model that time *may* advance all the way until the next time when some "event" must take place:
The variable `SYSTEM` matches the entire state of the system. The function `mte` (*m*aximal *t*ime *e*lapse) determines how much time can advance in a given state. If an instantaneous rule is enabled, it must be executed immediately; therefore, `mte` of a state must be zero when an instantaneous rule is enabled in that state.

```
var SYSTEM : Configuration .
var T : Time .
crl [tick] : {SYSTEM}  =>  {elapsedTime(SYSTEM, mte(SYSTEM))} in time T
    if T <= mte(SYSTEM) .
```

The function `mte` is the minimum of the `mte` of each actor and each message in the soup. As mentioned above, the `mte` must be 0 when the actor has a statement to execute which does not have the form `delay`(i), for an integer $i > 0$; in the latter case, the `mte` equals i. If there are no statements to be executed, the `mte` equals 0 if the actor has a message in its queue, and equals the infinity value `INF` if the message queue is empty.

The function `elapsedTime` models the effect of time elapsed on a state as follows: the effect of time elapsed on an actor is that the remaining time until the message deadline is decreased according to the elapsed time for each message in the queue. Furthermore, the remaining delay of a delay statement being executed is also decreased

Configuration	Standard Semantics		TA Based Semantics	
	#States	Time	#States	Time
1 customer	9	1< sec	801	1< sec
2 customers	107	1< sec	19M	5 hours
3 customers	550	1< sec	-	24> hours †
4 customers	2.86K	1< sec	-	24> hours †
5 customers	16.9K	1< sec	-	24> hours †
6 customers	114K	2 secs	-	24> hours †
7 customers	884K	3 sec	-	24> hours†

Table 7.1: Model checking times and size of state spaces, using two different semantics for the ticket service system. The † sign on the reported time shows that model checking takes more than the time limit (24 hours).

according to the elapsed time. For messages traveling between actors, their remaining delays and deadline are decreased according to the elapsed time. In both cases, if the deadline expires before the message is treated, the message is purged (i.e., becomes the empty configuration `none`).

7.3 Experimental Results

To compare the efficiency of the proposed semantics, we have to prepare a set of case studies which are modeled by Timed Rebeca and compare the size of their state spaces which are generated based on the semantics described in timed automata, realtime Maude, and direct semantics of this chapter. As there is no systematic way to figure out recurrent behaviors of models in the Realtime Maude based semantics, generated transition systems are infinite (time goes to infinity); so, they can not be included in the comparisons. The comparison for two other semantics is depicted in the following sections. Note that for each case study we describe an overview of that case, present the source code, and discussed on gained efficiencies or weaknesses.

7.3.1 Ticket Service System

My first example is the model of a *Ticket Service* system. The overview of this example is presented in Section 2.1. We created the extended version of this model and varying in the number of customers.

The results of Table 7.1 make it clear that the approach of using timed automata for the analysis of Timed Rebeca models results in the state space explosion even for the small sized models. It is mainly because of the unnecessary interleavings among automata and variables valuations.

7.3.2 A Toxic Gas Sensing and Rescue System

The second example of this section is the toxic gas sensing model.The overview and detailed description of this example are presented in Section 2.3.2.

Configuration	Standard Semantics		TA Based Semantics	
	#States	Time	#States	Time
1 Sensor	75	1< sec	-	24> hours †
2 Sensors	389	1< sec	-	24> hours †
3 Sensors	2.15K	1 sec	-	24> hours †
4 Sensors	12.37K	12 secs	-	24> hours †

Table 7.2: Model checking times and size of state spaces, using two different semantics for the Gas Sensing system. The † sign on the reported time shows that model checking takes more than the time limit (24 hours).

Configuration	Standard Semantics		TA Based Semantics	
	#States	Time	#States	Time
2 Clients	107	1< sec	-	24> hours †
3 Clients	550	1< sec	-	24> hours †
4 Clients	2.86K	1< sec	-	24> hours †
5 Clients	16.9K	1< sec	-	24> hours †

Table 7.3: Model checking times and size of state spaces, using two different semantics for the CA protocol. The † sign on the reported time shows that model checking takes more than the time limit (24 hours).

The performance of using timed automata is worse than using the fine-grained semantics of Timed Rebeca, based on the values of Table 7.1. The time automata approach does not work for even the simplest configuration.

7.3.3 The IEEE 802.11 RTS/CTS Collision Avoidance Protocol

The next example is the simplified version of IEEE 802.11 RTS/CTS protocol for collision avoidance in wireless networks.The overview and detailed description of this example are presented in Section 2.3.3.

Table 7.3 clearly shows that the trend of values in this example is the same as the previous one and the approach of mapping to time automata does not work for them.

Chapter 8

Conclusion and Future Work

This dissertation contains a set of techniques and algorithms which are developed to improve the time and memory consumptions of model checking of timed actors.

At the first step, we propose techniques for improving model checking of discrete time actors. To this end, we introduce a new model checking algorithm, which is an optimal TCTL$_{\leq,\geq}$ 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 TCTL$_{=}$ model checking of timed actors.

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 TCTL model checking algorithm; however, using the new algorithm avoids it. Although the new TCTL model algorithm works more efficiently 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. Although we have used Timed Rebeca to illustrate the techniques presented in this thesis, our results are not limited to this language and can be applied to any modeling formalism with a discrete notion of time. Note that before the work of this thesis, timed actor models have to be transformed to Realtime Maude or timed automata for TCTL model checking, which do not give acceptable execution performances. But, using the work of this thesis, both of the old and improved TCTL model checking algorithms outperform time and memory consumption of TCTL model checking in comparison to using transformations to Realtime Maude or timed automata. This way, model checking of bigger transition systems is possible.

At the second step, we introduced the notion of floating time transition system (FTTS) for schedulability and deadlock freedom analysis of Timed Rebeca models. FTTS exploits the key features of Timed Rebeca. In summary, having no shared variables, no blocking send or receive, single-threaded actors, and non-preemptive execution of each message server give us an isolated message server execution, meaning that execution of a message server of an actor will not interfere with execution of a message server of another actors. Moreover, for checking schedulability and deadlock freedom we can focus only on events. In FTTS, each transition shows releasing an event, or in other words, execution of a message server of an actor. As a result, in each state, actors may have different local times, but the transitions still give us a correct order of release times of events of a specific actor. Experimental evidence support that direct model checking of Timed Rebeca models using FTTS decreases both model checking state space size and time consumption in comparison with the previously proposed approaches. For example, in the case of models with many concurrently executing actors, FTTS is up to 90% smaller than its corresponding transition system, which is generated based on the standard semantics of Timed Rebeca. Therefore, we can efficiently model check more complicated models. We also proved that there is a weak bisimulation relation between timed transitions system – which is generated based on the standard semantics of Timed Rebeca – and floating time transitions system – as the big-step semantics for Timed Rebeca. So, a modeler can use FTTS for verification of branching-time properties in addition to checking for schedulability and deadlock freedom. Note that, although using FTTS for actor models results in smaller transition systems, it only supports analysis of the event-based properties.

Note that our technique and the proofs are based on the actor model of computation where the interaction is solely based on asynchronous message passing between the components. So, they are generalized enough to be applied to computation models which have message-driven communication and autonomous objects as units of concurrency such as agent-based systems.

We also modeled an application of distributed realtime sensor and actuator networks (WSAN) in collaboration with two people from University of Illinois at Urbana-Champaign. WSAN applications are very sensitive to their configurations: the effects of even minor modifications to configurations must be analyzed. With little additional effort required on behalf of the application developer, my approach provides a much more accurate view of an WSAN application's behavior and its interaction with the operating system and distributed middle-ware services than can be obtained by the sort of informal analysis or trial-and-error methods commonly in use today. Our realistic—but admittedly limited—experimental results support the idea that the use of formal tools may result in more robust WSAN applications. This would greatly reduce development time as many potential problems with scheduling and resource utilization may be identified early.

Future Work. The work reported in this dissertation paves the way to several interesting avenues for future work. In particular, we can define a special kind of DTGs 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 which category of TCTL properties benefits more from the provided efficiency of the proposed algorithm.

In addition, we also go for defining FTTS for dense-time actors and relaxing the limitation of having no non-deterministic assignment in the body of message servers. This way, a wider range of systems can be modeled and analyzed using Timed Rebeca and Afra.

For the case of WSAN applications, we only address the schedulability analysis of WSAN components and did not consider the interference on the wireless channel issues (the details of communication protocols). We assume that there is a reliable wireless infrastructure in the application which provides guaranteed delivery of messages, which is a reasonable assumption for a wide range of deployments of structural health monitoring and control systems. However, this work can be extended by taking the details of communication protocols into account together with noises and unreliability of wireless communication which results in errors. This way, only Ether and RCD actors have to be modified to contain the details of the protocols. Note that the implementation of the chosen MAC protocol as well as the interaction of the processing hardware with the transmitter has to be added to RCD to take hardware and software into account and provide combined analysis of the underlying hardware infrastructure as well as the application software. Other different assumptions, including fairness in access to B-MAC, time drift of actors, and uncertainties, can be added. Note that extending the number of modeled MAC layer protocols also can be performed as a future work of this thesis. Comparing the efficiency of MAC protocols in different cases to study their characteristics will be one of the outcomes of this extension.

Bibliography

[1] C. Ptolemaeus, *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.

[2] M. Sirjani and E. Khamespanah, “On Time Actors”, in *Theory and Practice of Formal Methods*, ser. Lecture Notes in Computer Science, vol. 9660, Springer, 2016, pp. 373–392.

[3] A. H. Reynisson, M. Sirjani, L. Aceto, M. Cimini, A. Jafari, A. Ingólfsdóttir, and S. H. Sigurdarson, “Modelling and Simulation of Asynchronous Real-Time Systems Using Timed Rebeca”, *Sci. Comput. Program.*, vol. 89, pp. 41–68, 2014.

[4] Z. Sabahi-Kaviani, R. Khosravi, P. C. Ölveczky, E. Khamespanah, and M. Sirjani, “Formal Semantics and Efficient Analysis of Timed Rebeca in Real-Time Maude”, *Sci. Comput. Program.*, vol. 113, pp. 85–118, 2015.

[5] M.-J. Izadi, “An Actor Based Model for Modeling and Verification of Real-Time Systems”, Master’s thesis, University of Tehran, School of Electrical and Computer Engineering, Iran, 2010.

[6] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, “Modeling and Verification of Reactive Systems using Rebeca”, *Fundam. Inform.*, vol. 63, no. 4, pp. 385–410, 2004.

[7] M. Sirjani, F. S. de Boer, and A. Movaghar-Rahimabadi, “Modular Verification of a Component-Based Actor Language”, *J. UCS*, vol. 11, no. 10, pp. 1695–1717, 2005.

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

[9] G. A. Agha, *ACTORS - A Model of Concurrent Computation in Distributed Systems*, ser. MIT Press series in artificial intelligence. MIT Press, 1990.

[10] M. Sirjani and M. M. Jaghoori, “Ten Years of Analyzing Actors: Rebeca Experience”, in *Formal Modeling: Actors, Open Systems, Biological Systems*, 2011, pp. 20–56.

[11] R. Alur and D. L. Dill, “A Theory of Timed Automata”, *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[12] P. C. Ölveczky and J. Meseguer, “Semantics and Pragmatics of Real-Time Maude”, *Higher-Order and Symbolic Computation*, vol. 20, no. 1-2, pp. 161–196, 2007.

[13] W. Yi, “CCS + Time = An Interleaving Model for Real Time Systems”, in *ICALP*, 1991, pp. 217–228.

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

[15] F. S. de Boer, T. Chothia, and M. M. Jaghoori, “Modular Schedulability Analysis of Concurrent Objects in Creol”, in *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers*, F. Arbab and M. Sirjani, Eds., ser. Lecture Notes in Computer Science, vol. 5961, Springer, 2009, pp. 212–227.

[16] M. M. Jaghoori, F. S. de Boer, and M. Sirjani, “Task Scheduling in Rebeca”, in *NWPT*, 2007, pp. 16–18.

[17] E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. T. Tarifa, and P. Y. H. Wong, “Formal Modeling and Analysis of Resource Management for Cloud Architectures: An Industrial Case Study Using Real-Time ABS”, *Service Oriented Computing and Applications*, vol. 8, no. 4, pp. 323–339, 2014.

[18] E. Khamespanah, Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, and M. Izadi, “Timed-Rebeca Schedulability and Deadlock-Freedom Analysis Using Floating-Time Transition System”, in *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, G. A. Agha, R. H. Bordini, A. Marron, and A. Ricci, Eds., ACM, 2012, pp. 23–34.

[19] Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, P. C. Ölveczky, and E. Khamespanah, “Formal Semantics and Analysis of Timed Rebeca in Real-Time Maude”, in *FTSCS*, C. Artho and P. C. Ölveczky, Eds., ser. Communications in Computer and Information Science, vol. 419, Springer, 2013, pp. 178–194.

[20] J. Armstrong and E. T. Ab, “The Development of Erlang”, in *In Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ACM Press, 1997, pp. 196–203.

[21] *Erlang programming language homepage*, http://www.erlang.org.

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

[23] L.-Å. Fredlund and H. Svensson, “McErlang: A Model Checker for a Distributed Functional Programming Language”, in *ICFP*, R. Hinze and N. Ramsey, Eds., ACM, 2007, pp. 125–136.

[24] *Mcerlang homepage*, https://babel.ls.fi.upm.es/trac/McErlang/.

[25] G. Agha and C. Hewitt, “Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming”, in *Research Directions in Object-Oriented Programming*, 1987, pp. 49–74.

[26] E. Khamespanah, M. Sirjani, Z. Sabahi-Kaviani, R. Khosravi, and M. Izadi, “Timed Rebeca Schedulability and Deadlock Freedom Analysis Using Bounded Floating Time Transition System”, *Science of Computer Programming*, vol. 98, pp. 184–204, 2015.

[27] K. G. Larsen, P. Pettersson, and W. Yi, “Diagnostic Model-Checking for Real-Time Systems”, in *Hybrid Systems*, 1995, pp. 575–586.

[28] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, “Verification of an Audio Protocol with Bus Collision Using UPPAAL”, in *Computer Aided Verification, 8th International Conference, CAV ’96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, R. Alur and T. A. Henzinger, Eds., ser. Lecture Notes in Computer Science, vol. 1102, Springer, 1996, pp. 244–256.

[29] M. Lindahl, P. Pettersson, and W. Yi, “Formal Design and Analysis of a Gear Controller”, *STTT*, vol. 3, no. 3, pp. 353–368, 2001.

[30] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, “UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems”, in *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, R. Alur, T. A. Henzinger, and E. D. Sontag, Eds., ser. Lecture Notes in Computer Science, vol. 1066, Springer, 1995, pp. 232–243.

[31] L. Lamport, “Real-Time Model Checking Is Really Simple”, in *CHARME*, D. Borrione and W. J. Paul, Eds., ser. Lecture Notes in Computer Science, vol. 3725, Springer, 2005, pp. 162–175.

[32] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, “Partial Order Reductions for Timed Systems”, in *CONCUR*, D. Sangiorgi and R. de Simone, Eds., ser. Lecture Notes in Computer Science, vol. 1466, Springer, 1998, pp. 485–500.

[33] M. Minea, “Partial Order Reduction for Model Checking of Timed Automata”, in *CONCUR*, J. C. M. Baeten and S. Mauw, Eds., ser. Lecture Notes in Computer Science, vol. 1664, Springer, 1999, pp. 431–446.

[34] J. Håkansson and P. Pettersson, “Partial Order Reduction for Verification of Real-Time Components”, in *FORMATS*, J.-F. Raskin and P. S. Thiagarajan, Eds., ser. Lecture Notes in Computer Science, vol. 4763, Springer, 2007, pp. 211–226.

[35] P. C. Ölveczky, “Semantics, Simulation, and Formal Analysis of Modeling Languages for Embedded Systems in Real-Time Maude”, in *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, G. Agha, O. Danvy, and J. Meseguer, Eds., ser. Lecture Notes in Computer Science, vol. 7000, Springer, 2011, pp. 368–402.

[36] E. Khamespanah, R. Khosravi, and M. Sirjani, “An Efficient TCTL Model Checking Algorithm and a Reduction Technique for Verification of Timed Actor Models”, *Sci. Comput. Program.*, vol. 153, pp. 1–29, 2018.

[37] T. A. Henzinger, Z. Manna, and A. Pnueli, “Timed transition systems”, in *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 600, Springer, 1991, pp. 226–251.

[38] D. Lepri, E. Ábrahám, and P. C. Ölveczky, “Timed CTL Model Checking in Real-Time Maude”, in *WRLA*, 2012, pp. 182–200.

[39] M. Archer, H. Lim, N. A. Lynch, S. Mitra, and S. Umeno, “Specifying and Proving Properties of Timed I/O Automata in the TIOA Toolkit”, in *4th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2006), 27-29 July 2006, Embassy Suites, Napa, California, USA*, IEEE Computer Society, 2006, pp. 129–138.

[40] C. B. Earle and L.-Å. Fredlund, “Verification of Timed Erlang Programs Using McErlang”, in *FMOODS/FORTE*, 2012, pp. 251–267.

[41] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan, “Quantitative Temporal Reasoning”, 4, vol. 4, 1992, pp. 331–352.

[42] S. V. A. Campos, E. M. Clarke, W. R. Marrero, M. Minea, and H. Hiraishi, “Computing Quantitative Characteristics of Finite-State Real-Time Systems”, in *RTSS*, IEEE Computer Society, 1994, pp. 266–270.

[43] F. Laroussinie, P. Schnoebelen, and M. Turuani, “On the Expressivity and Complexity of Quantitative Branching-Time Temporal Logics”, *Theoretical Computer Science*, vol. 297, no. 1-3, pp. 297–315, 2003.

[44] S. V. Campos and E. M. Clarke, “Theories and Experiences for Real-Time System Development”, in, T. Rus and C. Rattray, Eds., 1994, ch. Real-time Symbolic Model Checking for Discrete Time Models, pp. 129–145.

[45] F. Laroussinie, N. Markey, and P. Schnoebelen, “Efficient Timed Model Checking for Discrete-Time Systems”, *Theoretical Computer Science*, vol. 353, no. 1-3, pp. 249–271, 2006.

[46] C. Baier and J.-P. Katoen, *Principles of Model Checking.* MIT Press, 2008, pp. I–XVII, 1–975.

[47] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)* MIT Press, 2009.

[48] A. Sheibanyrad, A. Greiner, and I. M. Panades, “Multisynchronous and Fully Asynchronous NoCs for GALS Architectures”, *IEEE Design & Test of Computers*, vol. 25, no. 6, pp. 572–580, 2008.

[49] Z. Sharifi, M. Mosaffa, S. Mohammadi, and M. Sirjani, “Functional and Performance Analysis of Network-on-Chips Using Actor-Based Modeling and Formal Verification”, *ECEASST*, vol. 66, 2013.

[50] *Apache hadoop home page*, http://hadoop.apache.org.

[51] J. Dean and S. Ghemawat, “Mapreduce: Simplified Data Processing on Large Clusters”, *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[52] T. White, *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated).* O’Reilly, 2012.

[53] L. Linderman, K. Mechitov, and B. F. Spencer, “Tinyos-Based Real-Time Wireless Data Acquisition Framework for Structural Health Monitoring and Control”, *Structural Control and Health Monitoring*, 2012.

[54] L. Nachman, R. Kling, R. Adler, J. Huang, and 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.

[55] E. Khamespanah, K. Mechitov, M. Sirjani, and G. A. Agha, “Schedulability Analysis of Distributed Real-Time Sensor Network Applications Using Actor-Based Model Checking”, in *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, D. Bosnacki and A. Wijs, Eds., ser. Lecture Notes in Computer Science, vol. 9641, Springer, 2016, pp. 165–181.

[56] B. Meyer, *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 1997.

[57] M. Nykänen and E. Ukkonen, “The Exact Path Length Problem”, *J. Algorithms*, vol. 42, no. 1, pp. 41–53, 2002.

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

[59] E. Khamespanah, M. Sirjani, M. Viswanathan, and R. Khosravi, “Floating Time Transition System: More Efficient Analysis of Timed Actors”, in *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, C. Braga and P. C. Ölveczky, Eds., ser. Lecture Notes in Computer Science, vol. 9539, Springer, 2015, pp. 237–255.

[60] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, “A Foundation for Actor Computation”,

[61] C. Sprenger, “A Verified Model Checker for the Modal μ-Calculus in Coq”, in *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS ’98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, B. Steffen, Ed., ser. Lecture Notes in Computer Science, vol. 1384, Springer, 1998, pp. 167–183.

[62] H. Leifsson, “Analyzing Different Scheduling Policies in Natjam Using Timed Rebeca”, Master’s thesis.

[63] Z. Sharifi, “Formal Modeling and Analysis of Network-on-Chip”, Master’s thesis, University of Tehran, School of Electrical and Computer Engineering, Iran, 2013.

[64] M. Sirjani, E. Khamespanah, K. Mechitov, and G. Agha, “A Compositional Approach for Modeling and Timing Analysis of Wireless Sensor and Actuator Networks”, *SIGBED Review*, vol. 14, no. 3, pp. 49–56, 2017.

[65] G. Lipari and G. Buttazzo, “Schedulability Analysis of Periodic and Aperiodic Tasks with Resource Constraints”, *Journal of Systems Architecture*, vol. 46, no. 4, pp. 327–338, 2000.

[66] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System Architecture Directions for Networked Sensors”, *SIGPLAN Notices*, vol. 35, pp. 93–104, 11 Nov. 2000.

[67] A. H. Buss, “Modeling with Event Graphs”, in *Proceedings of the 28th conference on Winter simulation, WSC 1996, Coronado, CA, USA, December 8-11, 1996*, J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, Eds., IEEE Computer Society, 1996, pp. 153–160.

[68] A. El-Hoiydi, "Spatial TDMA and CSMA with Preamble Sampling for Low Power ad-hoc Wireless Sensor Networks", in *Proceedings of the Seventh IEEE Symposium on Computers and Communications (ISCC 2002), 1-4 July 2002, Taormina, Italy*, IEEE Computer Society, 2002, pp. 685–692.

[69] J. Polastre, J. L. Hill, and D. E. Culler, "Versatile Low Power Media Access for Wireless Sensor Networks", in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004, Baltimore, MD, USA, November 3-5, 2004*, J. A. Stankovic, A. Arora, and R. Govindan, Eds., ACM, 2004, pp. 95–107.

[70] Illinois SHM Services Toolsuite, `http://shm.cs.illinois.edu/software.html`.

[71] B. F. Spencer Jr., H. Jo, K. Mechitov, J. Li, S.-H. Sim, R. Kim, S. Cho, L. Linderman, P. Moinzadeh, R. Giles, and G. Agha, "Recent Advances in Wireless Smart Sensors for Multi-Scale Monitoring and Control of Civil Infrastructure", *Journal of Civil Structural Health Monitoring*, pp. 1–25, 2015.

[72] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters", in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, E. A. Brewer and P. Chen, Eds., USENIX Association, 2004, pp. 137–150.

[73] I. T. R. for Semiconductors-ITRS, `http://www.manmaker.com/manual`, 2011.

[74] C. Baier, B. R. Haverkort, H. Hermanns, and J. Katoen, "Performance Evaluation and Model Checking Join Forces", *Commun. ACM*, vol. 53, no. 9, pp. 76–85, 2010.

[75] J. Hu and R. Marculescu, "Dyad: Smart Routing for Networks-on-Chip", in *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, S. Malik, L. Fix, and A. B. Kahng, Eds., ACM, 2004, pp. 260–263.

School of Computer Science
Reykjavík University
Menntavegur 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.ru.is