

# Model Checking the Time to Reach Agreement\*

MARTIJN HENDRIKS

Institute for Computing and Information Sciences,  
Radboud University Nijmegen, The Netherlands  
*M.Hendriks@cs.ru.nl*

## Abstract

The timed automaton framework of Alur and Dill is a natural choice for the specification of partially synchronous distributed systems. The past has shown, however, that verification of these systems by model checking usually is very difficult. Therefore, model checking techniques have thus far not really been used for their design, even though these techniques are widely used in other areas, e.g., hardware verification. The present paper demonstrates that the revolutionary development of both the usability and the efficiency of model checking tools may change this. It is shown that a complex partially synchronous distributed algorithm can easily be modeled with the UPPAAL model checker, and that it is possible to analyze some interesting and non-trivial instances with reasonable computational resources. Clearly, such analysis results can greatly support the design of these systems: model checking tools may provide valuable early feedback on subtle design errors and hint at system invariants that can subsequently be used in the general correctness proof.

*Keywords:* Distributed systems, agreement algorithm, partially synchronous model, model checking, timed automata.

## 1 Introduction

Distributed systems are in general hard to understand and to reason about due to their complexity and inherent non-determinism. That is why formal models play an important role in the design of these systems: one can specify the system and its properties in an unambiguous and precise way, and it enables a formal correctness proof. The I/O-automata of Lynch and Tuttle provide a general formal modeling framework for distributed systems [21, 20, 19]. Although the models and proofs in this framework can be very general (e.g., parameterized by the number of processes or the network topology), the proofs require – as usual – a lot of human effort.

Model checking provides a more automated, albeit less general way of proving the correctness of systems [13]. The approach requires the construction of a

---

\*Supported by the European Community Project IST-2001-35304 AMETIST (Advanced Methods for Timed Systems), <http://ametist.cs.utwente.nl/>.

model of the system and the specification of its correctness properties. A model checker then automatically computes whether the model satisfies the properties or not. The power of model checkers is that they are relatively easy to use compared to manual verification techniques or theorem provers, but they also have some clear drawbacks. In general only *instances* of the system can be verified (i.e., the algorithm can be verified for 5 processes, but not for  $n$  processes). Furthermore, model checking suffers from the state space explosion problem: the number of states grows exponentially in the number of system components. This often renders the verification of realistic systems impossible.

Model checkers still can be useful for the design of distributed systems. Consider the following approach. First, one specifies the system in the language of the model checker. This can reveal inconsistencies and incompletenesses. Second, the model can be simulated using the model checker. This also may reveal design errors in an early stage of the design phase. When one is satisfied with the model, then, as a third step, one can try to verify some interesting properties for small instances of the system. Finally, if one has enough faith in the correctness of the system, then one can start with the construction of a general proof (either by hand or with a theorem prover), which in general is a very time-consuming task. The intuitions that one has gotten during the work with the model checker (*and* the invariants that were possibly obtained for some instances in the third step) can, however, make the construction of the proof less cumbersome. The work of [18] and [22] demonstrates the feasibility and effectiveness of the first three steps of this approach. In both papers, the SPIN model checker is used to give feedback on IEEE standards that at that time were still under development.

A class of distributed systems for which model checking has yielded no apparent successes is the subclass of partially synchronous systems in which (i) message delay is bounded by some constant, and (ii) many messages can be in transit simultaneously. In the partially synchronous model, system components have some information about timing, although the information might not be exact. It lies between the extremes of the synchronous model (the processes take steps simultaneously) on one end and the asynchronous model (the processes take steps in an arbitrary order and at arbitrary relative speeds) on the other end [19]. The timed automata framework of Alur and Dill [2] is a natural choice for the specification of such systems (as is the Timed I/O-automaton framework [17], which, however, does not support model checking). Verification of these systems by model checking is often very difficult since every message needs its own clock to model the bounds on message delivery time. This is disastrous since the state space of a timed automaton grows exponentially in the number of clocks. Moreover, if messages may get lost or message delivery is unordered, then on top of that also the discrete part of the model explodes rapidly.

Many realistic algorithms and protocols fall into the class of “difficult” partially synchronous systems. Examples include the sliding window protocol for the reliable transmission of data over unreliable channels [23, 11], a protocol to monitor the presence of network nodes [9, 16], and the ZeroConf protocol whose purpose is to dynamically configure IPv4 link-local addresses [10, 24]. Furthermore, the agreement algorithm described in [3] (see also Chapter 25 of [19]) also is a partially

synchronous system that is difficult from the perspective of model checking. The analysis of this algorithm with the UPPAAL model checker is the subject of the present paper. It is shown that some non-trivial instances can be formally verified (which has not been done before to the author's knowledge). Our results provide evidence that the class of partially synchronous distributed systems, which is an important class since many realistic algorithms and protocols fall into it, is within reach of the current state-of-the-art model checking tools.

The remainder of this paper is structured as follows. The timed automaton framework and the UPPAAL model checker are briefly introduced in Section 2. Section 3 then presents an informal description of the distributed algorithm of [3], which consists of two parts: a timeout task and a main task. Section 4 describes the UPPAAL model that is used to verify the timeout task. A model for the parallel composition of the timeout task and the main task is proposed in Section 5. Two properties of the timeout task that have been verified in Section 4 are used to reduce the complexity of this latter model. Finally, Section 6 discusses the present work. The UPPAAL models from this paper are available at <http://www.cs.ru.nl/ita/publications/papers/martijnh/>.

## 2 Timed Automata

This section provides a very brief overview of timed automata and their core semantics, and of the UPPAAL tool, which is a model checker for timed automata. The reader is referred to [6] and [8] for more details.

Timed automata are finite automata that are extended with real valued clock variables [2]. Let  $X$  be a set of clock variables, then the set  $\Phi(X)$  of clock constraints  $\phi$  is defined by the grammar  $\phi := x \sim c \mid \phi_1 \wedge \phi_2$ , where  $x \in X$ ,  $c \in \mathbb{N}$ , and  $\sim \in \{<, \leq, =, \geq, >\}$ . A clock interpretation  $\nu$  for a set  $X$  is a mapping from  $X$  to  $\mathbb{R}^+$ , where  $\mathbb{R}^+$  denotes the set of positive real numbers including zero. A clock interpretation  $\nu$  for  $X$  satisfies a clock constraint  $\phi$  over  $X$ , denoted by  $\nu \models \phi$ , if and only if  $\phi$  evaluates to *true* with the values for the clocks given by  $\nu$ . For  $\delta \in \mathbb{R}^+$ ,  $\nu + \delta$  denotes the clock interpretation which maps every clock  $x$  to the value  $\nu(x) + \delta$ . For a set  $Y \subseteq X$ ,  $\nu[Y := 0]$  denotes the clock interpretation for  $X$  which assigns 0 to each  $x \in Y$  and agrees with  $\nu$  over the rest of the clocks. We let  $\Gamma(X)$  denote the set of all clock interpretations for  $X$ .

A timed automaton then is defined by a tuple  $(L, l^0, \Sigma, X, I, E)$ , where  $L$  is a finite set of locations,  $l^0 \in L$  is the initial location,  $\Sigma$  is a finite set of labels,  $X$  is a finite set of clocks,  $I$  is a mapping that labels each location  $l \in L$  with some clock constraint in  $\Phi(X)$  (the *location invariant*) and  $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$  is a set of edges. An edge  $(l, a, \phi, \lambda, l')$  represents a transition from location  $l$  to location  $l'$  on the symbol  $a$ . The clock constraint  $\phi$  specifies when the edge is enabled and the set  $\lambda \subseteq X$  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 system with it. A state is a pair  $(l, \nu)$ , where  $l \in L$ , and  $\nu \in \Gamma(X)$  such that  $\nu \models I(l)$ . The initial state is  $(l^0, \nu^0)$ , where  $\nu^0(x) = 0$  for all  $x \in X$ . There are two types of transitions (let  $\delta \in \mathbb{R}^+$  and let  $a \in \Sigma$ ). First,  $((l, \nu), (l, \nu + \delta))$  is a  *$\delta$ -delay transition* iff

$\nu + \delta' \models I(l)$  for all  $0 \leq \delta' \leq \delta$ . Second,  $((l, \nu), (l', \nu'))$  is an *a-action transition* iff an edge  $(l, a, \phi, \lambda, l')$  exists such that  $\nu \models \phi$ ,  $\nu' = \nu[\lambda := 0]$  and  $\nu' \models I(l')$ . Note that location invariants can be used to specify progress, and that they can cause time deadlocks.

The transition system of a timed automaton is infinite due to the real valued clocks. The region and zone constructions, however, are finite abstractions that preserve Timed Computation Tree Logic (TCTL) formulas and a subset of TCTL formulas (most notably reachability) respectively [1, 14]. This enables the application of finite state model checking techniques as implemented by UPPAAL for instance.

The UPPAAL modeling language extends the basic timed automata as defined above with bounded integer variables and binary blocking (CCS style) synchronization. Systems are modeled as a set of communicating timed automata. The UPPAAL tool supports simulation of the model and the verification of reachability and invariant properties. The question whether a state satisfying  $\phi$  is reachable can be formalized as **EF**( $\phi$ ). The question whether  $\phi$  holds for all reachable states is formalized as **AG**( $\phi$ ). If such a property is not satisfied, then UPPAAL can give a run that proves this. This run can be replayed in the simulator, which is very useful for debugging purposes.

### 3 Description of the Algorithm

This section presents an informal description of an algorithm that solves the problem of *fault-tolerant distributed agreement* in a partially synchronous setting [3] (see also Chapter 25 of [19]). A system of  $n$  processes, denoted by  $p_1, \dots, p_n$ , is considered, where each process is given an input value and at most  $f$  processes may fail. Each process that does not fail must eventually (termination) choose a decision value such that no two processes decide differently (agreement), and if any process decides for  $v$ , then this has been the input value of some process (validity). The process's computation steps are atomic and take no time, and two consecutive computation steps of a non-faulty process are separated  $c_1$  to  $c_2$  time units. The processes can communicate by sending messages to each other. The message delay is bounded by  $d$  time units, and message delivery is unordered. Furthermore, messages cannot get lost nor duplicated. The constant  $D$  is defined as  $d + c_2$ . As mentioned above,  $f$  out of the  $n$  processes may fail. A failure may occur at any time, and if a process fails at some point, then an arbitrary subset of the messages that would have been sent in the next computation step, is sent. No further messages are sent by a failed process. It is convenient to regard the algorithm, which is run by every process, as the merge of a *timeout task* and a *main task*, such that a process's computation step consists of a step of the timeout task followed by a step of the main task.

The goal of the of the timeout task is to maintain the running state of all other processes. This is achieved by broadcasting an  $(alive, i)$  message every computation step. If process  $p_i$  has run for sufficiently many computation steps without receiving an  $(alive, j)$  message (from process  $p_j$ ), then it assumes that  $p_j$  halted ei-

ther by decision or by failure<sup>1</sup>. Figure 1 contains the description of a computation step of the timeout task of process  $p_i$  in precondition-effect style. The boolean variable *blocked* is used by the main task to stop the timeout task. Initially, this boolean is *false*. It is set to *true* if the process decides. The other state components are a set *halted*  $\subseteq \{1, \dots, n\}$ , initially  $\emptyset$ , and for every  $j \in \{1, \dots, n\}$  a counter *counter*( $j$ ), initially set to  $-1$ . Additionally, every process has a message buffer *buff* (a set), initially  $\emptyset$ .

```

Precondition:
   $\neg$  blocked
Effect:
  broadcast((alive, $i$ ))
  for  $j := 1$  to  $n$  do
    counter( $j$ ) := counter( $j$ ) + 1
    if (alive, $j$ )  $\in$  buff then
      remove (alive, $j$ ) from buff
      counter( $j$ ) := 0
    else if counter( $j$ )  $\geq \lfloor \frac{D}{c_1} \rfloor + 1$  then
      add  $j$  to halted
  od

```

Figure 1: The timeout task for process  $p_i$ .

Two properties of the timeout task have been proven in [3].

- $A_1$  If any  $p_i$  adds  $j$  to *halted* at time  $t$ , then  $p_j$  halts, and every message sent from  $p_j$  to  $p_i$  is delivered strictly before time  $t$ .
- $A_2$  If  $p_j$  halts at time  $t$ , then every  $p_i$  either halts or adds  $j$  to *halted* by time  $t + T$ , where  $T = D + c_2 \cdot (\lfloor \frac{D}{c_1} \rfloor + 1)$ .

Figure 2 contains the description of a computation step of the main task of process  $p_i$  in precondition-effect style. Apart from the input value  $v_i$  and the state components used by the timeout task, there is one additional state component, namely the rounds counter  $r$ , initially zero. The input values are assumed to be either zero or one for simplicity<sup>2</sup>.

Three main results that are obtained in [3] are the following.

- $M_1$  (Agreement, Lemma 5.9 of [3]). No two processes decide on different values.
- $M_2$  (Validity, Lemma 5.10 of [3]). If process  $p_i$  decides on  $n$ , then  $n = v_j$  for some process  $j$ .
- $M_3$  (Termination, Theorem 5.1 of [3]). The upper bound on the time to reach agreement equals  $(2f - 1)D + \max \{T, 3D\}$ .

<sup>1</sup>The message complexity of this algorithm is quite high. Recently, an alternative with an adjustable “probing load” for each node has been proposed in [9] and analyzed in [16].

<sup>2</sup>An extension to an arbitrary input domain is discussed in [3].

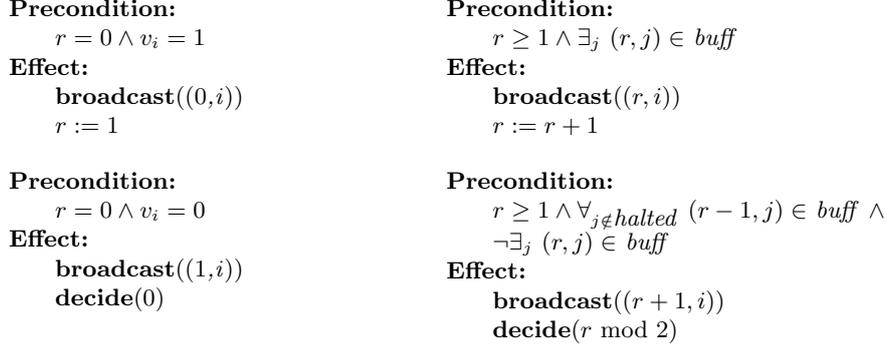


Figure 2: The main task for process  $p_i$ .

## 4 Verification of the Timeout Task

### 4.1 Modeling the Timeout Task

Note that every process runs the same algorithm, and that the timeout parts of different processes do not interfere with each other. Therefore, only two processes are considered, say  $p_i$  and  $p_j$ . By the same argument, only one direction of the timeout task is considered:  $p_i$  (*Observer*) keeps track of the running state of  $p_j$  (*Process*). As required by the algorithm, *Process* broadcasts an *alive* message at each computation step. This action is modeled by a *b*-synchronization, which activates an instance of the broadcast template, shown in Figure 3. This template is parameterized with a constant *id* in order to give each instance a unique identifier. Clearly, the UPPAAL model must ensure *output enabledness* of *Process*: it must be able to broadcast the alive message when it wants to. Since the maximal number of simultaneous broadcasts equals  $\lfloor \frac{d}{c_1} \rfloor + 2$ , this many instances of the broadcast template must be present in the model. The guard  $turn()$  and the assignments to  $active[id]$  implement a trick to reduce the reachable state space by exploiting the symmetry among the broadcast instances<sup>3</sup>. After a *b*-synchronization, a broadcast automaton may spend at most *d* time units in location *sending*, which is modeled using the local clock *x*. The actual message delivery is modeled by the assignment  $alive=true$  on the transition back to *idle*. The reset of the global clock *t* is used for the verification of property  $A_1$ .

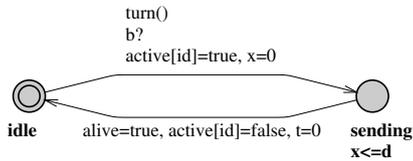


Figure 3: The broadcast template.

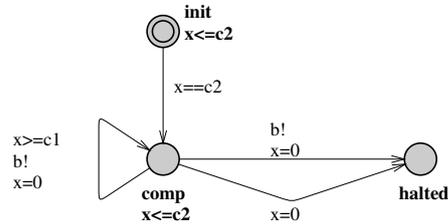


Figure 4: The *Process* automaton.

<sup>3</sup>The next release of UPPAAL will hopefully support symmetry reduction, which can automatically exploit the symmetry among broadcast automata [15].

Figure 4 shows the UPPAAL automaton of the merge of the timeout task and abstract main task of *Process* (the only functionality of the main task is to halt). It has one local clock  $x$  to keep track of the time between two consecutive computation steps. The *Process* automaton must spend exactly  $c_2$  time units in the initial location *init* before it takes the transition to location *comp* (the reason for this is explained below). It then immediately either fails or does a computation step. Failure of *Process* is modeled by the pair of edges to *halted*, which models the non-deterministic choice of the subset of messages to send. The computation step is modeled by the self-loop and by the upper transition to *halted* (a decision transition that blocks the timeout task)<sup>4</sup>. Note that  $x$  is reset on every edge to *halted* for verification purposes.

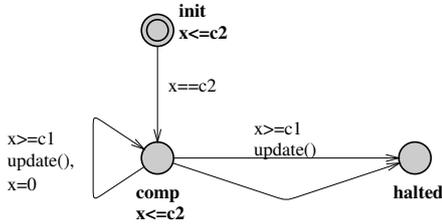


Figure 5: The *Observer* automaton.

```

void update ()
{
  cnt++;
  if (alive)
  {
    alive = false;
    cnt = 0;
  }
  has_halted = cnt >= (D/c1) + 1;
}

```

Figure 6: The *update()* function.

Figure 5 shows the automaton for the *Observer*, which is the composition of an abstract main task (whose only purpose again is to halt) and the “receiving part” of the timeout task. It has a local integer variable  $cnt$ , initialized to  $-1$ , and a local clock  $x$ . Furthermore, the boolean  $has\_halted$  models whether  $Process \in halted_{Observer}$ .

The *Observer* automaton must first spend  $c_2$  time units in the initial location before taking the edge to location *comp*. Then, it must immediately either do a computation step or fail. The computation step is modeled by the self-loop and by the upper transition to *halted*. The assignment *update()* updates the variables  $cnt$ ,  $has\_halted$  and  $alive$  as specified in Figure 6. Failure is modeled by the lower edge to *halted*.

Both the *Observer* automaton and the *Process* automaton must first spend  $c_2$  time units in their initial location. This is a modeling trick to fulfill the requirement from [3] that “every process has a computation or failure event at time 0”. I.e., our model starts at time  $-c_2$ . (If UPPAAL would allow the initialization of a clock to any natural number, then both initial locations can be removed.)

<sup>4</sup>A straightforward model contains a third edge to *halted* with the guard  $x \geq c_1$ , the synchronization  $b!$ , and the reset  $x = 0$ . Such an edge is, however, “covered” by the present upper edge to *halted* and can therefore be left out

## 4.2 Verifying the Timeout Task

Property  $A_1$  is translated to the following invariant property of the UPPAAL model (a broadcast automaton with identifier  $i$  is denoted by  $b_i$ ):

$$\mathbf{AG} \left( \begin{array}{c} has\_halted \longrightarrow \\ (Process.halted \wedge \forall_i b_i.idle \wedge t > 0) \end{array} \right) \quad (1)$$

The state property  $\forall_i b_i.idle \wedge t > 0$  ensures that all messages from *Process* to *Observer* are delivered strictly before the conclusion of *Observer* that *Process* halted. Property  $A_2$  is translated as follows:

$$\mathbf{AG} \left( \begin{array}{c} (Process.halted \wedge Process.x > T) \\ \longrightarrow \\ (Observer.halted \vee has\_halted) \end{array} \right) \quad (2)$$

The branching time nature of  $A_2$  is specified by this invariance property due to the structure of our model:  $Process.x$  measures the time that has been elapsed since *Process* arrived in the location *halted*.

Properties (1) and (2) have been verified for the following parameter values:

- $c_1 = 1, c_2 = 1$  and  $d \in \{0, 1, 2, 3, 5, 10\}$ .
- $c_1 = 1, c_2 = 2$  and  $d \in \{0, 1, 2, 3, 4, 5, 6\}$ , and
- $c_1 = 9, c_2 = 10$  and  $d \in \{5, 9 - 11, 15, 20, 50\}$ .

## 5 Verification of the Algorithm

The UPPAAL model of the parallel composition of the main task and the timeout task, which is used to verify properties  $M_1$ – $M_3$ , is presented in this section. It is assumed that every process receives an input by time zero (synchronous start), since otherwise the state space becomes too big to handle interesting instances. If the timeout task is modeled explicitly, then many *alive* messages must be send every computation step, which results in an overly complex model. Using properties  $A_1$  and  $A_2$ , however, the explicit sending of *alive* messages can be abstracted away.

### 5.1 Modeling the Algorithm

Figure 7 shows the UPPAAL template of the behavior of the algorithm. This template is parameterized with two constants, namely its unique identifier  $id$ , and a boolean *mayFail* which indicates whether this process may fail<sup>5</sup>.

Similar to the model of the timeout task, a process first waits  $c_2$  time units in its initial location. Then, it non-deterministically chooses an input value on an edge to *wait*. The global clock  $t$  is used to measure the running time of the

<sup>5</sup>Again, this is a trick that exploits the symmetry of processes to reduce the reachable state space.

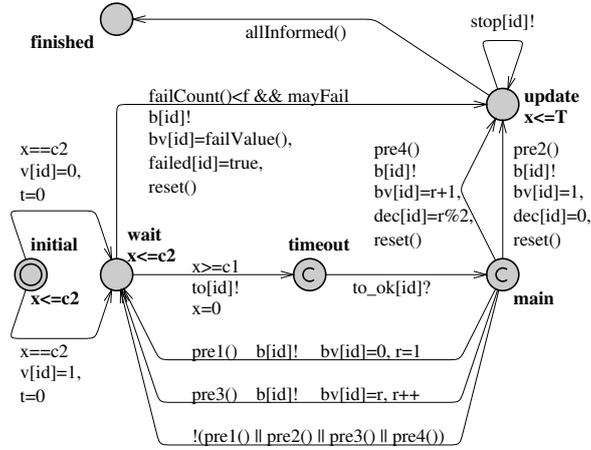


Figure 7: The process template.

algorithm. Then it either starts a computation step or fails. A computation step first activates the timeout automaton of the process, which is described below, on the edge to *timeout*. When the timeout automaton finishes (it may have updated the *halted* set), the transition to *main* is taken. Then there are five possibilities: one of the one of the four preconditions of the main task transitions is satisfied (note that they are all mutually exclusive), or none of them is satisfied. In the first case, the specified actions are taken, and in the second case nothing is done. The committed locations (those with a “C” inside) specify that a computation step is atomic and that it takes no time (if a committed location is active, then no delay is allowed and the next action transition must involve a committed component). Note that broadcasting the message  $(m, i)$  is achieved by assigning  $m$  to  $bv[id]$  on an edge with a  $b[id]$ -synchronization. Figure 8 shows the functions that implement the preconditions of the four transitions of the main task (see also Figure 2).

```

bool pre1 ()
{
    return r==0 && v[id]==1;
}

bool pre2 ()
{
    return r==0 && v[id]==0;
}

bool pre3 ()
{
    int j;
    if (r<=0)
        return false;
    for (j=0; j<N; j++)
        if (buff[id][r][j])
            return true;
    return false;
}

bool pre4 ()
{
    int j;
    if (r<=0 || pre3())
        return false;
    for (j=0; j<N; j++)
        if (!halted[id][j] &&
            !buff[id][r-1][j])
            return false;
    return true;
}

```

Figure 8: The preconditions for the four transitions of the main task.

A failure is modeled by the edge from *wait* to *update*. This edge is only enabled if less than  $f$  failures already have occurred. The *failValue()* function computes the value that would have been broadcast during the next computation step.

In location *update* the process has halted either by decision or by failure. It can stay there for a maximum of  $T$  time units and it provides a *stop[id]*-synchronization. This is used for the abstraction of the timeout task, which is explained below. When all other processes have been informed that this process has halted (*allInformed()* returns *true*), then the transition to location *finished* is enabled.

Similar to the model of the timeout task, the broadcasts are modeled by instances of the broadcast template which is shown in Figure 9.

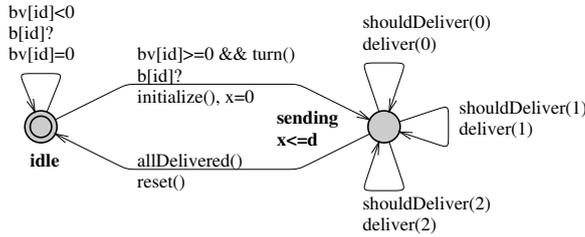


Figure 9: The broadcast template.

The template is parameterized with two constants, namely  $id$ , the identifier of the process automaton this broadcast automaton belongs to, and  $bid$ , an identifier that is unique among the other broadcast automata of process automaton  $id$ . Note that this template is tailored to a model with  $n = 3$  (there are three self-loops from location *sending*) for reasons of efficiency.

The broadcast automaton is started with a  $b[id]$ -synchronization. If the value of  $bv[id]$  is smaller than zero, then nothing is done (this is convenient for modeling in the process template). If the value is larger than or equal to zero and *turn()* returns *true*<sup>6</sup>, then this broadcast automaton can start delivering the message that has been passed to it in  $bv[id]$ . The *shouldDeliver()* and *allDelivered()* functions ensure that it delivers all messages on time, but only if necessary. I.e., it is not useful to deliver a message to a process that already has halted, since that message is never used; it only increases the reachable state space.

Each process automaton has a separate timeout automaton that has two functions. First, it is activated at the beginning of each computation step of the process it belongs to in order to update the *halted* set of the process. Second, it serves as a test automaton to ensure that the process it belongs to is output enabled<sup>7</sup>. The timeout template is shown in Figure 10. It has one parameter, namely the constant  $id$ , which refers to the process it belongs to.

<sup>6</sup>Similarly as in the model of the timeout task in the previous section, the guard *turn()* exploits the symmetry between the broadcast automata of a single process to reduce reachable the state space.

<sup>7</sup>In this model, the number of necessary broadcast automata is no longer easily to determine. Therefore, an explicit check is useful.

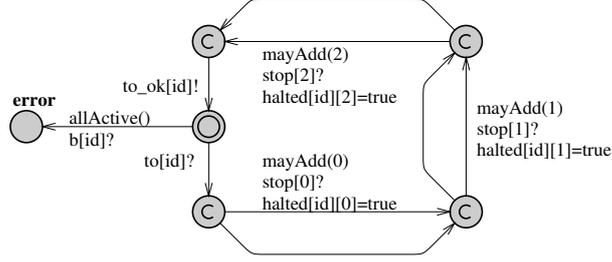


Figure 10: The timeout template.

The timeout template is tailored to  $n = 3$  for reasons of efficiency. When it is activated, it checks for each process  $j$  whether it may add it to the *halted* set, and if so, it non-deterministically chooses whether to add it or not. Here properties  $A_1$  and  $A_2$  of the timeout task come in. The function *mayAdd()* checks for a given process  $j$  whether all messages from  $j$  to this process have been delivered. If not, then it may not add  $j$  to *halted* (property  $A_1$ ). Furthermore, the synchronization over the channel *stop[j]* must be enabled. In Figure 7 can be seen that this is only the case for the  $T$  time units after  $j$  has halted (property  $A_2$ ). But if this process has not added  $j$  to *halted* by that time, then  $j$  cannot proceed to location *finished* (in that case *allInformed()* returns *false*), with a time deadlock as result. This is exactly the case when  $T - p_i.x < c_1 - p_j.x$  for processes  $i$  and  $j$ .

The second function of the timeout template is implemented by the edge to the *error* location. This location is reachable if the process wants to broadcast and all its broadcast automata are active already. In a correct model, the *error* location therefore is not reachable.

## 5.2 Verifying the Algorithm

Properties  $M_1$ – $M_3$  are translated as follows (where  $U$  is the upper bound on the running time of the protocol as specified before).

$$\mathbf{AG} \left( \forall_{i,j} dec_i \geq 0 \wedge dec_j \geq 0 \longrightarrow dec_i = dec_j \right) \quad (3)$$

$$\mathbf{AG} \left( \forall_i dec_i \geq 0 \longrightarrow \exists_j dec_i = v_j \right) \quad (4)$$

$$\mathbf{AG} \left( (\exists_i p_i.wait) \longrightarrow t \leq U \right) \quad (5)$$

The following properties are health checks to ensure that (i) the processes are output enabled, and (ii) the only deadlocks in the model are those that are expected.

$$\mathbf{AG} \left( \neg \exists_i T_i.error \right) \quad (6)$$

$$\mathbf{AG} \left( \begin{array}{l} deadlock \longrightarrow (\forall_i p_i.finished \vee \\ \exists_{i,j} p_j.x - p_i.x > T - c_1) \end{array} \right) \quad (7)$$

The properties (3)–(6) have been verified (using the convex-hull approximation of UPPAAL with a breadth-first search order) for the following parameter values:

- $n = 3$ ,  $f \in \{0, 1\}$ ,  $c_1 = 1$ ,  $c_2 = 1$ , and  $d \in \{0, 1, 2, 3, 5, 10\}$ ,
- $n = 3$ ,  $f \in \{0, 1\}$ ,  $c_1 = 1$ ,  $c_2 = 2$ , and  $d \in \{0, 1, 2, 3, 5, 10\}$ , and
- $n = 3$ ,  $f \in \{0, 1\}$ ,  $c_1 = 9$ ,  $c_2 = 10$ , and  $d \in \{5, 9, 10, 11, 15, 20, 50\}$ .

The verification of any instance needs at most 1.5 GB of memory and at most 30 minutes of time on a regular desktop computer. Property (7) has been verified for a subset of the above parameter values, namely for the models with the three smallest values for  $d$  in each item. This property is more difficult to model check since it involves the *deadlock* state property, which disables UPPAAL’s LU-abstraction algorithm [5] (a less efficient one is used instead), and which is computationally quite complex due to the symbolic representation of states.

## 6 Conclusions

Despite the fact that model checkers are in general quite easy to use (in the sense that their learning curve is not so steep as for instance the one of theorem provers), making a *good* model still is difficult. The algorithm that has been analyzed in this paper, for instance, can quit easily be modeled “literally”. The message complexity then is huge due to the many broadcasts of *alive* messages, with the result that model checking interesting instances becomes impossible. This has been solved by a non-trivial abstraction of the timeout task. Ideally of course, model checkers can even handle such “naive” models. Fortunately, much research still is aimed at improving these tools. For instance, the UPPAAL model checker is getting more and more mature, both w.r.t. usability as efficiency. An example of the former is the recent addition of a C-like language. This makes the modeling of the agreement protocol much easier, and makes the model more efficient. A loop over an array, as for instance used in the *pre3()* and *pre4()* functions shown in Figure 8, can now be encoded with a C-like function instead of using a cycle of committed locations and/or an auxiliary variable. This saves the allocation and deallocation of intermediate states and possibly a state variable. Other examples of efficiency improvements of UPPAAL are enhancements like symmetry reduction [15] and the sweep line method [12], which are planned to be added to UPPAAL soon. Especially symmetry reduction would greatly benefit distributed systems, which often exhibit full symmetry. Furthermore, current research also focuses on distributing UPPAAL, which may even give a super-linear speed-up [7, 4].

Summarizing, model checking tools become more and more applicable to the design of distributed systems due to the steady increase of their usability and efficiency. They may provide valuable early feedback on subtle design errors and hint at system invariants that can subsequently be used in the general correctness proof that is either constructed by hand or by use of a theorem prover. The present paper has demonstrated that model checking now even is feasible for some small yet interesting instances of an agreement algorithm which thus far was considered

out of reach for model checking technology. This result shows that the class of partially synchronous systems is within reach of the current state-of-the-art model checking tools.

*Acknowledgements.* The author thanks Frits Vaandrager and Jozef Hooman for valuable discussions and comments on earlier versions of the present paper.

## References

- [1] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking in dense real time. *Information and Computation*, 104:2–34, 1993.
- [2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
- [3] H. Attiya, C. Dwork, N. A. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, 1994.
- [4] G. Behrmann. Distributed reachability analysis in timed automata. *Software Tools for Technology Transfer*, November 2003. Issue: Online First.
- [5] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In *TACAS'04*, number 2988 in LNCS, pages 312–326. Springer-Verlag, 2004.
- [6] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In *SFM'04*, volume 3185 of LNCS, pages 200–236. Springer, 2004.
- [7] G. Behrmann, T. Hune, and F. W. Vaandrager. Distributed timed model checking – how the search order matters. In *CAV'00*, number 1855 in LNCS, pages 216–231. Springer-Verlag, 2000.
- [8] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, number 3098 in LNCS, pages 87–124. Springer-Verlag, 2004.
- [9] M. Bodlaender, J. Guidi, and L. Heerink. Enhancing discovery with liveness. In *CCNC'04*. IEEE Computer Society Press, January 2004.
- [10] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses, 2004. <http://www.ietf.org/internet-drafts/draft-ietf-zeroconf-ipv4-linklocal-17.txt>.
- [11] D. Chklier, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *TACAS'03*, number 2619 in LNCS, pages 113–127. Springer-Verlag, 2003.
- [12] A. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *TACAS'01*, number 2031 in LNCS, pages 450–464. Springer-Verlag, April 2001.
- [13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.
- [14] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 197–212. Springer-Verlag, 1989.

- [15] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding symmetry reduction to Uppaal. In *FORMATS'03*, number 2791 in LNCS, pages 46–59. Springer–Verlag, 2004.
- [16] J.-P. Katoen, H. Bohnenkamp, R. Klaren, and H. Hermanns. Embedded software analysis with MOTOR. In *SFM'04*, number 3185 in LNCS, pages 268–293. Springer–Verlag, 2004.
- [17] D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager. A framework for modelling timed systems with restricted hybrid automata. In *RTSS'03*, pages 166–177. IEEE Computer Society Press, 2003. A full version is available as MIT Technical Report MIT/LCS/TR-917.
- [18] I. A. van Langevelde, J. M. T. Romijn, and N. Goga. Founding firewire bridges through promela prototyping. In *IPDPS'03, FMPPTA'03*. IEEE Computer Society Press, 2003.
- [19] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
- [20] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC'87*, pages 137–151, 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.
- [21] N. A. Lynch and M. R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, 1989.
- [22] A. J. Mooij, N. Goga, W. Wesselink, and D. Bosnacki. An analysis of medical device communication standard IEEE 1073.2. In *Proceedings of the Second IASTED International Conference on Communication Systems and Networks*. ACTA Press, 2003.
- [23] A. S. Tanenbaum. *Computer Networks*. Prentice–Hall, 1996.
- [24] 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. To appear.