Model Checking the Time to Reach Agreement*

Martijn Hendriks

Institute for Computing and Information Sciences,
Radboud University Nijmegen,
Toernooiveld 1, 6525 ED 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 (systems which have only partial information about timing, e.g., only an upper bound on the message delay). The past has shown that verification of these systems by model checking usually is very difficult. The present paper demonstrates that an agreement algorithm of Attiya et al, which falls into a – for model checkers – particularly problematic subclass of partially synchronous distributed systems, 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. Although existing techniques are used, this is an interesting case study in its own right that adds to the existing body of experience. Furthermore, the agreement algorithm has not been formally verified before to the author's knowledge.

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 [1, 2, 3]. 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 [4]. The approach requires the construction of a 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

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

algorithm can be verified for 3 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.

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 [3]. The timed automata framework of Alur and Dill [5] is a natural choice for the specification of partially synchronous systems (as is the Timed I/O-automaton framework [6], which, however, does not support model checking). Verification of the above mentioned subclass of "difficult" partially synchronous systems by model checking, however, 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 [7, 8], a protocol to monitor the presence of network nodes [9, 10, 11], and the ZeroConf protocol whose purpose is to dynamically configure IPv4 link-local addresses [12, 13]. Furthermore, the agreement algorithm described in [14] (see also Chapter 25 of [3]) 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. The main contribution consists of the formal verification of some non-trivial instances of the algorithm, which has not been done before to the author's knowledge. Although standard modeling and verification techniques are used, the case study is interesting in its own right, and increases the existing body of case-study experience. Independently of the present work, Leslie Lamport has also analyzed a distributed algorithm that falls into the class of difficult partially synchronous systems as defined above [15].

The remainder of this paper is structured as follows. The timed automaton framework and the Uppaal model checker are very briefly introduced in Section 2. Section 3 then presents an informal description of the distributed algorithm of [14], 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/`. Note that the Uppaal development version 3.5.7 has been used.

2 Timed Automata

This section provides a very brief overview of timed automata and their semantics, and of the Uppaal tool, which is a model checker for timed automata. The reader is referred to [16] and [17] for more details.

Timed automata are finite automata that are extended with real valued clock variables [5]. Let X be a set of clock variables, then the set $\Phi(X)$ of clock constraints ϕ is defined by the grammar $\phi := x \sim c \,|\, \phi_1 \wedge \phi_2$, where $x \in X$, $c \in \mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$. A clock interpretation ν 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 ν for X satisfies a clock constraint ϕ over X, denoted by $\nu \models \phi$, if and only if ϕ evaluates to *true* with the values for the clocks given by ν. 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 ν over the rest of the clocks. We let $\Gamma(X)$ denote the set of all clock interpretations for X.

A timed automaton then is 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, Σ 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 ϕ 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, ν), where $l \in L$, and $\nu \in \Gamma(X)$ such that $\nu \models I(l)$. The initial state is (l^0, ν^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 *δ-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 [18, 19]. This enables the application of finite state model checking techniques as implemented, for instance, by the Uppaal tool.

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 ϕ is reachable can be formalized as $\mathbf{EF}(\phi)$. The question whether ϕ holds for all reachable states is formalized as $\mathbf{AG}(\phi)$. If a reachability property holds or an invariant property does not hold, then UPPAAL can provide 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 [14] (see also Chapter 25 of [3]). A system of n processes, denoted by $p_1, ..., p_n$, 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)[1]. 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 can get neither 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.

3.1 Description of the Timeout Task

The goal of the timeout task is to maintain the running state of all other processes. To this end, every process p_j broadcasts an $(alive, j)$ message in every computation step. If process p_i has run for sufficiently many computation steps without receiving an $(alive, j)$ message, then it assumes that p_j halted either by decision or by failure[2]. 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 $halted \subseteq \{1, ..., n\}$, initially $\emptyset$, and for every

[1] This is required to avoid trivial solutions in which every process always decides for some predetermined constant value.

[2] 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], further analyzed in [10], and improved in [11].

```
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
```

Fig. 1. The timeout task for process p_i

$j \in \{1, ..., n\}$ a counter *counter*(j), initially set to -1. Additionally, every process has a message buffer *buff* (a set), initially $\emptyset$. Two properties of the timeout task have been proven in [14].

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 $T = D + c_2 \cdot (\lfloor \frac{D}{c_1} \rfloor + 1)$.

These two properties are used in [14] for the correctness proof of the complete algorithm. In this paper, these two properties are first mechanically verified for a number of instances of the algorithm. Consequently, they are used to make an abstract model of the complete algorithm in Section 5.

3.2 Description of the Main Task

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 round counter r, initially zero. The input values are assumed to be either zero or one for simplicity[3].

Each process tries to decide in each round. Note that a process may decide for 0 only in even rounds, and for 1 only in odd rounds. Furthermore, if a process fails to decide in round r, then it broadcasts r before going to round $r + 1$. On the other hand, if a process decides in round r, it broadcasts $r+1$ before halting. In order for a process to decide in a round $r \geq 1$, it ensures that it has received the message $r - 1$ from all non-halted processes, and no message r from any process. Three main results that are obtained in [14] are the following.

[3] An extension to an arbitrary input domain is discussed in [14].

```
**Precondition:**
    $r = 0 \wedge v_i = 1$
**Effect:**
    **broadcast**$((0,i))$
    $r := 1$
```

```
**Precondition:**
    $r \geq 1 \wedge \exists_j \ (r,j) \in \mathit{buff}$
**Effect:**
    **broadcast**$((r,i))$
    $r := r + 1$
```

```
**Precondition:**
    $r = 0 \wedge v_i = 0$
**Effect:**
    **broadcast**$((1,i))$
    **decide**$(0)$
```

```
**Precondition:**
    $r \geq 1 \wedge \forall_{j \notin \mathit{halted}} \ (r-1,j) \in \mathit{buff} \ \wedge$
    $\neg\exists_j \ (r,j) \in \mathit{buff}$
**Effect:**
    **broadcast**$((r+1,i))$
    **decide**$(r \bmod 2)$
```

Fig. 2. The main task for process p_i

M_1 (Agreement, Lemma 5.9 of [14]). No two processes decide on different values.
M_2 (Validity, Lemma 5.10 of [14]). If process p_i decides on n, then $n = v_j$ for some process j.
M_3 (Termination, Theorem 5.1 of [14]). The upper bound on the time to reach agreement equals $(2f-1)D + \mathit{max}\ \{T, 3D\}$.

These results are mechanically verified in Section 5 for a number of non-trivial instances of the algorithm.

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*).

Figure 3 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

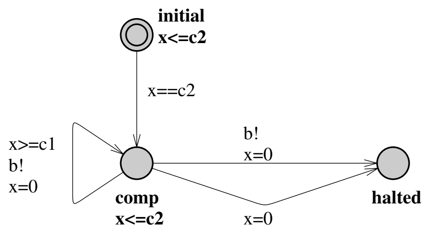

Fig. 3. The *Process* automaton

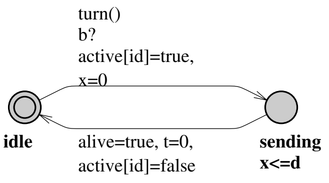

Fig. 4. The broadcast template

to *halted* (a decision transition that blocks the timeout task)[4]. Note that x is reset on every edge to *halted* for verification purposes.

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 4. 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 partially exploiting the symmetry among the broadcast instances[5]. 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 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 procedure *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 [14] that "every process has a computation or failure event

[4] 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.

[5] A next release of UPPAAL will hopefully support symmetry reduction, which can automatically exploit the symmetry among broadcast automata [20].

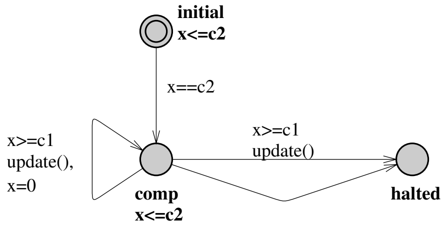

Fig. 5. The *Observer* automaton

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

Fig. 6. The *update()* function

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.)

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):

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

The state property $\forall_i\ b_i.\mathit{idle} \wedge t > 0$ 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:

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

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[6]:

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

Each of the above instances could be verified within 5 minutes using at most 25 MB of memory.

[6] A 3.4 GHz Pentium 4 machine with 2 GB of main memory running Fedora Core 4 has been used for all measurements. The tool *memtime* (available via the UPPAAL website `http://www.uppaal.com/`) has been used to measure the time and memory consumption.

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 large to handle interesting instances. If the timeout task is modeled explicitly, then many *alive* messages must be sent 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[7].

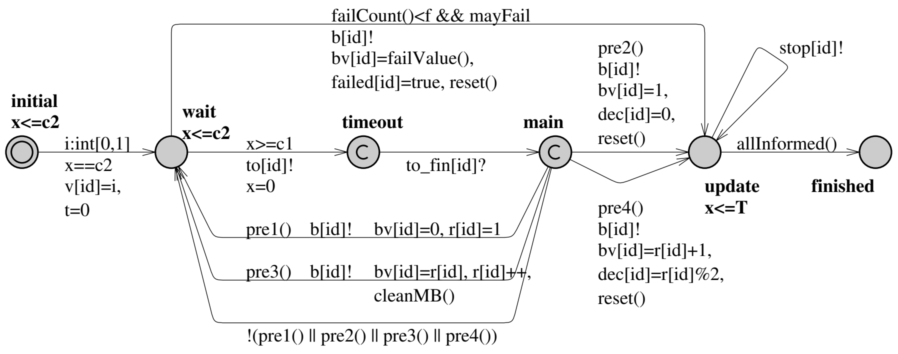

Fig. 7. The process template

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 in $\{0, 1\}$ on the edge to *wait*. The global clock t is used to measure the running time of the algorithm, and is only reset on this edge. 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 edge to *main* is taken. Then there are five possibilities: 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"

[7] Again, this is a trick that exploits the symmetry of processes to reduce the reachable state space.

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[id]==0 && v[id]==1;
}

bool pre3 ()
{
  if (r[id]<=0)
    return false;
  for (j:pid_t)
    if (buff[id][r[id]][j])
      return true;
  return false;
}
```

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

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

Fig. 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 fewer 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.

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 au-

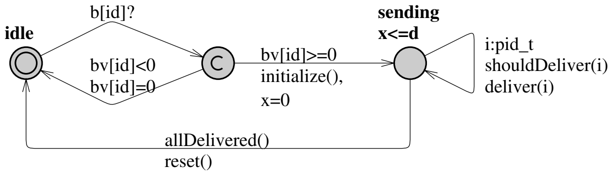

Fig. 9. The broadcast template

tomaton *id*. The broadcast automaton is started – if it is its turn[8] – 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). In location *sending* it starts 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[9]. The timeout template is shown in Figure 10. It has one parameter, namely the constant *id*, which refers to the process it belongs to.

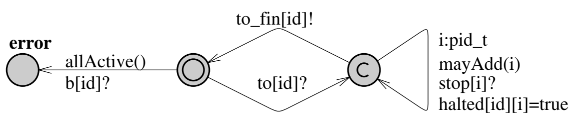

Fig. 10. The timeout template

When a timeout process is activated, it non-deterministically picks a subset of processes that have halted and adds them to the *halted* set. 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. We believe that this abstraction of the timeout task is safe, i.e., every admissible computation path in the original model of [14] can be mapped to an equivalent path in the UPPAAL model.

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.

[8] Similarly as in the model of the timeout task in the previous section, the guard *turn()* partially exploits the symmetry between the broadcast automata of a single process to reduce the reachable state space.

[9] In this model, the number of necessary broadcast automata is no longer easily to determine. Therefore, an explicit check is useful.

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).

$$\textit{Agreement:}\ \mathbf{AG}\Big(\forall_{i,j}\ dec_i \geq 0 \wedge dec_j \geq 0 \longrightarrow dec_i = dec_j\Big) \tag{3}$$

$$\textit{Validity:}\ \mathbf{AG}\Big(\forall_i\ dec_i \geq 0 \longrightarrow \exists_j\ dec_i = v_j\Big) \tag{4}$$

$$\textit{Termination:}\ \mathbf{AG}\Big((\exists_i\ p_i.wait) \longrightarrow t \leq U\Big) \tag{5}$$

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.

$$\mathbf{AG}\Big(\neg\exists_i\ T_i.error\Big) \tag{6}$$

$$\mathbf{AG}\ \big(\ deadlock \longrightarrow (\forall_i\ p_i.finished\ \vee \exists_{i,j}\ p_j.x - p_i.x > T - c_1)\big) \tag{7}$$

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[6]:

- $n = 3$, $f \in \{0, 1\}$, $c_1 = 1$, $c_2 = 1$, 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 $d \in \{5, 9 - 11, 15, 20, 50, 100\}$.

Each of the above instances could be verified within 11 minutes using at most 1014 MB of memory. 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 the convex-hull approximation is not useful and it involves the *deadlock* state property, which disables Uppaal's LU-abstraction algorithm [21] (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 can easily be modeled "literally". The message complexity then, however, 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 [20] and the sweep line method [22], which are planned to be added to UPPAAL soon. Especially symmetry reduction would greatly benefit distributed systems, which often exhibit full symmetry. Furthermore, recent research also focuses on distributing UPPAAL, which may even give a super-linear speed-up [23, 24].

It seems that the class of partially synchronous systems, which is notoriously difficult from the perspective of model checking, now slowly comes within reach of present model checking tools. Therefore, these tools have the potential to play a valuable role in the design of these systems. 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.

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

References

1. Lynch, N.A., Tuttle, M.R.: An introduction to Input/Output automata. CWI-Quarterly **2** (1989) 219–246
2. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC'87. (1987) 137–151 A full version is available as MIT Technical Report MIT/LCS/TR-387.
3. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
5. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
6. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: A framework for modelling timed systems with restricted hybrid automata. In: RTSS'03, IEEE Computer Society Press (2003) 166–177 A full version is available as MIT Technical Report MIT/LCS/TR-917.
7. Tanenbaum, A.S.: Computer Networks. Prentice–Hall (1996)
8. Chkliaev, D., Hooman, J., de Vink, E.: Verification and improvement of the sliding window protocol. In: TACAS'03. Number 2619 in LNCS, Springer–Verlag (2003) 113–127
9. Bodlaender, M., Guidi, J., Heerink, L.: Enhancing discovery with liveness. In: CCNC'04, IEEE Computer Society Press (2004)
10. Katoen, J.P., Bohnenkamp, H., Klaren, R., Hermanns, H.: Embedded software analysis with MOTOR. In: SFM'04. Number 3185 in LNCS, Springer–Verlag (2004) 268–293
11. Bohnenkamp, H., Gorter, J., Guidi, J., Katoen, J.P.: Are you still there? A lightweigth algorithm to monitor node absence in self-configuring networks. In: Proceedings of DSN 2005. (2005)

12. Cheshire, S., Aboba, B., Guttman, E.: Dynamic configuration of IPv4 link-local addresses (2004) *http://www.ietf.org/internet-drafts/draft-ietf-zeroconf-ipv4-linklocal-17.txt.*
13. Zhang, M., Vaandrager, F.: Analysis of a protocol for dynamic configuration of IPv4 link local addresses using Uppaal. Technical report, NIII, Radboud University Nijmegen (2005) To appear.
14. Attiya, H., Dwork, C., Lynch, N.A., Stockmeyer, L.: Bounds on the time to reach agreement in the presence of timing uncertainty. Journal of the ACM **41** (1994) 122–152
15. Lamport, L.: Real-time model checking is really simple. In Borrione, D., Paul, W., eds.: CHARME 2005. LNCS, Springer (2005) To appear.
16. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: SFM'04. Volume 3185 of LNCS., Springer (2004) 200–236
17. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In Desel, J., Reisig, W., Rozenberg, G., eds.: Lectures on Concurrency and Petri Nets: Advances in Petri Nets. Number 3098 in LNCS, Springer–Verlag (2004) 87–124
18. Alur, R., Courcoubetis, C., Dill, D.L.: Model checking in dense real time. Information and Computation **104** (1993) 2–34
19. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Automatic Verification Methods for Finite State Systems. Number 407 in LNCS, Springer–Verlag (1989) 197–212
20. Hendriks, M., Behrmann, G., Larsen, K.G., Niebert, P., Vaandrager, F.W.: Adding symmetry reduction to Uppaal. In: FORMATS'03. Number 2791 in LNCS, Springer–Verlag (2004) 46–59
21. Behrmann, G., Bouyer, P., Larsen, K.G., Pelańek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: TACAS'04. Number 2988 in LNCS, Springer–Verlag (2004) 312–326
22. Christensen, A., Kristensen, L.M., Mailund, T.: A sweep-line method for state space exploration. In: TACAS'01. Number 2031 in LNCS, Springer–Verlag (2001) 450–464
23. Behrmann, G., Hune, T., Vaandrager, F.W.: Distributed timed model checking – how the search order matters. In: CAV'00. Number 1855 in LNCS, Springer–Verlag (2000) 216–231
24. Behrmann, G.: Distributed reachability analysis in timed automata. Software Tools for Technology Transfer **7** (2005) 19–30