A Comparative Study of Decision Diagrams for Real-time Model Checking

Omar Al-Bataineh[⋆], Mark Reynolds[†], and David Rosenblum[⋆]

[⋆]National University of Singapore
[†]University of Western Australia

Abstract. The timed automata model, introduced by Alur and Dill, provides a powerful formalism for describing real-time systems. Over the last two decades, several dense-time model checking tools have been developed based on that model. This paper considers the verification of a set of interesting real-time distributed protocols using dense-time model checking technology. More precisely, we model and verify the distributed timed two phase commit protocol, and two well-known benchmarks, the Token-Ring-FDDI protocol, and the CSMA/CD protocol, in three different state-of-the-art real-time model checkers: UPPAAL, RED, and Rabbit. We illustrate the use of these tools using one of the case studies. Finally, several interesting conclusions have been drawn about the performance, usability, and the capability of each tool.

1 Introduction

Real-time systems are systems that are designed to run applications and programs with very precise timing and a high degree of reliability. These systems can be said to be failed if they can not guarantee response within strict time constraints. Ensuring the correctness of real-time systems is a challenging task. This is mainly because the correctness of real-time systems depends on the actual times at which events occur. Hence, real-time systems need to be rigorously modeled and verified in order to have confidence in their correctness with respect to the desired properties.

Because of time constraints in real-time systems, traditional model checking approaches based on finite state automata and temporal logic are not sufficient. Since they can not capture the time requirements of real-time systems upon which the correctness of these systems relies. Several researchers have proposed different modeling formalisms for describing real-time systems such as timed transition systems [21], timed I/O automata [20], and timed automata model [4]. Although a number of formalisms have been proposed, the *timed automata model* of Alur, Courcoubetis, and Dill [4] has become the standard.

In this contribution, we conduct a comparative study of a number of model checking tools, based on a variety of approaches to representing real-time systems. We have selected three real-time protocols, the *timed two phase commit protocol* (T2PC) [16], the *Token-Ring-FDDI protocol* [19], and the *CSMA/CD protocol* [27], implemented them in quite different 'dense' timed model checkers, and verified their relevant properties. Specifically, we consider the model checkers UPPAAL [7], Rabbit [12] and RED [26]. We focus more on the particular T2PC protocol since the protocol has not been model checked before and we use it to illustrate how one can use the three tools to model real-time systems. The tools use different decision diagrams to model and verify real-time systems. UPPAAL deals with the logic of TCTL [2] using an algorithm based on DBMs (Difference Bound Matrices) [17]. Rabbit is a model checker based on timed automata extended with concepts for modular modeling and performs reachability analysis using BDD (Binary Decision Diagrams) [15]. RED is a model checker with dense-time models based on CRD (Clock-Restriction Diagrams) [26]. Comparing model-checking tools is challenging because it requires mastering various modelling formalisms to model the same concepts in different paradigms and intimate knowledge of tools' usage.

We compare the three tools from four different perspectives: (a) their modeling power, (b) their verification and specification capabilities, (c) their theoretical (algorithmic) foundation, and (d) their efficiency and performance. RED outperformed both UPPAAL and Rabbit in two of the case studies (T2PC and FDDI) in terms of scalability, and expressivity of its specification language. On the other hand, Rabbit outperformed both RED and UPPAAL on the CSMA/CD case study. However, UPPAAL was a lot faster than both tools in cases where it gave a result, but it is less scalable than RED.

The CRD-based data structure implemented in RED turns out to be an efficient data structure for handling case studies with huge number of clocks since it scales better with respect to number of clocks. The data structure BDD turns out to be efficient for handling case studies with huge number of discrete variables but it is very sensitive to the scale of clock constants in the model. The DBM-based data structure implemented in UPPAAL handles the complexity of timing constant magnitude very well, but when the number of clocks increases its performance degrades rapidly. It is interesting to mention also that the three tools agreed on the results of all the experiments that we conducted.

Related Work. Some work has already been done on the verification of commitment protocols using formal techniques. In particular, the basic 2PC protocol has frequently been the focus of studies of verification of distributed computing [22, 6, 24], but it is just one of several variants discussed in the literature. One of the interesting variants of the protocol is the T2PC protocol that has complex timing constraints. In this work we have shown how the T2PC protocol can be analyzed with three various tools: UPPAAL, Rabbit, and RED. To the best of our knowledge the T2PC protocol has not been model checked before.

The literature of timed automata theory is a rich literature since it was introduced by Alur and Dill in 1990. In [3] Alur et al. showed that the TCTL is in PSPACE-complexity and gave a model checking algorithm of TCTL. In [18] Henzinger et al. proposed an efficient algorithm for model checking TCTL. Alur and Madhusudan [5] present a full survey of known results for decidability problems in timed automata theory. Ober et al. [23] proposed a timed unified modeling language (UML) for real-time systems and showed how to translate timed UML into timed automata that can be used for formal analysis. Tripakis [25] gives algorithms and techniques to verify timed systems using TCTL logic and Timed Buchi Automata. which have implemented in KRONOS model checking tool. KRONOS is a full DBM-based model checker that supports both forward and backward TCTL model checking. One of the limitations of KRONOS is that its input language supports a very restricted data types that allow only the declaration of clock variables. For this reason we have not included KRONOS in our comparative study since the case study that we consider requires much richer modeling language.

The BDD-like data structures have been also used in the verification of timed systems. The model checkers Rabbit and RED have been developed based on BDD-like technology. Empirical results given in [26] and [12] have shown that RED and Rabbit outperformed UPPAAL in some particular examples such as Fisher mutual exclusion and FDDI Token Ring protocol. However, the empirical results presented in these works were reported using an old version of UPPAAL (v3.2.4), which lack many of the optimisations that are used in the current version of the tool (v4.1.13). In [13] Beyer shows that the size of the BDD and the CRD representation of the reachability set depends on two properties of the models: the number of automata and the magnitude of the clock values. In [26] Wang shows that CRDs outperform DBMs when verifying specifications that contain large number of clocks. However, he pointed out that CRDs consume much space (memory) in handling intermediate data structures, and therefore require intensive use of garbage collection.

2 Preliminaries

In this section, we introduce the basics of the timed two phase commit protocol, which is the main case study considered in this. We then give a brief review of the syntax and semantics of timed automata model, real-time temporal logic, and the zone abstraction which is an abstraction for representing the timing information in the timed models.

2.1 Timed Two Phase Commit Protocol (T2PC)

The T2PC protocol aims to maintain data consistency of all distributed database systems as well as having to satisfy the time constraints of the transaction under processing. The protocol is mainly based on the well-known two phase commit (2PC) protocol, but it incorporates several intermediate deadlines in order to be able to handle real-time transactions. We describe first the basic 2PC protocol (without deadlines) and then discuss how it can be modified to be used for real-time transactions. The 2PC protocol can be summarised as follows [11].

A set of processes $\{p_1, .., p_n\}$ prepare to involve in a distributed transaction. Each process has been given its own subtransaction. One of the processes will act as a coordinator and all other processes are participants. The protocol proceeds into two phases. In the first phase (voting phase), the coordinator broadcasts a start message to all the participants, and then waits to receive vote messages from the participants. The participant will vote to commit the transaction if all its local computations regarding the transaction have been completed successfully; otherwise, it will vote to abort. In the second phase (commit phase), if the coordinator received the votes of all the participants, it decides and broadcasts the decision. If all the votes are 'yes' then the coordinator will commit the transaction. However, if one process voted 'no', then the coordinator will decide to abort the transaction. After sending the decision, the coordinator waits to receive a COMPLETION messages from all the participants.

Three intermediate deadlines have been added to the basic 2PC protocol in order to handle real-time transactions [16]: the deadline V which is a deadline for a participant to send its vote, DEC the deadline for the coordinator to broadcast the decision, and D_p the deadline for a participant to send a COMPLETION message to the coordinator. Note that the correctness of the T2PC protocol depends mainly on the way we select the values of the above timing parameters. In particular, the coordinator should choose the value of D to be sufficiently long to allow the participants to receive the start message and return the completion message in time for the coordinator to determine the result. The correctness of the protocol depends also on a condition that a fair scheduling policy is imposed, this condition is necessary in order to avoid situations in which some participants may miss the deadline if they schedule to execute until after the deadline D. Note also that the protocol can only guarantee correctness in the absence of failures both node failures and link failures, since a failure if happens might delay the execution of some processes until after the deadline expires, which therefore cause the protocol to fail.

2.2 The Timed Automata Model and Real-time Temporal Logic

Timed automata are an extension of the classical finite state automata with clock variables to model timing aspects [4]. Let X be a set of clock variables, then the set $\Phi(X)$ of clock constraints ϕ is defined by the following grammar

$$\phi ::= t \sim c \mid \phi_1 \wedge \phi_2$$

where $t \in X$, $c \in \mathbb{N}$, and $\sim \in \{<, \leq, =, >, \geq\}$. A clock interpretation v for a set X is a mapping from X to $\mathbb{R}^+$ where $\mathbb{R}^+$ denotes the set of nonnegative real numbers.

Definition 1. *A timed automaton A is a tuple $(\Sigma, L, L_0, X, E, \mathcal{L})$, where*

- *Σ is a finite set of actions.*
- *L is a finite set of locations.*
- *L_0 is a finite set of initial locations.*
- *X is a finite set of clocks.*
- *$E \subseteq L \times L \times \Sigma \times 2^X \times \Phi(X)$ is a finite set of transitions. An edge $(l, l^{'}, a, \lambda, \sigma)$ represents a transition from location l to location $l^{'}$ after performing action a. The set $\lambda \subseteq X$ gives the clocks to be reset with this transition, and σ is a clock constraint over X.*

– $\mathcal{L} : L \to 2^{AP}$ *is a labeling function mapping each location to a set of atomic propositions.*

The semantics of a timed automaton $(\Sigma, L, L_0, X, E, \mathcal{L})$ can be defined by associating a transition system with it. With each transition a clock constraint is associated. The transition can be taken only if the clock constraint on the transition is satisfied. There are two basic types of transitions:

1. delay transitions that model the elapse of time while staying at some location,
2. action transitions that execute an edge of the automata.

A state $s = (l, v)$ consists of the current location and the set of clock valuations at that location. The initial state is (l_0, v_0) where the valuation $v_0(x) = 0$ for all $x \in X$. A timed action is a pair (t, a) where $a \in \Sigma$ is an action performed by an automaton $\mathcal{A}$ after $t \in \mathbb{R}^+$ time units since $\mathcal{A}$ has been started.

Definition 2. *An execution of a timed automaton* $\mathcal{A} = (\Sigma, L, L_0, X, E, \mathcal{L})$ *with an initial state* (l_0, v_0) *over a timed trace* $\zeta = (t_1, a_1), (t_2, a_2), (t_3, a_3), ..$ *is a sequence of transitions of the form.*

$$\langle l_0, v_0 \rangle \xrightarrow{d1} \xrightarrow{a1} \langle l_1, v_1 \rangle \xrightarrow{d2} \xrightarrow{a2} \langle l_2, v_2 \rangle \xrightarrow{d3} \xrightarrow{a3} \langle l_3, v_3 \rangle ...$$

satisfying the condition $t_i = t_{i-1} + d_i$ *for all* $i \geq 1$. □

In order to allow the verification of dense-time properties we need to add bounds in the classical CTL temporal operators. The extended logic is called TCTL. We now give the syntax and the semantics of the TCTL logic.

$$\textbf{TCTL} \ni \varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid E\varphi_1 U_I \varphi_2 \mid A\varphi_1 U_I \varphi_2$$

where I is an interval of $\mathbb{R}^+$ that can be either bounded or unbounded. The basic TCTL modality in the above definition is the U-modality which can be used to define the time interval in which the property should be true. Given a formula φ and a state (ℓ, v) of a timed automata $\mathcal{A}$, the satisfaction relation $(\ell, v) \models \varphi$ is defined inductively on the syntax of φ as follows.

– $(\ell, v) \models p$ iff $p \in \mathcal{L}(\ell)$
– $(\ell, v) \models \neg\varphi$ iff $(\ell, v) \not\models \varphi$
– $(\ell, v) \models \varphi \vee \psi$ iff $(\ell, v) \models \varphi$ or $(\ell, v) \models \psi$
– $(\ell, v) \models E\varphi U_I \psi$ iff there is a run ζ in $\mathcal{A}$ from (ℓ, v) such that $\zeta \models \varphi U_I \psi$.
– $(\ell, v) \models A\varphi U_I \psi$ iff for any run ζ in $\mathcal{A}$ from (ℓ, v) such that $\zeta \models \varphi U_I \psi$.
– $(\ell, v) \models \varphi U_I \psi$ iff there exists a position $\pi > 0$ along a run ζ such that $\zeta[\pi] \models \psi$, for every position $0 < \pi^{'} < \pi$, $\zeta[\pi^{'}] \models \varphi$, and duration $(\zeta_{\leq\pi}) \in I$.

2.3 The Zone-based Abstraction Technique

In the original work of Alur and Dill [4], they presented an abstraction technique by which an infinite timed transition system (i.e., timed automata) can be converted into an equivalent finitely symbolic transition system called region graph where reachability is decidable. However, it has been shown that the region automaton is highly inefficient to be used for implementing practical tools. Instead, most real-time model checking tools like UPPAAL, Kronos and RED apply abstractions based on so-called zones, which is much more practical and efficient for model checking real-time systems.

In a zone graph [17], zones are used to denote symbolic states. A zone is a pair (ℓ, Z), where l is a location in the TA model and Z is a clock zone that represents sets of clock valuations at l. Formally a clock zone is a conjunction of inequalities that compare either a clock value or the difference between two clock values to an integer. In order to have a unified form for clock zones we introduce a reference clock x_0 to the set of clocks X in the analyzed model that is always zero. The general form of a clock zone can be described by the following formula

$$(x_0 = 0) \wedge \bigwedge_{0 \leq i \neq j \leq n} ((x_i - x_j) \sim c_{i,j})$$

where $x_i, x_j \in X$, $c_{i,j}$ represents the difference between them, and $\sim \in \{\leq, <\}$. Considering a timed automaton $\mathcal{A} = (\Sigma, L, L_0, X, E, \mathcal{L})$, with a transition $e = (\ell, a, \psi, \lambda, \ell^{'})$ in E we can construct an abstract zone graph $\mathcal{Z}(\mathcal{A})$ such that states of $\mathcal{Z}(\mathcal{A})$ are zones of $\mathcal{A}$. The clock zone $succ(Z, e)$ will denote the set of clock valuations $Z^{'}$ for which the state $(\ell^{'}, Z^{'})$ can be reached from the state (ℓ, Z) by letting time elapse and by executing the transition e. The pair $(\ell^{'}, succ(Z, e))$ will represent the set of successors of (ℓ, Z) under the transition e. Since every constraint used in the invariant of an automaton location or in the guard of a transition is a clock zone, we can use zones for various state reachability analysis algorithms for timed automata.

It is interesting to note that without further abstractions ("extrapolation"), the zone graph can be infinite. To obtain a finite zone graph most model checkers use some kind of extrapolation of zones. In the last two decades, there has been a considerable development in the extrapolation procedure for TA for the purpose of providing coarser abstractions of TA [10, 14, 8]. We refer the reader to [1] for more details about different kinds of extrapolation techniques.

2.4 Data Structures for Representing Zone Graphs

In this section we review briefly the three data structures DBM, BDD, and CRD which have been used respectively to represent clock zones in the tools UPPAAL, Rabbit, and RED.

Difference Bound Matrices (DBMs) [17] are two-dimensional matrices that record the difference upper bounds between clock pairs up to a certain constant. Each row in the matrix represents the bound difference between the value of the clock x_i and all the other clocks in the zone, thus a zone can be represented by at most $|X|^2$ atomic constraints. The element $D_{i,j}$ in DBM is on the form $(n, \sim)$ where $x_i, x_j \in X$, n represents the difference between them, and $\sim \in \{\leq, <\}$. DBM-technology generally handles the complexity of timing constant magnitude very well. But when the number of clocks increases, its performance degrades rapidly. The DBM-based technology has been implemented in the tool UPPAAL.

Binary Decision Diagrams (BDDs) [15] are propositional directed acyclic graphs. The BDD graph consists of a set of decision nodes and has two terminal nodes TRUE-terminal and FALSE-terminal. Each decision node is labeled by a Boolean variable and has two child nodes called low child and high child. A path from the root node to the TRUE-terminal represents a variable assignment for which the represented Boolean function is true. As the path descends to a low child (high child) from a node, then that node's variable is assigned to FALSE (TRUE). For untimed system veification, BDD has shown great success. But for timed system verification, so far, all BDD-like structures have not performed as well as the popular DBM. The BDD data structure is used in the tool Rabbit.

Clock Restriction Diagrams(CRDs) [26] is a BDD-like data structure for representation of sets of zones, with related set-oriented operations for fully symbolic verification of real-time systems. It has similar structure as BDD without FALSE terminal. Unlike BDD, CRD is not a decision diagram for state space membership. It acts like a database for zones and is appropriate for manipulation of sets of clock difference constraints. It has been claimed that CRDs provide more efficient space representation of timed automata than DBMs data structure [26]. The CRD technology is used in the current version of the tool RED. It is worth mentioning here that the CRD data structure of RED is very similar to the CDD data-structure (clock difference diagram) [9].

3 Modeling the Protocol in the Three Tools

In this section we describe how we formalize the T2PC protocol in the tools UPPAAL, RED, and Rabbit. The reason for choosing these tools in the comparison is due to the fact that the tools have been mainly developed for real-time model checking based on TA formalism (or some of its variants). The other reason is that the specification languages of the tools allow one to express common properties of real-time systems in a very natural and easy way (specially the tools UPPAAL and RED). The availability of the user guide of the tools which describes the different options of the tools is another reason for choosing them in this comparison. It is interesting to mention that the protocol has been verified while considering the different available verification options of each tool including the research order, state space representation, and extrapolation technique.

3.1 UPPAAL Model Checker

UPPAAL [7] is a model checker for real-time systems developed in conjunction by Uppsala University, Sweden, and Aalborg University, Denmark. It extends the basic timed automata with features for concurrency, communication, data variables, and priority. UPPAAL uses a client-server architecture which splits the tool into a graphical user interface (client) and a model checking engine (server). The user interface consists of three main sections: system editor, simulator, and verifier. The editor allows the user to model the system as a network of timed automata. The simulator gives the user the capability to interactively run the system to check if there are some trivial errors in the system design. The verifier allows the user to enter the properties to be verified in a sub-language of TCTL. UPPAAL can verify safety, bounded liveness, and reachability properties. UPPAAL uses fragment of TCTL language and it does not support the direct verification of bounded response properties.

The T2PC Protocol in UPPAAL. The coordinator template is depicted in Fig. 1. Initially, the coordinator attempts to reserve a CPU time slot via sending a reservation request signal to the CPU resource manager (see Fig. 3) using the channel `reserve[rsc_id]` indexed with the resource to be allocated. If the CPU is busy in executing other tasks, the manager will add the coordinator process to the waiting queue. Otherwise, it will send immediately the process to the CPU for processing. When the manager receives a `finished` signal from the CPU indicating that the CPU has finished processing the current process and it is currently in an `idle` state, the manager will send the process at the front of the queue (if any) to the CPU for processing. The abstract model of the CPU (see Fig. 4) has two locations `idle` and `InUse` which reflects the status of the CPU. When it receives a `ready[pid]` signal from process `pid`, it moves from `idle` to `InUse`, and then returns from `InUse` to `idle` after the determined execution time is completed. If the resource (CPU) is granted (`rsc_granted ==true`), the coordinator initiates the protocol via broadcasting a `start` message to all the participants. The coordinator then waits to receive the votes of the participants. If V time units passed before receiving all the votes, the coordinator decides to abort and then terminate. Otherwise, it will move to location $m2$ at which it decides and broadcast the decision.

A function `result(part_vote)` returns the result of the transaction based on the values of the received votes. The coordinator broadcasts this result using the broadcast channel `fin_result` and the global variable `outcome`. The coordinator then moves to location $m3$ at which it waits to receive the completion messages of the participants. If D_p time units passed before receiving all completion messages, it decides to abort and then terminate. The protocol ends at location `finished` at which the coordinator updates its database server.

The template of the participants is depicted in Figure 2. All the participants start their execution at location `idle` where they wait to receive a `start` signal from the coordinator. Once they receive that signal, each participant i will try to reserve t_i time units via signalling the resource manager component. If the CPU is busy at that time, it will join the waiting queue until it gets executed. If the deadline V expired before sending their votes to the coordinator they decide to abort and then terminate. Each participant then moves to location $r2$ at which it waits to receive the decision of the coordinator. If it does not receive it within DEC time units, it decides to abort the transaction and terminate. Otherwise, it sets its `comp` variable to true and moves to location $r4$ where it updates its database server and terminates.

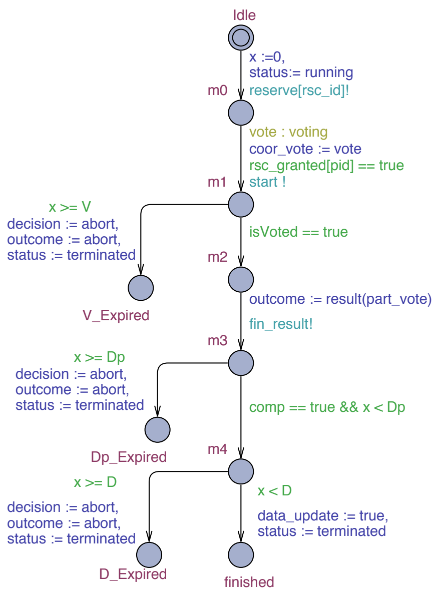

Fig. 1. The coordinator template

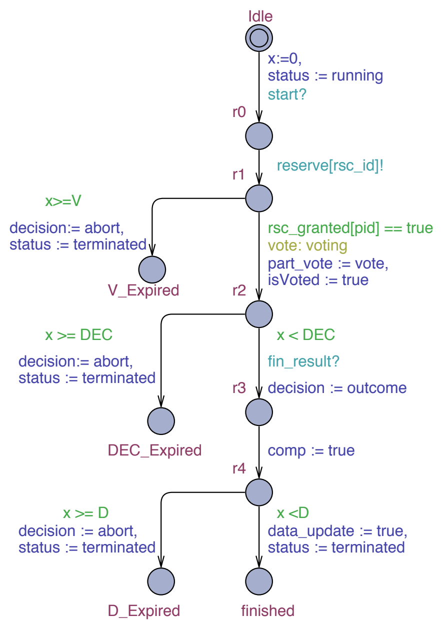

Fig. 2. The participant template

3.2 Rabbit Model Checker

Rabbit [12] is a model checking tool for real-time systems. The theoretical foundation of the tool is mainly based on timed automata extended with concepts for modular modeling. We give an informal description of the formalism of Cottbus Timed Automata (CTA), which is used in the modeling language of Rabbit.

A CTA system consists of a set of modules that can be defined in a hierarchical way. Each module in the system model should have the following components:

– An *identifier*. Identifiers are used to name the modules within the system description. Using identifiers we can create several instances of the modules associated with these identifiers.
– An *Interface*. The interface of a module contains the declarations of the variables that are used in that module. In a CTA module, we can declare clock variables, discrete variables, and synchronisation labels.
  - *Synchronisation labels*. Sometimes called signals which are used to synchronise timed automata that exist in different modules in the system.

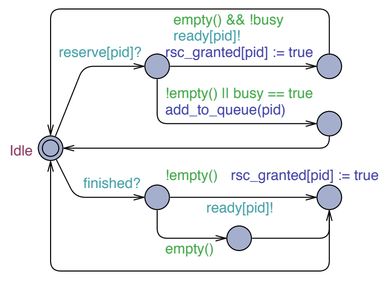

Fig. 3. The resource manager template

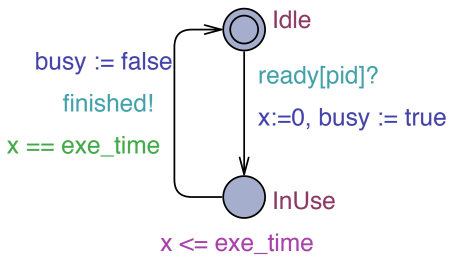

Fig. 4. The CPU template

The concept of synchronisation labels in modules is very similar to the concept of events in CSP.

- *Variables.* Rabbit allows us to declare both continuous (clock) variables and discrete variables. The values of these variables can be updated using assignment statements in the transition rules of the automaton.

– A *timed automaton.* Each module contains a timed automaton. The automaton consists of a finite set of states, a finite set of transitions, and a set of synchronisation labels. In the CTA language, a process transition is declared as a transition rule which starts with the keyword `TRANS`, while the locations of the automaton are declared using the keyword `STATE`.

– *Initial condition.* This is a formula over the module variables and the states of the module's automaton, which specifies the initial state of the module.

– *Instances.* In the CTA model, a module can contain instances of the other defined modules in the model.

Due to space reasons, we do not present the full model of the T2PC protocol in Rabbit here, and we refer the reader to Appendix A of the longer version of the paper at `http://arxiv.org/abs/1201.3416`. However, we pick some statements in the Rabbit model of the T2PC protocol in order to explain how to declare the model behaviour structure with Rabbit. The declaration is a sequence of `STATE` declarations. The statements declare a state whose name is `InUse` and whose invariance condition is "`x<= exe_time`". Inside the transition `TRANS` we have a synchroniser `finished`, a triggering condition "`x == exe_time`", and two actions "`DO busy' = 0;"` and "`GOTO Idle;`".

```
AUTOMATON CPU
{
  STATE InUse{  INV  x<= exe_time;
                TRANS  {GUARD x = exe_time; SYNC ! finished;
                DO busy' = 0; GOTO Idle;} }
  STATE Idle{  TRANS  {SYNC ? ready; DO x' =0 AND
                busy' =1; GOTO InUse;} }
}
```

3.3 RED Model Checker

RED [26] stands for (Region-Encoding Diagrams) is a TCTL model checker for real-time systems. An interesting feature of the RED model checker is that it is totally based on symbolic technology with BDD-like diagrams.

In RED, systems are described as parametrized communicating timed automata, where processes can be model processes, specification processes, or environment processes. In a system with n processes, the user invokes the RED model checker via telling it which processes are for the model, and which for the specification. The remaining processes will be for the environment. Since the automata in RED are parametrised automata then we can declare many process automata with the same automaton template and identify each process automaton with a process index. RED supports both forward and backward analyses, deadlock detection, and counter-example generation. In RED, users can declare global and local variables of type boolean, discrete, clock-restriction variable, and hybrid-restriction variable. Due to space reasons, we don't present here the full model of the T2PC protocol in RED, and we refer the reader to Appendix B of the longer version of the paper at `http://arxiv.org/abs/1201.3416`. The statements declare a mode whose name is `InUse` and whose invariance condition "`x<= exe_time`". Inside the transition rule `when` we have a synchronizer `finished`, a triggering condition "`x == exe_time`", and the two actions "`busy = 0`" and "`goto idle`".

```
mode InUse (x<= exe_time)
{
 when !finished (x == exe_time) may busy = 0; goto Idle;
}
 mode Idle (true)
{
 when ? ready (true) may busy = 1; x =0; goto InUse;
}
```

4 Correctness Conditions of the T2PC Protocol

The first formula of interest is global atomicity (i.e. all processes must agree on the final decision: all must abort or all must commit.)

Specification 1: The global atomicity is always guaranteed.

$$\mathbf{AG}\ (\bigwedge_{i \neq j} \neg(i.\texttt{decision} = \mathit{abort} \wedge j.\texttt{decision} = \mathit{commit}))$$

Note that the variable `decision` can take one of the following values {*undecided*, *abort*, *commit*}. Initially, all agents are *undecided*. Recall that the goal of the protocol is to preserve data consistency as well as to satisfy all designated intermediate deadlines D_p, DEC, and V. If any of these deadlines expired during the execution of the transaction, all processes will decide to abort. Note that the execution of the transaction may be delayed due to queuing delay or due to a communication delay which might cause the protocol to miss its deadlines. The following specifications verify whether the protocol can satisfy these deadlines.

Specification 2: If the coordinator sent successfully a commit request message, then it is guaranteed to receive all participants' votes within V time units.

$$\mathbf{AG}\ ((\texttt{C.request\_sent}) \Rightarrow\ \mathbf{AF}_{\leq V}\ (\bigwedge_{i=1..n}\ (\texttt{C.vote\_rcvd[i]})))$$

Specification 3: If the coordinator received all the votes successfully, then all the participants can receive the decision within DEC time units.

$$\mathbf{AG}\ ((\bigwedge_{i=1..n}(\texttt{C.vote\_rcvd[i]})) \Rightarrow\ \mathbf{AF}_{\leq DEC}\ (\bigwedge_{i=1..n}(i.\texttt{dec\_rcvd})))$$

Specification 4: If the coordinator announced the decision successfully, it can receive acknowledgement signals within D_p time units.

$$\mathbf{AG}\ ((\texttt{C.dec\_sent}) \Rightarrow\ \mathbf{AF}_{\leq D_p}\ (\bigwedge_{i=1..n}\ (\texttt{C.ack[i]})))$$

We discuss now how we specify the properties of the protocol in the input language of each model checker. UPPAAL uses fragment of TCTL logic, RED uses full TCTL logic, while on other hand, TCTL is not available in Rabbit and it uses techniques based on reachability analysis to verify systems properties. Due to space limitation, we consider here only specification 1. For more details about how we specify the whole protocol's properties in each tool we refer the reader to the full version of this paper (`http://arxiv.org/abs/1201.3416`).

In UPPAAL, we can capture specification 1 as follows.

```
A[] not (coor.decision == commit and part.decision == abort)
```

Since Rabbit does not support the TCTL language, it alternatively provides an analysis command language to write a simple segment of code for verifying properties based on reachability analysis. Using this language, we declare a set of variables that are used to represent a set of states, called regions, followed by a set of iterative command statements. We then check whether the model can reach a region where the formula can be violated.

```
REACHABILITY CHECK T2PC {
1   VAR  initial, error, reached : REGION;
2   COMMANDS
3   initial:= INITIALREGION;
4   error := ((coor.decision == 1) AND (part.decision ==2));
5   reached := REACH FROM initial FORWARD;
6   IF (EMPTY(error INTERSECT reached)){
7     PRINT "Specification 1 satisfied.";}
8   ELSE  { PRINT " Specification 1 violated.";}  }
```

The first line declares three regions. Region `initial` represents the set of initial states from the Rabbit's modules. Lines 4 characterizes the set of states that violate specification 1 of the protocol: some process decided to abort while some other process decided to commit. Line 5 assigns to `reached` the set of states reachable from the initial state. The specification is satisfied if the intersection between the `reached` region and the `error` region is empty. However, in RED we can express specification 1 as follows.

```
forall always not (decision[1] == 1 && decision[2] == 2)
```

5 Comparing the Performance of the Three Tools

In this section, we present the model checking runtimes obtained in testing the tools, with version 4.1.13 for UPPAAL, 2.1 for Rabbit, and 5.0 for RED. All experiments are conducted on a PC with 32-bit Redhat Linux 7.3 with Intel (R) core CPU at 2.66 GHz and with 4 GB RAM. The specifications of the T2PC protocol were checked with backward and forward analysis in Rabbit and RED, and using the on the fly approach for UPPAAL. In the tables below we show the CPU time used by the system on behalf of the calling process (system time). An entry of “x” indicates that the model checker ran out of memory on that specification. As shown in section 4 some properties of the T2PC protocol require us to use a full TCTL language and to verify formulas with nested temporal modalities which are not allowed in Rabbit. Moreover, Rabbit does not allow the direct verification of bounded liveness properties of the form $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}_{\leq p}\ \psi)$ which are necessary for the verification of the T2PC protocol. We therefore reduce the bounded liveness properties of the protocol into reachability properties and then add extra monitor automata which interact with the actual model of the protocol in order to capture correctly the required properties. This in fact represents an extra unnecessary overhead and a big disadvantage for the tool Rabbit. In RED we can verify such properties directly. However, UPPAAL supports this special case of nested properties by offering leads-to operator $\rightarrow$ and thus the property $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}_{\leq p}\ \psi)$ can be expressed by: $(\phi \rightarrow (x \wedge c \leq p)\ \psi)$, where x is a clock and c is reset upon ϕ.

Backward analysis					
		Specification			
Number of processes	Model Checker	1	2	3	4
6	Rabbit	1.22	1.21	1.37	1.5
6	RED	10.88	12.9	11.26	9.57
9	Rabbit	x	x	x	x
9	RED	554	249	734	981
12	Rabbit	x	x	x	x
12	RED	2667	6135	6283	4339

Table 1. Model checking runtimes (seconds) for T2PC protocol using Rabbit and RED backward analysis

Forward analysis					
		Specification			
Number of processes	Model Checker	1	2	3	4
6	Rabbit	160	160	161	163
6	RED	2.58	1.19	1.36	1.52
9	Rabbit	x	x	x	x
9	RED	69.7	26.9	29.5	31
12	Rabbit	x	x	x	x
12	RED	3088	884	939	943

Table 2. Model checking runtimes (seconds) for T2PC protocol using Rabbit and RED forward analysis approach

We scaled the model of the protocol until the tools could not verify the protocol properties, due to the state space problem. Note that the T2PC protocol uses a huge number of discrete variables and huge number of clocks which increases as we increase the number of processes in the model. In Table 1 we give the runtimes obtained in checking the protocol using Rabbit backward reachability analysis and RED backward TCTL model-checking. RED could verify successfully the protocol up to 12 processes with 8 clocks, while Rabbit could verify only the simplest cases of the protocol. In Table 2 we report the runtimes obtained in testing the tools Rabbit and RED using forward reachability analysis. Optimizations used in RED make it more scalable than Rabbit by several order of magnitude.

In Table 3 we give the model-checking runtimes of the protocol using UPPAAL's on the-fly approach. UPPAAL could verify successfully the protocol up to 9 processes with 6 clocks. However, UPPAAL was a lot faster in cases where it gave a result, but it is less scalable than RED. As we can see, the DBM-based tool UPPAAL outperforms the CRD-based tool RED when considering small instances of the protocol with small number of clocks. However, when considering instances involving larger numbers of processes and larger numbers of clocks we find that RED outperforms UPPAAL where we could analyze the protocol up to 12 processes in RED while we fail to do so in UPPAAL.

On The Fly Approach				
	Specification			
Number of processes	1	2	3	4
6	0.01	0.001	0.002	0.003
9	4.4	5.5	10.3	8.84
12	x	x	x	x

Table 3. Model checking runtimes (seconds) for T2PC protocol using UPPAAL on the fly approach

In table 4 we summarize information about the time taken to do the modeling and verification in each tool, the number of code line, the number of automata used to model and verify the basic case of the T2PC protocol, and the available verification options in each tool. Note that the time spent to learn the language of UPPAAL and then to verify the protocol is significantly shorter than the time spent to learn and model the protocol in both RED and Rabbit since UPPAAL is a very user-friendly tool. It is interesting to mention that the experience level of the authors about the three tools before conducting the experiments was initially the same. It is worth mentioning also that in Rabbit we use 9 automata: 6 automata to model the processes of the protocol and 3 extra monitor automata to capture bounded liveness properties (see specifications 2-4 in Section 4). On the other hand, we use only 6 automata to model and verify the protocol in RED and UPPAAL since they allow us to verify directly bounded liveness properties. In interesting to mention that in addition to the GUI automata used in the UPPAAL's model, we use also some extra simple functions as shown in Figures 1, 2, and 3, namely `add_queue(pid)`, `rsc_granted(pid)`, and `result(part_vote)`. The implementationand of these functions are very straightforward which requires only a few lines of code (about 35 lines).

Tool	Time Spent	# of Code Line	# of Automata	Verification Options
UPPAAL	$\approx$ 18 hours	GUI automata plus 35 lines	6	Breadth, Depth, random On-the-fly
RED	$\approx$ 45 hours	85	6	Backward/Forward TCTL
Rabbit	$\approx$ 52 hours	110	9	Backward/Forward Reachability

Table 4. Modeling Time and Effort for the T2PC protocol in the Three tools

Now we turn to discuss the model checking runtimes obtained in testing the three tools on the following two benchmarks. The models of the benchmarks have been taken from the distributed installation package of each tool.

Token-Ring-FDDI Protocol Fiber Distributed Data Interface (FDDI) [19] is a high speed protocol for local networks based on token ring technology. We use a simplified model of N-processes. One process models the ring, that hands the token in one direction to $N-1$ symmetric processes, that may hand back the token in a synchronous (high-speed) fashion. The ring process owns a local clock and every station owns three local clocks. This case study uses a huge number of clocks and a huge number of synchronisation labels. Here again RED outperformed both UPPAAL and Rabbit since RED is the only tool that succeeded to verify the protocol up to 16 senders. In fact the number of reachable locations in the RED model does not explode with growing number of senders. This proves again that the CRD-technology scales better with respect to number of clocks.

CSMA/CD Protocol Carrier Sense Multiple Access with Collision Detection (CSMS/CD) [27] is a protocol for communication on a broadcast network with a multiple access medium. This case study uses a huge number of synchronisation labels and discrete variables and small number of clocks. For this case study, Rabbit outperformed both RED and UPPAAL since the BDD-based tool

no. of Senders	2	4	6	8	10	12	14	16
UPPAAL	0.01	0.03	0.16	1.42	18.2	280	4535	x
RED	0.02	0.09	0.26	0.61	1.18	3.8	3.6	8.9
Rabbit	0.04	0.25	0.99	4.20	11.7	29.9	x	x

Table 5. Time for the computation of the reachability set of FDDI protocol

Rabbit handles case studies with huge discrete variable much better than the CRD-based tool RED and the DBM-based tool UPPAAL.

no. of processes	2	4	6	8	10	12	14	16	32
UPPAAL	0.01	0.04	7.1	9.5	x	x	x	x	x
RED	0.05	0.27	1.25	7.88	51.2	518	x	x	x
Rabbit	0.02	0.08	0.25	0.79	1.5	2.8	14.6	65.8	3260

Table 6. Time for the computation of the reachability set of CSMA/CD protocol

Several conclusions can be drawn from the above reported results. RED is able to verify properties that are not expressible in UPPAAL and Rabbit and it supports full TCTL language with fairness assumptions. RED also allows verifying bounded liveness formulas that contain nested temporal modalities. On the other hand, UPPAAL's specification language supports fragment of TCTL and Rabbit specification language is restricted to reachability formulas. We believe that this limitation of the specification language of Rabbit is something that can lift the usability of the tool in particular when considering systems with timing constraints of the form $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}_{\leq p}\ \psi)$.

Unlike UPPAAL, RED and Rabbit provide no graphical interface or simulation facilities. Moreover, UPPAAL allows a very natural formalization of systems this is not, or less, possible in Rabbit or RED. In case the specification fails, UPPAAL provides a counterexample and allows one to trace (simulate) the counterexample state by state in a very intuitive way. RED also provides this facility (it generates a counterexample when a specification fails) but in a less intuitive way than UPPAAL. The CRD-based data structure implemented in RED turns out to be an efficient data structure for handling case studies with huge number of clocks since it scales better with respect to number of clocks. The data structure BDD turns out to be efficient for handling case studies with huge number of discrete variables but it is very sensitive to the scale of clock constants in the model. While the DBM-based data structure implemented in UPPAAL handles the complexity of timing constant magnitude very well, its performance degrades rapidly when the number of clocks increases.

6 Conclusion

We have verified three timed distributed protocols (T2PC, FDDI, and CSMA/CD) in the model checkers UPPAAL, Rabbit, and RED. The three model checkers vary in how easy, or difficult, it is to formalise the protocol and its properties in the language of each model checker. In summary, to model and verify real-time systems that have complex timing requirements, we recommend using the tool RED as it supports a full TCTL language which allows to express a wide variety of timed properties. For timed systems with complex modeling details, we recommend using the tool UPPAAL as it has richer expressiveness in modeling systems than Rabbit and RED. Since Rabbit supports modular modelling that allows one to represent systems components in a hierarchical way, we recommend using it when the system has components with different levels of hierarchy.

References

1. Omar I. Al-Bataineh, Mark Reynolds, and Tim French. Finding minimum and maximum termination time of timed automata models with cyclic behaviour. *Theoretical Computer Science*, 665:87–104, 2017.
2. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. In *Information and Computation*, pages 2–34. 1993.
3. R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *Proceeding of the 5th Annual Sympoisum on Logic in Computer Science*, pages 414–425, 1990.
4. R. Alur and D. Dill. A theory of timed automata. In *TCS*, pages 183–235. 1994.
5. R. Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *International School on Formal Methods for the design of Computer, Communication and Software Systems, SFM-RT 2004*, pages 200–236. 2004.
6. M. Atif. Analysis and verification of two-phase commit and three-phase commit protocols. In *Emerging Technologies ICET'09*, pages 326–331, 2009.
7. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-time Systems (SFM-RT 2004)*, pages 200–236. Springer, 2004.
8. Gerd Behrmann, Patricia Bouyer, Kim G. Larsen, and Pelnek Radek. Lower and upper bounds in zone-based abstractions of timed automata. *International Journal on Software Tools for Technology Transfer*, pages 204–215, 2006.
9. Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *CAV'99*, pages 341–353. Springer-Verlag, 1999.
10. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets.* Springer–Verlag, 2004.
11. A. Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. Addison-Wesley, 1987.
12. D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A tool for BDD-based verification of realtime systems. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, pages 122–125, 2003.
13. D. Beyer and A. Noack. Can decision diagrams overcome state space explosion in real-time verification? In *Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003)*, pages 193–208, 2003.
14. Patricia Bouyer. Forward analysis of updatable timed automata. *Formaml Methods of System Design*, 24:281–320, 2004.

15. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, pages 677–691, 1986.
16. S. Davidson, I. Lee, and V. Wolfe. A protocol for times atomic commitment. In *Proceeding of 9th International Conference On Distributed Computing System*, 1989.
17. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212. Springer-Verlag New York, Inc., 1990.
18. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992.
19. Raj Jain. *FDDI Handbook: High-Speed Networking Using Fiber and Other Media.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.
20. D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed I/O Automata: a mathematical framework for modelling and analyzing real-time systems. In *Proceedings 24th IEEE International Real-Time Systems Symposium (RTSS03)*, pages 166–177, 2003.
21. K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 14–24, 1997.
22. Jeff Magee. Analyzing synchronous distributed algorithms. 2003.
23. Iulian Ober, Susanne Graf, and Ileana Ober. Validation of uml models via a mapping to communicating extended timed automata. In *Proc. 11th Int. SPIN Workshop on Model Checking of Software, LNCS 2989, Springer-Verla*, pages 127–145. Springer-Verlag, 2004.
24. Ölveczky Peter Csaba. Formal Modeling and Analysis of a Distributed Database Protocol in Maude. In *Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering - Workshops*, pages 37–44, 2008.
25. S. Tripakis. *The analysis of timed systems in practice.* PhD thesis, Universite Joseph Fourier, Grenoble, France, 1998.
26. Farn Wang. Symbolic verification of complex real-time systems with clock-restriction diagram. In *Proceedings of the IFIP TC6/WG6.1*, pages 235–250. Kluwer, B.V., 2001.
27. Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997.