Formal Modeling and Analysis of a Distributed Transaction Protocol in UPPAAL

Omar Al-Bataineh Tim French Terry Wooding

Abstract

We present a formal analysis of the well-known two phase atomic commitment protocol. The protocol is modeled as networks of timed automata using the model checker UPPAAL. The protocol has been verified in two different crash models, the crash-stop model, and the crash-recovery model. The analysis allows us to discover several interesting conclusions about the protocol. The paper also describes how dense-timed model checking technology may be applied to discover the worst-case execution time and the corresponding worst-case scenario of the protocol. The analysis also allows us to illustrate various features of the UPPAAL tool, which shows that the specification language of the tool lacks the expressiveness to capture some desired properties of the protocol.

1. Introduction

Distributed systems have proven to be hard to understand, design, and reason about due to their complexity and non-deterministic nature. They usually involve subtle interactions of a number of components that have high level of parallelism. This is why the correctness of these systems is difficult to ensure. Several systems and protocols have been proven not to succeed in satisfying their intended goals after they have been published [19, 15]. One of the promising solutions to this problem is the use of formal verification techniques such as model checking technique [13].

Model checking is one of the well-known formal verification techniques that is used in checking critical systems such as hardware designs, communication protocols, and distributed systems. Model checking is an automated method for verifying finite state systems. It uses a variety of sophisticated heuristics and symbolic implementation techniques to check that a logical formula holds in the given system design. Typically, formulas are expressed using logics such as temporal logic, belief logic, or epistemic logic, while the system will be expressed as a set of states (finite number of states) and a set of transitions. When the system fails to meet a desired property, the model checker produces a counterexample that helps to identify the source of the error in the system design.

Typically modelers use untimed models (i.e. models that do not capture timing details in the systems) to describe distributed systems. This has several advantages. First, untimed models are simpler to represent, describe, and to reason about. Second, ignoring the timing details in these protocols helps to reduce significantly the size of the state space of the system under consideration, since in real-time systems the state space grows exponentially not only with the number of the concurrent components, but also with the number of clocks in the system under consideration. There are many interesting systems and protocols where timing assumptions can be essential to their correctness, and therefore can not be accurately described using untimed models. This applies to a large class of synchronous distributed systems. Examples include, sliding windows protocol for reliable transmission of data over unreliable channels [10], a protocol to monitor the presence of network nodes [8], atomic commitment protocols [7, 16], the ZeroConfig protocol [9, 22], and the agreement algorithm described in [5]. The correct behaviour of these protocols typically depends on determining the minimum length of timeout intervals; if transmission delays are not properly taken into account, data may be lost if timeouts occur too soon. Modeling such protocols are in general non-trivial and their correctness is not obvious. In order to specify and model check this class of protocols finite state automata and linear temporal logic are not sufficient. They especially lack the ability to trace and analyse real-time system properties. However, the timed automata framework is an appropriate choice for describing and modeling this class of protocols.

Timed automata [3] is an extension of the classical finite state automata with clock variables to model timing aspects. Timed automata has been shown to provide a useful formalism for describing systems in which timing constraints plays a critical factor for their correctness. The range of problems to which timed automata has been applied successfully includes distributed systems, secure authentication protocols, embedded systems, and much more. These applications have motivated the development of verification tools based on timed automata and temporal logic such as the UPPAAL model checker tool [6]. UPPAAL is a tool designed for modeling real-time systems using timed automata extended with features for concurrency, communication, data variables, and priority.

The atomic commitment protocols (ACPs) are one of the interesting examples of synchronous distributed protocols that are difficult to analyse from the perspective of model checking. The ACP protocol example is based on the description given in [7]. The protocol coordinates all the processes that participate in a distributed atomic transaction on whether to commit or abort (roll back) the transaction, while preserving data consistency. The protocol has received great attention from the research community since it has a large number of applications in transaction processing, databases and computer networking. The analysis of this protocol with the UPPAAL model checker tool is the subject of this paper.

We consider the protocol in two different crash models, the crash-stop model, in which processes that crashed never recovered, and the crash-recovery model, in which processes might crash and recover several times during the execution of the protocol. The models capture realistic temporal properties of the protocol (i.e. the coordinator times out if it does not receive a vote from the participant soon enough). One of the main difficulties in these models is that, one can not predict the future behaviour of the agents since they might remain correct and follow the rules of the original protocol or they might fail and follow a modified protocol according to the type of the fault.

During our analysis, we have been able to discover some interesting conclusions about the protocol. In particular in the crash-recovery model, we find that the central property of the protocol, the data consistency property, might be temporarily broken during the execution of the protocol. However, when the protocol does terminate, we find that the consistency is preserved. We describe a methodology for the use of dense-timed model checking to determine the longest execution time of an algorithm and then to discover the corresponding worst-case scenario. The methodology is partially automated. We illustrate the methodology using our case study. As UPPAAL does not allow nested modalities, some properties of the protocol can not be verified directly. We therefore strengthened the models and weakened the properties until UPPAAL could verify them

2 Preliminaries

2.1 Timed Automata and UPPAAL

Timed automata is an extension of the classical finite state automata with clock variables to model timing aspects [3]. Let X be a set of clock variables, then the set of $\Phi(X)$ of clock constraint 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.

A timed automata is formally defined as a tuple $(L, l_0, \Sigma, X, I, E)$, where L is a finite set of locations, l_0 is the initial location, Σ is a finite set of labels, X is a finite set of clocks, I is a mapping that maps each location $l \in L$ with some clock constraint in $\Phi(X)$, and $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a set of edges. An edge $(l, a, \phi, \sigma, l^{'})$ represents a transition from location l to location $l^{'}$ after performing the action a. The clock constraint ϕ determines when the edge is enabled and the set $\sigma \subseteq X$ gives the clocks to be reset with this edge. The semantics of a timed automaton $(L, l_0, \Sigma, X, I, E)$ is defined by associating a transition systems with it. A state (l, v) or configuration consists of the current location and the current values of clocks. The initial state is (l_0, v_0) where $v_0 = 0$ for all $x \in X$. A timed action is a pair (t, a) where $a \in \Sigma$ is an action performed by an Automata A after $t \in \mathbb{R}^+$ time units since A has been started. An execution of a timed automata $A = (L, l_0, \Sigma, X, I, E)$ 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:

$$\langle l_0, v_0 \rangle \xrightarrow{d_1} \xrightarrow{a_1} \langle l_1, v_1 \rangle \xrightarrow{d_2} \xrightarrow{a_2} \langle l_2, v_2 \rangle \xrightarrow{d_3} \xrightarrow{a_3} \langle l_3, v_3 \rangle ...$$

satisfying the condition $t_i = t_{i-1} + d_i$ for all $i \geq 1$. Recall that in timed automata models there are two types of transitions: delay transition d_i and action transition a_i.

The transition system in timed automata is infinite as clocks are real-valued, therefore timed automata models can not be directly model checked. Several techniques have been proposed to make the model checking problem decidable. The region [2] and zone [1] approaches, however, are finite abstractions that preserve Timed Computation Tree Logic (TCTL) formulas. The zone approach has been implemented in UPPAAL to make the model checking problem traceable.

UPPAAL [6] 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 dense-time model to describe systems, where each clock variable evaluates to a real number. UPPAAL model is a parallel composition of all of its timed automata. All automata start at its initial state (location) and run independently of each other unless synchronization with other automata is required. A transition is enabled when all enabling conditions are evaluated to true and all the synchronization statements are executed. If more than one transitions are enabled, one of them is chosen non-deterministically. There are two types of transitions in UPPAAL: (a) delay transitions that model the elapse of time, and (b) action transitions that execute an edge of the automata. Note that each transition consumes a finite amount of time that can be bounded by the use of invariants.

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 some trivial errors in the system design. The verifier allows the user to enter the properties to be verified in a restricted language of CTL. UPPAAL can verify safety, bounded liveness, and reachability properties. The syntax of the fragment of the UPPAAL's specification language relevant for this paper is given by the following grammar:

$$Prop ::= AG\phi \mid AF\phi \mid EG\phi \mid EF\phi \mid \phi_1 \rightarrow \phi_2$$

where ψ can be a clock constraint, a boolean combination of equalities or inequalities over local variables, or an expression of the form *process.location*. Intuitively, $AG\psi$ means that every reachable state in the model satisfies ψ , $AF\psi$ means in all possible paths in the model there is a state satisfies ψ, $EG\psi$ means there is a path where ψ is always true, $EF\psi$ indicates that there is a path where eventually ψ is true, and $\psi 1 \rightarrow \psi_2$ means whenever ψ_1 holds eventually ψ_2 will hold as well.

2.2 The Two Phase commit Protocol

This section presents an informal description of a protocol that solves the problem of processing a distributed transaction in a synchronous setting. The algorithm description here is adapted from [7] and following [16].

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. The protocol terminates whenever the coordinator receives all COMPLETION messages successfully.

The basic idea in the design of the protocol is to preserve data consistency in distributed systems in case of failure or delay in message transmission, assuming the systems are able to recover after failure. The above version of the 2PC protocol has frequently been the focus of studies of verification of distributed computing, but it is just one of several variants discussed in the paper. We focus here on one particular variant of the protocol in which the coordinator and the participants use a set of timers in order to detect and handle node failures. Note that the basic 2PC protocol (without node failures) has been proved correct, but what about the situations in which some participants can crash permanently at any point of time or crash and recover several times during the execution of the protocol? It is not immediately clear that there are no scenarios where atomicity and consistency can be violated. One of the benefits of verification by model checking is that it permits different scenarios of a protocol to be investigated efficiently without requiring reconstruction of possibly complex proofs.

2.3 Related Work

Some work has already been done on the verification of commitment protocols using formal techniques. The work in [17] has model checked the 2PC protocol using the LTSA tool, but they do not model explicitly the timing details of the protocol. Also they consider the protocol only in a crashed-stop model so they did not analyse it in a crash-recovery failure model as we have done in this paper. The protocol has been also analysed using process algebra mCRL2 [4]. The author considers a setting with two nodes, one of them is designed as a coordinator, and the other as participant. He claims that the protocol violates consistency property in case a single participant fails, but he has not given a clear scenario where the protocol violates data consistency. However, we believe that in order to study the protocol correctly, it should be considered in a setting with a larger number of participants not just with a single participant, so one can verify properties about participant i when for example participant j fails. Also, he considers the protocol only in a crash-stop model, so he does not consider it in a crash-recovery model. Simplified versions of the 2PC protocol (without timing, node failures, and recovery) have been also analysed in [12] using algebraic techniques. However, their analysis has not led them to discover any subtleties in the protocol. Closet to our work is the work of Ölveczky [20] who have taken the approach of explicitly including time in verifying the 2PC, using the Maude tool. He analysed the protocol under the assumption that some processes can crash and recover. However, he does not consider the questions we have studied concerning strong and weak termination, nor he does try to find the upper bound for termination in the crash-recovery model.

3 Modeling The 2-phase Commit Protocol in UPPAAL

In this section, we model the protocol as networks of timed automata using the UPPAAL model checker tool. We consider a setting with four processes, one as a coordinator and the other three as participants. We investigate the correctness of the protocol in two different crash models, the crash-stop model, and the crash-recovery model

3.1 The Coordinator

The coordinator process in the crash-recovery model is given in Figure 1. For space limitation we omit the crash-stop model since it is similar to the crash-recovery model except that processes do not execute a recovery protocol when they crash. The coordinator initially chooses non-deterministically its vote using the select statement `vote:votes[abort,commit]`. This is modelled by the initial transition from $m0$ to *begin*. We assume here that agents do not change their held votes. Then it will either start the protocol or it may fail. The coordinator (master) starts the protocol by broadcasting a commit request message to all the participants in the network. The broadcast channel `commit-request` is used for this purpose. It then activates a timer via sending a message in the channel `set` to a timeout automaton (see Figure 2). The coordinator uses the timeout automaton to measure the time units that are spent in waiting the participants' votes. Now the master will either receive all votes or it will time-out, if at least one of the participants has not sent his vote within a certain time limit. In case a time-out occurs, the coordinator will decide to abort the transaction and enforce (instruct) all the participants to abort. This is modeled in the transition from location $m6$ to location *incompleteTrans*. Intuitively, if the coordinator receives all the votes before the timeout expires, it will reset the timer and jump to location $m5$. Note that the channels in UPPAAL are not communicating channels but synchronising channels, in the sense they can not carry information. To model data transfer through channels we need to use global variables. The global array variable `vote[i]` is used to represent the announcement (vote) made by participant i. The `isVoted[i]` variable is used to indicate whether participant i has sent his vote to the coordinator. If the coordinator received all the votes successfully, he can then decide whether to commit or to abort the transaction. A function `decision(votes)` 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`. These actions are modelled in the transition from $m5$ to $m7$. The coordinator then waits c time units in location $m8$ in order to receive acknowledgement from the participants. The protocol ends successfully at location *finished* if the coordinator receives these signals within the time bound c. Note that the coordinator repeatedly sends the commit or abort command to all the participants that have not been responded, and wait for their acknowledgements. The location *crashed* indicates that the process has crashed due to a failure. The failure can be either temporary (soft) failure or permanent failure. In case a soft failure occurs, a restart action will be taken to recover the process, as modeled in the transition from *crashed* to *recovered*. When the process fails it stops all its activities, including sending messages to other processes, until it recovers. The local variable `status` shows the current status of the process, which could be one of the following values $\{correct, crash, term\}$, where $term$ stands for termination. At the beginning of the protocol all processes will be in a *correct* status but it can be changed to any other status during the execution of the protocol. The local variable `rcd` is used to restart the 2PC protocol after soft crashes of the coordinator. The value of the `rcd` variable determines the most recent state prior to a failure. Clearly, if the coordinator fails in the initial state, it simply restarts the protocol in the initial state. Failures in the final state do not require any action from the coordinator. For the remaining states, restart protocol will return the coordinator process to the state where the process was before the failure occurs. The variable `k` is used to record the number of times the process has crashed and recovered during the execution of the protocol. The local clock `time_to_recover` represents the time that the process can spend in the *recovered* location which is bounded by the constant `c1`.

3.2 The Participants

The template of the participants is depicted in Figure 3. All three participants i, j, k use the same template, but with different environment variables bound to the local variable of the template. The template is parametrised with the following variables: `vote, isVoted, ack`, and `outcome`. The use of these variables is already explained in the description of the coordinator template. Similar to the model of the coordinator, each participant initially chooses its vote non-deterministically and then waits for a request commit signal from the coordinator. If they do not receive that signal within the timeout limit, they will decide to abort the transaction and terminate. On the other hand, if they received the signal successfully, they will send their votes to the coordinator and then set the boolean variable `isVoted` to true. The participants then wait to receive either a signal to collect the final outcome of the transaction or a signal to

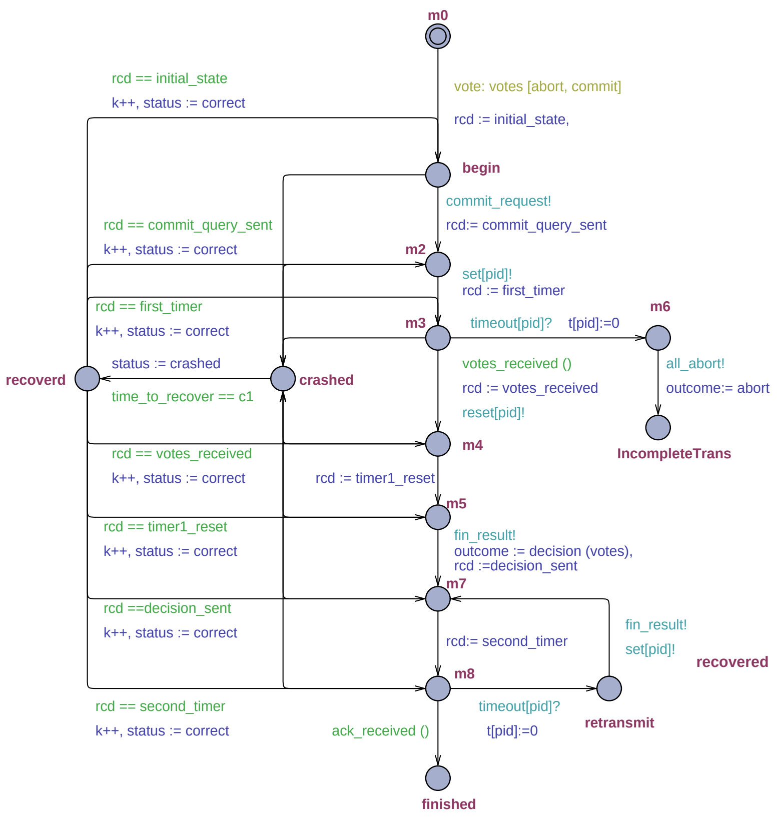

Figure 1. The coordinator template in the crash-recovery model

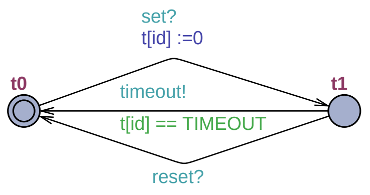

Figure 2. The timer template

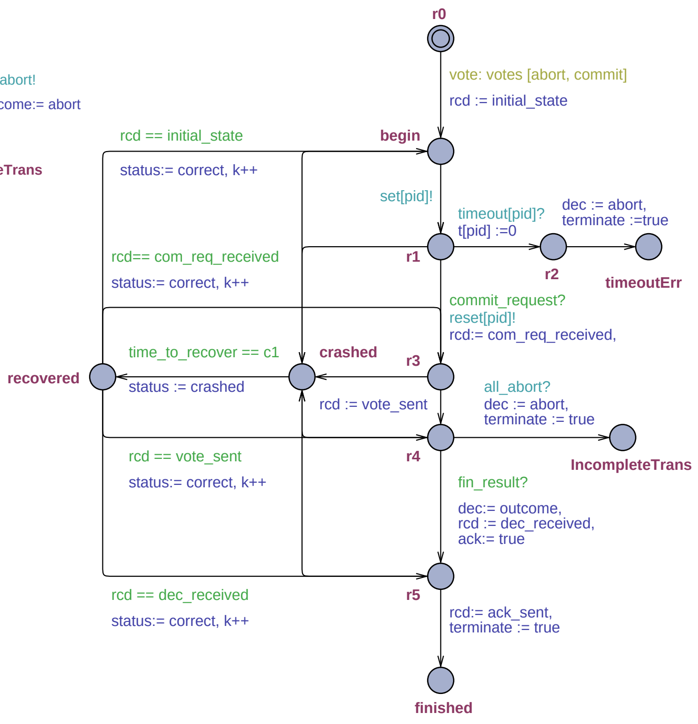

Figure 3. The participant template in the crash-recovery model

abort the transaction due to a timeout failure at the coordinator side. Once they receive the signal to collect the final result they will set their `ack` variable to true and then terminate. Note that the recovery protocol implemented at the participants' crash-recovery template is identical to the one discussed at the coordinator crash-recovery template.

4 Verification of the Protocol

In this section, we explain the properties that any distributed transaction protocol must satisfy which are: validity, atomicity, consistency, and termination.

4.1 Correctness Conditions

Before we discuss the correctness conditions of the protocol let us first formalise the behaviour of the processes in the crash models that we consider. The behaviour of the processes in the crash-stop (CS) model can be described by the following temporal formula.

$$\mathbf{CS} = \bigwedge_{i=1..n} \big((t = 0 \Rightarrow i.\mathtt{status} = \mathit{correct}) \wedge (i.\mathtt{status} = \mathit{crash} \Rightarrow \mathbf{AG}\,(i.\mathtt{status} = \mathit{crash})) \wedge (i.\mathtt{status} = \mathit{term} \Rightarrow \mathbf{AG}\,(i.\mathtt{status} = \mathit{term}))\big)$$

where t is a local clock to process i. The processes initially start with a correct status which might be changed to the crash status at any point of time. In case they remain correct they follow the rules of the original protocol until they terminate, however, if they crashed they stop all their activities and never recovered. On the other hand, we can describe the behaviour of the processes in the crash-recovery (CR) model as follows.

$$\mathbf{CR} = \bigwedge_{i=1..n} \big((t = 0 \Rightarrow i.\mathtt{status} = \mathit{correct}) \wedge \mathbf{AF}\,(i.\mathtt{status} = \mathit{term})\big)$$

The formula states that processes in the crash-recovery model start the protocol with a correct status but they might crash and recover several times during the execution of the protocol, however, there is a time after which processes will be up until they decide and then terminate.

For both models, we assume initially that no agent has received a message, no agent has decided, and all of them have a correct status, which can be formalized as follows.

$$\mathbf{I} = \bigwedge_{i=1..n} (\mathtt{m}_i = \perp \wedge\ dec_i = \perp \wedge\ \mathtt{status}_i = correct)$$

Now we turn to discuss the correctness conditions of the protocol. The first formula of interest is global atomicity (i.e. all processes must abort or all must commit.) As remarked above, it has been claimed that the 2-phase commit protocol is guaranteed the global atomicity even when a failure occurs during the execution of the protocol.

Specification 1a: (Safety property) Any two processes that have decided never decide differently.

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

We can also formalise atomicity property as follows.

$$\mathbf{AG}\ ((coord.dec = commit) \Rightarrow \bigwedge_{i=1..n} \neg(i.dec = abort))$$

Intuitively the above formula states that if the coordinator decided to commit the transaction then none of the participants will decide to abort it.

An important property of the protocol is that a decision made during its execution must be taken by all the participants. We can formalise this property as follows.

Specification 1b: (Liveness property) Every correct process eventually decides.

$$\mathbf{AF}\ (\bigwedge_{i=1..n} (i.dec = commit\ \vee\ i.dec = abort))$$

Note that the variable dec can take one of the following values $\{$*undecided*$, abort, commit\}$. All agents are initially undecided. By verifying atomicity property we verify implicitly consistency property since ensuring atomicity guarantees data consistency on all distributed database systems.

Note that through the execution of the distributed transaction, some of the involving processes might fail while executing, however, the effects of the transaction on the distributed databases must be all-or-nothing. This is very important when we consider the protocol in practical applications, where the processes that failed might recover later.

In the crash-stop model, we want to verify that if the coordinator crashed before announcing its knowledge of the outcome of the votes, then no-one commits the transaction.
Specification 1c: If the coordinator crashed then no-one commits unless the decision has already been made before the crash.

$$(C.\mathtt{status} = crash \wedge \mathtt{outcome} = undecided) \\ \Rightarrow \bigwedge_i (i.dec \neq commit)$$

Next, we want to verify whether the protocol guarantees validity property (i.e. if process i votes 'no' then 'no' is the only possible decision value, also if all processes vote 'yes' then 'yes' is the only decision value).

Specification 2: Abort-validity condition in the crash models.

$$(\bigvee_{i=1..n} (i.\mathtt{vote} = no)) \Rightarrow\ \mathbf{AF}\ (\mathtt{outcome} = abort)$$

Specification 3: Commit-validity is guaranteed in the crash models.

$$(\bigwedge_{i=1..n} (i.\mathtt{vote} = yes)) \Rightarrow\ \mathbf{AF}\ (\mathtt{outcome} = commit)$$

Next assuming some nodes can fail during the execution of the protocol, we want to verify that correct nodes still can terminate successfully.

Specification 4b: (Strong termination condition for the crash-stop model): Every nonfaulty process eventually decides and then terminates.

$$\mathbf{AG}\ (i.\mathtt{status} \neq crash) \Rightarrow\ \mathbf{AF}\ (i.\mathtt{status} = term)$$

Specification 4c: (Strong termination condition for the crash-recovery model) If all failures are repaired, then every participant eventually reaches a decision and terminate.

$$\mathbf{AF}\ (\bigwedge_{i=1..n} (i.\mathtt{status} = term))$$

4.2 Verification Results

In Table 1 we report the verification results of the protocol for both crash models. We report 'NA' in the table when the specification is not applicable for the given failure model and 'X' when UPPAAL is unable to verify the given specification. UPPAAL successfully verifies *Specification 1a* in both failure models which proves that the protocol guarantees atomicity property in all possible scenarios including failure scenarios. When model checking *Specification 1b* in the crash-stop model we found the specification fails where UPPAAL produced a counterexample. The generated counterexample represents a typical scenario where the 2PC protocol blocks. This happens when the coordinator crashes at the beginning of the second phase before announcing his knowledge of the outcome of the votes. In this case all processes that voted to commit in the first phase can not decide. On the other hand, all processes that voted to abort will know that the only possible decision is to abort and therefore they do not need to wait for the decision of the coordinator.

Note that in the crash-recovery model it might not be possible for the coordinator to deliver its decision to all the participants at the same time, since some of the participants might be crashed at the time at which the coordinator transmits its decision. So we might have situations in the protocol where participant i commits its subtransaction at real time t, while j commits at $t + 4$, k commits at $t + 10$, and C commits at $t + 20$. The protocol allows such situations to occur, in particular when some processes fail during the execution of the protocol. Our analysis shows that atomicity and consistency properties in the crash-recovery model might be *temporarily broken* if some processes crash for a period of time, while some other processes commit the results of their transactions during that period of time. So temporal failures might lead to temporal inconsistency. However, when all failures are eventually repaired we found the model satisfies atomicity and consistency properties.

Next we discuss the question whether the protocol can succeed to terminate. Note that UPPAAL specification language does not allow nested modalities, we therefore could not verify specification 4a. We alternatively considered a weaker termination condition for the crash-stop model via using the same termination condition we use for the crash-recovery model (specification 4b). We found the specification fails where UPPAAL discovered a counter-example where the protocol can not terminate. This happens when the coordinator fails after receiving the votes of the participants. In this case the transaction remains unterminated for indeterminate period of time, since all processes are waiting the decision of the coordinator. Note that the participants can not solve the transaction even if they cooperate with each other since they do not know the vote of the coordinator. On the other hand, UPPAAL successfully verifies termination condition in the crash-recovery model.

Specification	Crash-stop model	Crash-recovery model
1a	holds	holds
1b	fails	holds
1c	holds	NA
2	holds	holds
3	holds	holds
4a	X	NA
4b	fails	holds

Table 1. The verification Results of the 2PC protocol in UPPAAL

5 A Methodology for Finding Longest Execution Path of Timed Systems Using Model Checking

When analysing real-time systems, we are often interested in studying questions related to the Worst-Case Execution Time (WCET) and the scenarios that lead to the WCET. Since determining upper bounds on execution times is a necessary step in the development and verification process for real-time systems.

In general, it is difficult to determine the exact worst-case scenario for a given system. Typically, researchers use static timing analysis techniques to compute the worst-case execution time for the given system. These techniques have been shown to be potentially inaccurate, since they rely on observing the worst-case scenarios during testing or simulation or even via code inspection. Moreover, processes in the system can have many timers (clocks) whose timing values can vary with each event processes perform, therefore, the timing analysis can be very complicated and time consuming. Thus a need to automate the timing analysis is highly desirable. Since model checking comprehensively explores all possible traces of the system, it can be used to answer such questions efficiently.

It is claimed in [21] that model checking is inadequate for WCET analysis. However, Metzner in [18] has shown that model checking can be used efficiently for WCET analysis. He used model checking to improve WCET analyses for hardware with caching. In fact there are few worked examples on using model checking for WCET analysis. The work in [11] has used model checking to measure WCET of real-world, modern processors with good performance. A combination of static methods and model checking has been used in [14] to analyse WCET in programs whose loops are allowed to change dynamically.

We describe here a partially automated approach, using timed model checking, that can be followed to find the longest execution path of a given timed-based system (provide this terminates in a bounded time). In order to verify the maximum time that the system takes to terminate, we use the timed model checker to verify formulas of the following form:

$$\textbf{AF}\ ((system.finished)\ \wedge t_1 \leq x)$$

where $system$ is a process that describes the behavioral model of the analysed system, the location $finished$ represents the location at which the process terminates, and t_1 is a timer that is used to measure the execution time of the system. The timer should be set initially to 0. Intuitively, the formula states that the system can always reach the location $finished$ within the time bound x.

Now in order to verify the correctness of the above formula, the user needs first to guess a value for x. The model checker can then be used to verify the correctness of the guess. In general, the guess concerning the value of the timer may be incorrect, and the model checker will report the error. In this case, the model checker can be used to generate an error trace. The next step of the process requires to analyse this error trace. Recall that in dense-time models all the clocks progress synchronously. This may make the analysis a bit harder. However, the error trace will help to estimate the amount of time that the process needs in order to reach the required location. As a result of this analysis, a new guess for the clock value will be introduced. The model checker is then invoked again to check the new guess. This process is iterated until a guess is produced for which the formula holds. By the end of this process the user may be able also to discover the scenarios that lead to the WCET via analysing the generated counterexamples that he obtains while discovering the WCET.

We describe now a confirmation process by which the user can verify the correctness of his WCET calculations and then discover the corresponding worst-case scenario. Note that the user can skip the confirmation process if he can discover worst case scenarios during his analysis of the WCET. However, the process boosts the confidence of the user about the correctness of his analysis.

Once we know the longest execution time of the system, we can investigate the path(s) that lead to the worst-case scenario using also a process based on timed model checking technology. In fact the process of finding the worst-case scenario is much harder than the above one since it might require some ingenuity on the part of the user. Also it requires some manual effort of the user. The first step of the process requires the user to guess a scenario that is supposed to be the worst-case scenario of the system (by inspection and human reasoning). The next step is then to formalise the supposed scenario using the language of the given model checker via building what we call a test automaton that captures the intended scenario. The third step is to model check the actual model of the system with the constructed test automaton, and then to verify formulas of the following form:

$$\mathbf{EF}\ ((test.finished)\ \wedge t_2 = y)$$

where $test$ is a process that models the proposed worst-case scenario, and t_2 is a timer that is used to measure the execution time of that scenario. The formula simply verifies whether there exists a path where process $test$ can reach location $finished$ with the time bound y. Now the user needs to follow the first described process in order to verify the above formula and to find the value of y that makes the formula hold. Intuitively, if the value of y matches the value of x, then we found the worst-case scenario that we seek. On the other hand, if the two values do not match then the proposed scenario is not the worst-case scenario and the user needs to guess a new scenario and then to repeat the second process again.

5.1 Finding Longest Execution Path of The 2PC Protocol

Since UPPAAL successfully verifies all termination properties in the crash-recovery model, it is then interesting to study the upper bound for termination in that model. Note that processes may either terminate in the first phase of the protocol if some processes crash early, or they may terminate after successfully completing the protocol. The repeated run of the simulation of UPPAAL and the manual inspection lead us to guess that the following scenario is the worst-case scenario of the model. We expect that the worst case scenario results when at least one of the participants let say i crashed $\mathtt{r}$ of times during the first phase of the protocol such that $\mathtt{r} \leq \mathtt{k}$, but it recovered before the timeout of the coordinator expires ($\mathtt{r} \times \mathtt{c1} < C.\mathtt{timeout}$), this guarantees that the coordinator will receive the votes of all participants successfully and then go to the second phase of the protocol. Recall that $\mathtt{k}$ represents the maximum number of times processes can crash during the protocol execution, and that $\mathtt{c1}$ represents the upper bound for crash recovery. Now assume that the coordinator crashed $\mathtt{k}$ times before sending the decision and recovered after $\mathtt{k}\times\mathtt{c1}$ time units, and that when it recovered some other participant $j \neq i$ crashed also $\mathtt{k}$ times before it receives the decision of the coordinator. Recall that in the second phase of the protocol the coordinator every $T = C.\mathtt{timeout}$ will retransmit its decision if some of the participants have not received it successfully. From the structure of the protocol, we note that if all participants went successfully to the second phase, then all of them will be able to receive the decision of the coordinator within $(2 \times \mathtt{k} \times \mathtt{c1} + \mathtt{timeout})$. This represents the maximum length of time processes can spend in the second phase of the protocol, which includes the time delay results if the coordinator and any of the participants crash $\mathtt{k}$ of times. It is then clear that within $(\mathtt{r}\times\mathtt{c1}+2\times\mathtt{k}\times\mathtt{c1}+\mathtt{timeout})$ time units the coordinator will be able to finish the protocol and terminate. Recall that we assume here that processes communicate with each other using perfect communication links so there is no time delay. This assumption is made only to simplify the computations. Note that the maximum time for termination in the crash-recovery model does not depend directly on the number of processes that crash during the protocol execution. For example, failure of n participants $\mathtt{k}$ of times during the second phase will delay termination exactly $\mathtt{k} \times \mathtt{c1}$ time units which equal to the time delay results if a single node fails $\mathtt{k}$ of times! This is because of the time interleaving between the processes running in the protocol.

Following the above described methodology we will show how we can measure the longest execution time of the protocol and then prove that the above described scenario is indeed the worst-case scenario. In order to verify the maximum time that the protocol takes to terminate, we use the UPPAAL model checker to verify the following formula:

$$\textbf{AF}\ ((C.\texttt{finished} \lor C.\texttt{IncompleteTrans}) \land \texttt{t} \leq x)$$

Intuitively, the above formula states that the coordinator C can always reach the `finished` location or the `IncompleteTrans` location within the time bound x. Recall that the protocol can either terminate at location `finished` if the transaction completed successfully, or at location `IncompleteTrans` if some failures occurred and caused the timeout of the coordinator or the participants to expire during the first phase of the protocol.

We verify the formula for the following parameter values:

- `c1 = 2`, `k = 2`, `timeout = 4`,
- `c1 = 2`, `k = 4`, `timeout = 4`,
- `c1 = 3`, `k = 2`, `timeout = 4`,
- `c1 = 2`, `k = 2`, `timeout = 6`

The selection of the parameter values in this way helps to show how each parameter influences the execution time of the protocol. We discover by repeated verification attempts that for the above parameter values respectively with `t` less than or equal to 14, 22, 19, and 18 will ensure that the protocol can always terminate.

Since we obtained the longest execution times of the model under different settings, the next step is then to prove that the above described scenario indeed represents the longest execution path of the model. To do so we need to enforce the protocol to go through certain selected paths and then to measure the execution time of these paths. If the measured values matched the above worst-case execution times then the scenario indeed represents the worst-case scenario of the model. We add a special automaton (called the test automaton) which models the supposed scenario via interacting with the actual model of the protocol, as shown in Figure 4.

The test automaton is forced to only take the paths leading to the supposed scenario. The automaton first waits to receive a signal from any of the three participants in case they crashed and recovered `r` times during the first phase of the protocol. It then waits to receive a signal from the coordinator if it crashed and recovered `k` times at the beginning of the second phase. Now according to the supposed worst-case scenario, the test automaton needs again to wait for a signal from any of the participants if they crashed and recovered `k` times in the second phase. The test automaton terminates whenever the coordinator send it a termination signal.

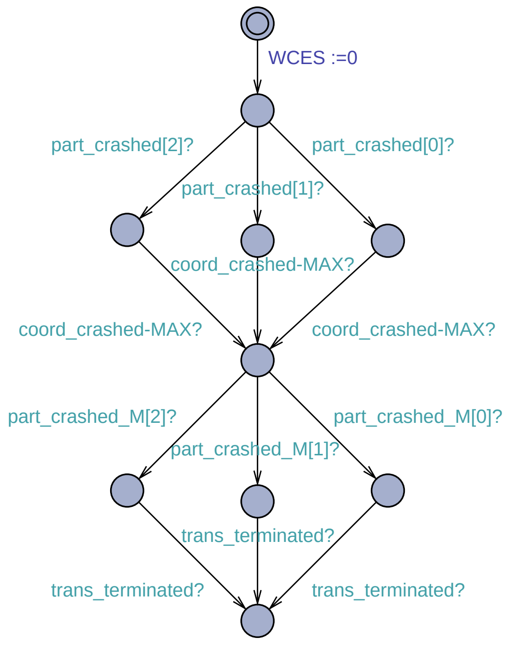

Figure 4. The test automaton of the worst-case scenario

We then verify the following formula:

$$\textbf{EF}\ (test.\texttt{finished} \land \texttt{WCES} = y)$$

We discover by repeated verification attempts that for the same above parameter values respectively with `WCES` less than or equal to 14, 22, 19, and 18 there exists a path where test automaton can reach the location `finished`. These values match exactly the longest execution time of the model when we verified it under the same parameter values. We therefore conclude that the scenario modeled by the test automaton is indeed the worst-case scenario.

6 Conclusion and Future Work

We have modeled and analysed the 2-phase commit protocol in different failure settings using the UPPAAL model checker tool. The consideration of time in the analysis allows us to specify the protocol in detail, with timeouts, retransmissions, and recovery mechanisms, which is closer to an implementation. We have chosen UPPAAL among all available real-time model checkers because it is the most well-known and mature tool in the literature. However, we found that UPPAAL does not support a full TCTL language and therefore we were unable to consider all interesting properties of the protocol. We described also a partially automated methodology based on dense-timed model checking that can be followed to discover the longest execution paths of timed systems. An interesting research challenge is then to fully automate the methodology. We plan to address this in future work.

References

[1] R. Alur. Timed automata. In *NATO ASI Summer School on Verification of Digital and Hybrid Systems*. 1998.

[2] 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. IEEE Computer Society Press, 1990.

[3] R. Alur and D. Dill. A theory of timed automata. In *TCS*, pages 183–235. 1994.

[4] M. Atif. Analysis and verification of two-phase commit and three-phase commit protocols. In *Emerging Technologies ICET'09*, pages 326–331, 2009.

[5] H. Attiya, C. Dwork, N. Lynch, , and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM, 41(1):*, pages 122–15, 1994.

[6] G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-time Systems (SFM-RT 2004)*, pages 200–236. Springer, 2004.

[7] A. P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. Addison-Wesley, 1987.

[8] M. Bodlaender, J. Guidi, and L. Heerink. Enhancing discovery with liveness. In *CCNC'04, IEEE Computer Society Press*, 2004.

[9] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 linklocal addresses (2004). 2004.

[10] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *TACAS'03. Number 2619*. LNCS, Springer-Verlag, 2003.

[11] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, pages 113–123, 2010.

[12] W. Janssen and J. Zwiers. Protocol design by layered decomposition. In *Proc. FTRTFTS 2nd International Symposium*, pages 307–326. LNCS, 1992.

[13] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[14] S. Kim, H. D. Patel, and S. A. Edwards. Using a Model Checker to Determine Worst-case Execution Time. 2009.

[15] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of Lecture Notes in Computer Science, pages 147–166. Springer Verlag, 1996.

[16] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[17] J. Magee. Analyzing synchronous distributed algorithms. 2003.

[18] A. Metzner. Why model checking can improve WCET analysis. In *Proceeding of the International Conference on Computer-Aided Verification (CAV)*, pages 334–347, 2004.

[19] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. In *Communications of the ACM*, pages 993–999. 1978.

[20] 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.

[21] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *Bernhard Steffen and Giorgio Levi, editiors, VMCAI*, pages 309–322, 2004.

[22] M. Zhang and F. Vaandrager. Analysis of a protocol for dynamic configuration of IPv4 link local addresses using Uppaal. Technical report, NIII, Radboud University Nijmegen (2005), 2005.