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

Omar Al-Bataineh and Mark Reynolds

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. The 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 [13], timed I/O automata [12], and timed automata model [2]. Although a number of formalisms have been proposed, the *timed automata model* of Alur, Courcoubetis, and Dill [2] 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) [9], the *Token-Ring-FDDI protocol* [11], and the *CSMA/CD protocol* [17], implemented them in quite different 'dense' timed model checkers,

and verified their relevant properties. Specifically, we consider the model checkers UPPAAL [4], Rabbit [7] and RED [16]. 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 [1] using an algorithm based on DBMs (Difference Bound Matrices) [10]. Rabbit is a model checker based on timed automata extended with concepts for modular modeling and performs reachability analysis using BDD (Binary Decision Diagrams) [8]. RED is a model checker with dense-time models based on CRD (Clock-Restriction Diagrams) [16]. 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. 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 w.r.t 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, but when the number of clocks increases its performance degrades rapidly. Finally, it is interesting to mention that the three tools agreed on the results of all the experiments that we conducted.

2 Preliminaries

2.1 The 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 [6].

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 decide to commit the result of the transaction. However, if one vote said '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 be able to handle real-time transactions [9]: V the deadline for a participant to vote, DEC the deadline for sending a decision by the coordinator, and D_p the deadline for sending a completion message by a participant to the coordinator. We refer to [9] for more details about how to compute these deadlines.

The basic 2PC protocol (without deadlines and timers) has been extensively studied and analysed using model checking [14, 3, 15]. However, no work has been done on model checking the real-time version of the protocol (T2PC). Of course, analysing real-time commit protocols is much harder than analysing conventional commit protocols since real-time protocols usually involve many timers in their design which increase the algorithmic complexity of the analysis.

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 [2]. 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 $\sim \in \{<, \leqslant, =, >, \geqslant\}$. 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 (Σ, L, L_0, X, E), 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.* □

The semantics of a timed automaton (Σ, L, L_0, X, E) is defined by associating a transition systems 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 $\mathcal{A} = (\Sigma, L, L_0, X, E)$ 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 $i \geqslant 1$. □

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.

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

where I is an interval of $\mathbb{R}^+$ that can be bounded, unbounded, singular, non-singular, closed or open, or semi closed. The basic TCTL modality in the above definition is the *until* modality 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 $(l, v) \models \varphi$ is defined inductively on the syntax of φ as follows.

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

2.3 The Zone-based Abstraction Technique

In the original work of Alur and Dill [2], they proposed 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 [10], 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 analysed model that is always zero. The general form of a clock zone can be described by the following formula.

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

where $x_i, x_j \in X$, $c_{i,j}$ represents the difference between them, and $\sim \in \{\leqslant, <\}$. Considering a timed automaton $\mathcal{A} = (\Sigma, L, L_0, X, E)$, with a transition $e = (l, a, \psi, \lambda, l^{'})$ 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 $(l^{'}, Z^{'})$ can be reached from the state (l, Z) by letting time elapse and by executing the transition e. The pair $(l^{'}, succ(Z, e))$ 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.

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 Difference Bound Matrices [10] 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 $\sim \in \{\leqslant, <\}$. 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 tools UPPAAL and Kronos.

Binary Decision Diagrams Binary Decision Diagrams (BDDs) [8] 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 Clock Restriction Diagrams (CRDs) [16] 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 [16]. 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) [5].

3 Modeling The Protocol in The Three Tools

3.1 UPPAAL Model Checker

UPPAAL [4] 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.

The T2PC Protocol in UPPAAL The coordinator template is depicted in Figure 1. Initially, the coordinator attempts to reserve a CPU time slot via sending a reservation request signal to the CPU resource manager (see Figure 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 Figure 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 successfully 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.

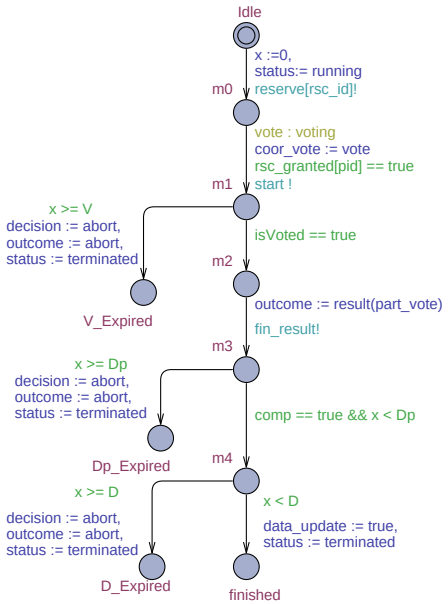

Fig. 1. The coordinator template

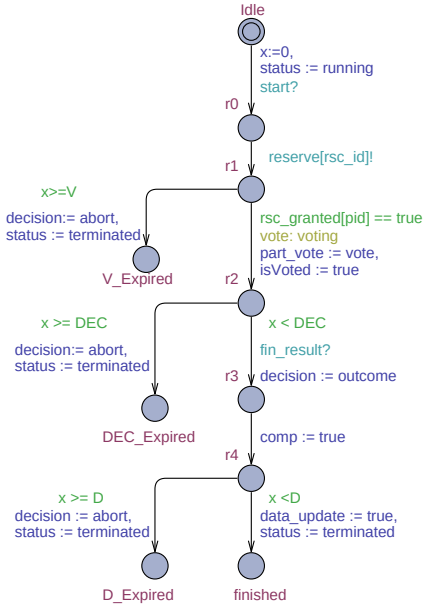

Fig. 2. The participant template

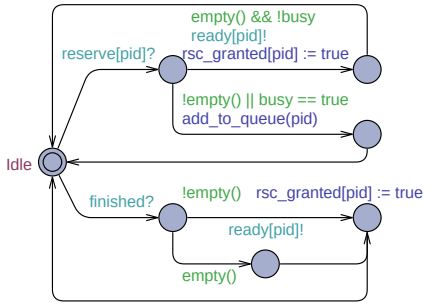

Fig. 3. The resource manager template

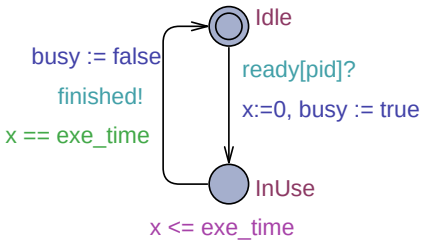

Fig. 4. The abstract CPU template

3.2 Rabbit Model Checker

Rabbit [7] 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. 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.
- *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 CTA model, a module can contain instances of of the other defined modules in the model. This is useful when we model systems containing subsystems, and that these subsystems occur several times in a system.

The textual description of the protocol in Rabbit's language is given in Appendix A.

3.3 RED Model Checker

RED [16] 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. The textual description of the protocol in RED is given in Appendix B.

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 three specifications verify whether the protocol can satisfy these deadlines.

Specification 2: If there are no faults and the coordinator C sent successfully a commit request message, then it is guaranteed to receive all participants' votes within V time units.

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

Specification 3: If there are no faults and the coordinator received all the votes successfully, then all the participants can receive the decision within DEC time units.

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

Specification 4: If there are no faults and that the coordinator sent successfully the decision, then it can receive all acknowledgement signals within D_p time units.

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

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 (see appendix A). 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 $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}_{\leqslant p}\ \psi)$ 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 $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}_{\leqslant p}\ \psi)$ can be expressed by: $(\phi \rightarrow (x \wedge c \leqslant p)\ \psi)$, 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 Rabbit and RED With 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 Rabbit and RED With Forward Analysis

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 obtainted 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. 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 analyse 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

Table 3. Model Checking Runtimes (seconds) for UPPAAL With On the Fly Approach

In table 4 we summarise 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.

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.

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

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

Token-Ring-FDDI Protocol Fiber Distributed Data Interface (FDDI) [11] 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 w.r.t number of clocks.

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

CSMA/CD Protocol Carrier Sense Multiple Access with Collision Detection (CSMS/CD) [17] 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 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 $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}_{\leqslant p}\ \psi)$. 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. 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 w.r.t 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, but when the number of clocks increases its performance degrades rapidly.

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 RED tool since 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 UPPAAL tool since it has richer expressiveness in modeling systems than Rabbit and RED. Since Rabbit supports modular modelling that allows us to represent systems components in a hierarchical way we recommend using it when we have a system where its components have different levels of hierarchy.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. In *Information and Computation*, pages 2–34. 1993.
2. R. Alur and D. Dill. A theory of timed automata. In *TCS*, pages 183–235. 1994.
3. M. Atif. Analysis and verification of two-phase commit and three-phase commit protocols. In *Emerging Technologies ICET'09*, pages 326–331, 2009.
4. 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.
5. 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.
6. A. Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. Addison-Wesley, 1987.

7. 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.
8. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, pages 677–691, 1986.
9. S. Davidson, I. Lee, and V. Wolfe. A protocol for times atomic commitment. In *Proceeding of 9th International Conference On Distributed Computing System*, 1989.
10. 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.
11. Raj Jain. *FDDI Handbook: High-Speed Networking Using Fiber and Other Media*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.
12. 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.
13. 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.
14. Jeff Magee. Analyzing synchronous distributed algorithms. 2003.
15. 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.
16. 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.
17. Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997.

A The Textual Description of The T2PC Protocol in Rabbit

```
MODULE Coordinator
{
   LOCAL
      x : CLOCK; vote : DISCRETE(3) ; update : DISCRETE;
      decision : DISCRETE(3) ; status : DISCRETE;
   INPUT
       D : CONST;    //the deadline of the transaction.
       V : CONST;   // the deadline for receiving participants' votes.
       Dp: CONST;  //  the deadline for receiving participants' completion messages.
       commit :   SYNC;  // to commit the transaction.
       abort  :   SYNC; // to abort the transaction.
       comp   :   SYNC; // to receive the participant's completion message
  OUTPUT
      reserve   : SYNC;   // to reserve a CPU time slot.
      start     : SYNC;  //  to start the T2PC protocol.
  MULTREST
     outcome : DISCRETE;
     resource_granted : DISCRETE;
     dec_sent : DISCRETE;
  // Set initial state of automaton and clock values.
INITIAL
   STATE(Coor) = init AND x = 0 AND status =0;
AUTOMATON Coor
{
   STATE init{ TRANS{ SYNC ! reserve;   GOTO begin;}
                         TRANS {GUARD x >= V;     GOTO fail;} }
   STATE begin{ TRANS{ GUARD resource_granted = 1;
                       SYNC ! start; DO vote' =1; GOTO waitVotes;}
                TRANS  {GUARD resource_granted = 1;
                        SYNC ! start; DO vote' =2; GOTO waitVotes;}}
   STATE waitVotes  {TRANS{  SYNC ?abort; DO decision' =1 AND dec_sent' =1
                             AND outcome' =1;  GOTO waitCompMsg;}
                     TRANS  {GUARD vote =2; SYNC ?commit; DO decision' =2 AND
                             dec_sent' =1 AND outcome' =2;
                             GOTO waitCompMsg;}
                     TRANS  {GUARD vote =1; SYNC ?commit; DO decision' =1 AND
                             dec_sent' =1 AND outcome' =1;
                             GOTO waitCompMsg;}
                     TRANS  {GUARD x >= V; GOTO fail;}}
  STATE waitCompMsg{  TRANS {SYNC ?comp; GOTO finish; }
                      TRANS {GUARD x >= Dp; GOTO fail;}}

  STATE finish{  TRANS  {GUARD x <=D; DO update' = 1 AND status' =1;
```

```
                        GOTO exception;}
                TRANS  {GUARD x> D; GOTO exception;} }
  STATE fail{   TRANS  {DO decision' = 1 AND outcome' =1
                        AND status' =1; GOTO exception;}}
  STATE exception {}
}
}
// The following module represents the template of the participant.
MODULE Participant
{
 LOCAL
     x : CLOCK; vote : DISCRETE (3);
     decision : DISCRETE(3) ; status : DISCRETE(2); update : DISCRETE;
 INPUT
     D : CONST;  V : CONST ;  DEC : CONST;  start : SYNC;
 OUTPUT
      reserve :  SYNC;  commit :  SYNC;   abort :  SYNC;  comp :  SYNC;
 MULTREST
     outcome : DISCRETE;
     resource_granted : DISCRETE;
     dec_sent : DISCRETE;
 // Set initial state of automaton and clock values.
 INITIAL
     STATE(Part) = init AND x = 0 AND status =0;
AUTOMATON Part
{
  STATE init{   TRANS {SYNC? start;  GOTO reserveTimeSlot;}
                TRANS  {GUARD x >= V; GOTO fail;} }
  STATE reserveTimeSlot{  TRANS  {SYNC ! reserve; GOTO sendVote;}
                          TRANS  {GUARD x >= V;   GOTO fail;} }
  STATE sendVote{  TRANS {GUARD resource_granted = 1 AND x <V;
                          SYNC !abort; DO vote' = 1;  GOTO waitDec;}
                   TRANS {GUARD resource_granted = 1 AND x <V;
                          SYNC !commit; DO vote' = 2; GOTO waitDec;}
                   TRANS  {GUARD x >= V; GOTO fail;}}

  STATE waitDec{   TRANS  {GUARD x >= DEC; GOTO fail;}
                   TRANS  {GUARD x < DEC AND dec_sent=1; DO decision' = outcome;
                           GOTO sendCompMsg;} }
  STATE sendCompMsg{  TRANS  {SYNC ! comp; GOTO finish;} }
  STATE fail{         TRANS  {DO decision' = 1 AND status' =1;  GOTO exception;} }
  STATE finish{       TRANS  {GUARD x <D; DO update' = 1 AND status' =1;
                              GOTO exception;}
                      TRANS  {GUARD x>= D; GOTO exception;} }
  STATE exception {}
```

```
}
}
// The following module represents an abstract model of the CPU.
MODULE resource
{
LOCAL
  x : CLOCK;
INPUT
  ready : SYNC;
  exe_time : CONST;
OUTPUT
  finished : SYNC;
MULTREST
  busy : DISCRETE; // 0 means the CPU is free, and 1 means it is busy.
INITIAL
   STATE (CPU) = Idle AND x =0 AND busy =0;
AUTOMATON CPU
{
  STATE Idle{  TRANS  {SYNC ? ready; DO x' =0 AND
                       busy' =1; GOTO InUse;} }
  STATE InUse{  INV  x<= exe_time;
              TRANS  {GUARD x = exe_time; SYNC ! finished;
              DO busy' = 0; GOTO Idle;} }
}
}
// The template of the CPU manager.
MODULE ResouceManager
{
LOCAL
   wait : DISCRETE;
INPUT
   reserve : SYNC; finished : SYNC;
OUTPUT
  ready : SYNC;
MULTREST
  busy : DISCRETE; // 0 means the CPU is free, and 1 means it is busy.
  resource_granted : DISCRETE;
INITIAL
   STATE (Manager) = Idle;
AUTOMATON Manager
{
  STATE Idle { TRANS  {SYNC ? reserve; GOTO M1;}
              TRANS  {SYNC ? finished; GOTO M2;} }
  STATE M1{  TRANS {GUARD busy =0;
                    SYNC !ready; DO resource_granted' = 1; GOTO Idle;}
```

```
            TRANS {GUARD busy = 1;
                   DO wait' =1; GOTO Idle;} }
  STATE M2{ TRANS  {GUARD wait =1;  SYNC !ready; DO wait' =0
                    AND resource_granted' = 1; GOTO Idle;}
            TRANS {GUARD wait = 0;  GOTO Idle;}}
}
}
```

We pick some statements in the Rabbit model in order to explain how to declare the model behaviour structure with Rabbit. The declaration is a sequence of `STATE` declarations. A typical `STATE` declaration can be found from the statements (S1) to (S3). 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;`". Another statement that we believe it needs to be explained is the declaration statement `decision: DISCRETE(3)`. The parameter in the statement tells Rabbit how many values we want to store in the variable `decision`. The statement indicates that the `decision` variable can take one of the following values $\{0, 1, 2\}$. This gives Rabbit the opportunity to spend the correct number of bits in the BDD representation. Note that synchronization between automata is done via synchronization labels and the semantic is as in CSP, i.e., one automaton can take a transition with label S, if all other automata also take a transition with S.

B The Textual Description of The T2PC Protocol in RED

```
#define D 80
#define D_p 52
#define DEC 40
#define V 15
#define exe_time 52
process count = 6;
/* One local clock. */
local clock x;
/* A set of synchronizers. */
global synchronizer start, reserve1, reserve2, yes, no, commit, abort,
                    comp, ready1, ready2, finished1, finished2;
local discrete vote: 1..2;
local discrete decision: 0..2;
local discrete update: 0..1;
local discrete status: 0..1;
global discrete busy1: 0..1;
global discrete busy2: 0..1;
global discrete outcome: 1..2;
```

```
global discrete resource_granted1: 0..1;
global discrete resource_granted2: 0..1;
global discrete wait1: 0..1;
global discrete wait2: 0..1;
/* 7 modes for the coordinator. */
 mode coor_idle (true)
 {
   when ! reserve1 (true) may goto coor_begin;
 }
  mode coor_begin (true)
 {
   when !start (resource_granted1 ==1)  may x= 0; vote =1; goto wait_votes;
   when !start (resource_granted1 ==1)  may x= 0; vote =2; goto wait_votes;
 }
 mode wait_votes (x<=D)
 {
   when ?yes (vote ==2 && x < V)  may decision =2; outcome =2; goto sendDecision;
   when ?yes (vote ==1 && x < V)  may decision =1; outcome =1; goto sendDecision;
   when ?no  (x < V) may decision = 1; outcome =1; goto sendDecision;
   when (x >= V) may goto coor_fail;
 }
 mode sendDecision (x <=D)
 {
   when !commit (decision == 2) may goto waitCompMsg;
   when !abort  (decision == 1) may goto waitCompMsg;
 }
 mode waitCompMsg (x<=D)
 {
   when ?comp (x < D_p) may goto coor_final;
   when (x>D_p) may goto coor_fail;
 }
 mode coor_final (true)
 {
   when (x < D) may update =1; status =1;
   when (x> D) may goto coor_fail;
 }
 mode coor_fail (true)
 {
   when (true) may decision =1; outcome =1; status =1;
 }
/* The behaviour of the participant can be described as follows.*/
 mode part_idle (true)
 {
   when ?start (true) may x = 0; goto part_reserve;
 }
```

```
 mode part_reserve(true)
 {
    when ! reserve2 (true) may goto part_start;
 }
 mode part_start (x < D)
 {
   when !yes (resource_granted2 ==1 && x < V) may vote =2; goto part_wait;
   when !no  (resource_granted2 ==1 && x < V) may  vote =1; goto part_wait;
   when (x >= V) may goto part_fail;
 }
 mode part_wait (true)
 {
   when ?abort (x < DEC) may decision =1; goto sendCompMsg;
   when ?commit (x < DEC) may decision =2; goto sendCompMsg;
   when (x>= DEC) may goto part_fail;
 }
 mode sendCompMsg (true)
 {
   when !comp(true) may goto part_final;
 }
  mode part_final (true)
 {
   when (x < D) may update =1; status =1;
   when (x> D) may goto part_fail;
 }
 mode part_fail (true)
 {
   when (true) may decision =1; status =1;
 }

/* The template of the CPU1 */
 mode CPU1_idle (true)
 {
   when ? ready1 (true) may busy1 = 1; x =0; goto CPU1_InUse;
 }
 mode CPU1_InUse (x<= exe_time)
 {
   when !finished1 (x == exe_time) may busy1 = 0; goto CPU1_idle;
 }
 mode CPU2_idle (true)
 {
   when ? ready2 (true) may busy2 = 1; x =0; goto CPU2_InUse;
 }
 mode CPU2_InUse (x<= exe_time)
```

```
{
  when !finished2 (x == exe_time) may busy2 = 0; goto CPU2_idle;
}
/* The template of the CPU1 manager */
mode Manager1_idle (true)
{
  when ?reserve1 (true) may goto M11;
  when ?finished1 (true) may goto M21;
}
mode M11 (true)
{
  when !ready1 (busy1 ==0) may resource_granted1 =1; goto Manager1_idle;
  when (busy1 ==1) may wait1 =1; goto Manager1_idle;
}
mode M21 (true)
{
  when !ready1 (wait1 ==1) may wait1 =0; resource_granted1 =1; goto Manager1_idle;
  when  (wait1 ==0)  may goto Manager1_idle;
}
/* The template of the second CPU2 manager */
mode Manager2_idle (true)
{
  when ?reserve2 (true) may goto M12;
  when ?finished2 (true) may goto M22;
}
mode M12 (true)
{
  when !ready2 (busy2 ==0) may resource_granted2 =1; goto Manager2_idle;
  when (busy2 ==2) may wait2 =1; goto Manager2_idle;
}
mode M22 (true)
{
  when !ready2 (wait2 ==1) may wait2 =0; resource_granted2 =1; goto Manager2_idle;
  when  (wait2 ==0)  may goto Manager2_idle;
}
```

We also pick some statements in the RED model in order to explain how to declare the model behaviour structure with RED. The declaration is a sequence of `mode` declarations. A typical `mode` declaration can be found from the statements (S1) to (S2). The statements declare a mode whose name is `InUse` and whose invariance condition "`x<= exe_time`". Inside the transition rule `when` we have a synchroniser `finish`, a triggering condition "`x == exe_time`", and the two actions "`busy = 0`" and "`goto idle`". The initial condition of the protocol declares six processes. The first process represents the coordinator database server which starts at `coor_idle` location, the second process represents the participant server which starts at the `part_idle` location, the third and the

fourth processes represent the CPU at each server, and finally processes 5 and 6 represent the resource manager on each database server.