Formal Specification and Analysis of Accelerated Heartbeat Protocols

Muhammad Atif MohammadReza Mousavi
Department of Computer Science,
Eindhoven University of Technology
{ M.Atif, M.R.Mousavi}@tue.nl

Abstract

We present a formal analysis of all different variations of accelerated heartbeat protocols presented in [M.G. Gouda and T.M. McGuire, Accelerated Heartbeat Protocols, Proc. of ICDCS'98]. We formalize the specification of the protocols both in a process-algebraic and in an automata-theoretic formalism. Then, we formulate some natural functional requirements on the above-mentioned protocols and formalize these requirements. Using model-checking techniques, we verify these requirement on each and every version. We report counter-examples witnessing that the formulated requirements are not satisfied. We propose fixes for different version of the protocol and model check the fixed versions; the model checking results indicate that the fixed versions indeed satisfy the requirements.

1 Introduction

Heartbeat protocols are used as the underlying synchronization mechanism for many other distributed protocols [GM00, HSC95, TSLC02, Vog96, WGZC05]. The basic idea behind a heartbeat protocol is that once a participating process or a communication channel crashes, other processes become aware of this fact and become inactive within a certain interval. To this end, processes periodically exchange simple messages, called *heartbeats*, to inform each other about their liveness. If an expected heartbeat is not received after a specific time, it is assumed that either the respective process has failed or the communication medium is down. After a number of periods without any response, the expecting processes eventually become inactive, thus guaranteeing timely inactivation of all participants after a process or channel crash.

In [GM98], several variations of heartbeat protocols are presented. These protocols aim at achieving the above-mentioned goal while reducing the overhead, i.e., the rate of heartbeat transmissions. Moreover, they try to minimize the detection delay (the interval between the crash and the deactivation of all processes) and maximize reliability (minimizing the probability of inactivation due to lost heartbeats).

We formally model and analyze all different versions of heartbeat protocols presented in [GM98]. To this end, we give a formal specification of these protocols in two formalisms: the process algebra mCRL2 [GMvWU06] and the timed-automata language of UPPAAL [BDL04]. Note that both process-algebraic and automata-theoretic models are complete models of the protocols and can be independently used to present the same results. Then, we specify basic properties about the safety and liveness of the protocols, namely that upon a crash, all processes will eventually be deactivated within a certain period of time (to be specified precisely by the protocol specification) and if no process crashes and no message is lost or delayed (beyond its allowed limit), then no process will decide to deactivate (i.e., suspect any crash). We verify these, rather basic, requirements on the protocols given in [GM98]. For the process algebraic specification, we specify the requirements using a combination of monitor processes and modal μ-calculus formulae and use the Caesar/Aldebaran tool-set [GMLS07] to model-check the specified properties on the

formal specification of the protocols. For the automata-theoretic specifications, we use a combination of monitor timed-automata and reachability properties and use UPPAAL to model-check them. To our surprise, for each of the protocols we found situations where one or both of the above properties are not satisfied. In [MG04], slightly modified versions of some of the protocols in [GM98] are presented. We have also analyzed the modified versions and briefly discuss the results in the remainder of this paper.

Structure of the paper. Heartbeat protocols are presented informally in Section 2. In Sections 3 and 4, respectively, we give an overview of the process-algebraic and the automata-theoretic specification of the protocol. Section 5 is devoted to the specification of requirements and their analysis. In Section 6, the discovered counter-examples are discussed and some fixes for the protocols are proposed. The fixed versions of the protocol are then model-checked and shown to be correct. The paper is concluded in Section 7.

2 Accelerated heartbeat protocols

In this section, we briefly present the following four different types of accelerated heartbeat protocols introduced in [GM98].

1. The binary heartbeat protocol.
2. The static heartbeat protocol.
3. The expanding heartbeat protocol.
4. The dynamic heartbeat protocol.

All the protocols, to be presented in the remainder of this section, have the following basic assumptions in common.

1. Every process is active in the beginning.
2. Any active process can become inactive anytime (due to a crash) but cannot become active again (recover) afterwards. (We call the crash of a process its *voluntary* inactivation, as opposed to non-voluntary inactivation, which is caused by the protocol.)
3. Every sent message will be received provided that the communication medium is up. In particular, messages sent to crashed processes will be received but will be given no reply. If a message is to be delivered (the channel is up), it is delivered within a certain period of time; the maximum round-trip delay of channels is bound by the constant $tmin$.

2.1 The binary heartbeat protocol

In this protocol, only two processes participate in exchanging their heartbeats in a round-based fashion. Let p[0] and p[1] be the processes and $tmax$ and $tmin$ be the maximum and minimum waiting time, respectively, for each round. Let t be the waiting time of p[0] for each round such that $tmin \leq t \leq tmax$. (Note that $tmin$ is the same constant as the upper bound on the round-trip channel delays.) The process p[0] iteratively follows the steps given below to run the protocol:

1. It waits for a period of length t, where t is initially set to $tmax$.
2. It sends a heartbeat message to p[1].
3. For the next round, the value for t is $tmax$ if p[0] has received the heartbeat from p[1] in the current round, or otherwise t becomes $t/2$. However, if the new value of t is less than $tmin$, p[0] itself becomes inactive (non-voluntarily).

To respond to p[0], p[1] performs the following three steps.

1. Receives a heartbeat from p[0].
2. Sends its heartbeat.
3. If it does not receive a heartbeat from p[0] for a period of length $3tmax - tmin$, it becomes inactive (non-voluntarily).

In [MG04], a slightly modified version of the binary (and static) heartbeat protocols are presented, in which p[0] does not wait initially but starts off by sending its heartbeat to p[1]. In the remainder of this paper, we refer to this version as the *revised binary heartbeat protocol*.

Also, a modified version of binary heartbeat protocol, called *two-phase heartbeat protocol*, is presented in [GM98]. In the two-phase heartbeat protocol the value of t is reduced immediately to $tmin$ if p[0] does not receive a heartbeat from p[1] in the previous round. Otherwise, the specification of the two-phase heartbeat protocol is identical to its binary counterpart.

2.2 The static heartbeat protocol

The static heartbeat protocol extends the binary heartbeat protocol by allowing for a fixed number of participating processes. In this protocol, p[0] broadcasts its heartbeat to n processes, where the value of n is fixed and a priori known to p[0]. Process p[0] exchanges heartbeat messages periodically by executing the binary heartbeat protocol with every p[i], where $1 \leq i \leq n$. (Note that all heartbeat exchanges are with p[0] and other processes do not exchange their heartbeats among themselves.) All of the processes in the network commonly use the values of $tmax$ and $tmin$ (introduced in Section 2.1). Process p[0] maintains a list of type Boolean to record the response of every process with respect to its sent heartbeats. It assigns the value *true*, if it receives a heartbeat within t time units, or otherwise *false* to the respective process. Process p[0] also maintains a list of time periods tm (initialized with $tmax$ as the value of all cells), indicating the waiting time for each process. This list is changed after each round according to the received heartbeat from respective process(es), using the same procedure described for t in Section 2.1, i.e., after each round, if p[0] receives a heartbeat from p[i] within t time units, then the value of $tm[i]$ is set to $tmax$, otherwise, it is set to $tm[i]/2$. In each round, the waiting time t of p[0] is defined by $min(tm[1..n])$.

2.3 The expanding heartbeat protocol

The expanding heartbeat protocol extends the static heartbeat protocol in that the participating processes may join the protocol gradually in the course of protocol execution. At the start of this protocol, p[0] is the only process and later any number of other processes can join by sending their heartbeats. The new process recognizes that it has joined the protocol when it receives the heartbeat from p[0]; otherwise, it continues sending its heartbeat every $tmin$ time units until the limit $3tmax - tmin$ is reached, upon which it decides that p[0] or the communication channel has crashed and goes to the (non-voluntarily) inactivated state. Process p[0] maintains the list jnd of joined processes and executes the static heartbeat protocol with them. The response time and waiting time of each process and each round, respectively, are computed exactly as in the static heartbeat protocol, discussed in Section 2.2.

2.4 The dynamic heartbeat protocol

This version is the most flexible one as compared to the other protocols introduced in [GM98]. Each process can join and then leave (permanently) at will. To encode join and leave messages in this protocol, heartbeats are parameterized with a boolean parameter. For joining or remaining in protocol, heartbeats carry *true* and for leaving, they carry *false* as the data parameter. If a process p[i] decides to join the protocol, it keeps on sending its heartbeat with parameter *true* every $tmin$ units of time. Receiving a heartbeat with parameter *true* from p[0] indicates that p[i] has actually joined the protocol. Again if p[i] does not receive a heartbeat with parameter

true within a period of $3tmax - tmin$, p[i] will assume a crash in the nodes or channels and will inactivate itself non-voluntarily. When joined the protocol, the specification of p[i] is identical to the expanding version, except for the fact that it can leave the protocol by sending heartbeats with parameter *false*. Again, p[0] acknowledges this by sending a heartbeat with the same parameter. It is important to differentiate between leaving the protocol and crashing (voluntarily becoming inactive). The former will not affect the other participants at all while the latter will cause the inactivation of every process in the network.

3 Formal specification in mCRL2

3.1 Introduction

In this section, first we present a general overview of our formal specification in the process algebraic formalism mCRL2 and then the notational aspects of the formalism used in our specification of the accelerated heartbeat protocols. Our formal specification in mCRL2 comprises the following aspects.

Data types. The first part of our formal specification is dedicated to formalizing the data types used in the specification of the protocol. This part involves no novelty, as data types such as booleans, integers and lists are already built in the mCRL2 syntax and we only need to compose them in order to obtain more complex data structures. Also, we defined straightforward operations on these data types to check and update data values in different data structures.

Main processes. The participants of the protocols are modelled as a choice among a number of sequential processes, each of which are triggered by a certain event. For example, the specification of process p[0] in the binary heartbeat protocol comprises the following choices:

1. crashing,
2. receiving a heartbeat from p[1],
3. receiving a time-out (t time units after sending the heartbeat), checking whether a heartbeat is received from from p[1], and either becoming inactivated non-voluntarily or adjusting the next waiting time accordingly,
4. passing one time unit.

The structure of p[1] is very similar to that of p[0]; it only replies to the heartbeat received from p[0] right away and has a time-out on $3tmax - tmin$ resulting in immediate non-voluntary inactivation.

The structure of the main processes in the other three versions are quite similar, except that more choices are added to the specification (e.g., for joining and leaving the protocol) and more details are added to each sequential process (e.g., calculating the minimum of the list of waiting times).

Channels. Channels are simple processes, which receive a message from one side and non-deterministically decide to lose it or deliver it to the other side. We explain below how the upper-bound on the round-trip delay of channels is enforced.

Timing. Timing constraints play a crucial role in the heartbeat protocols; they are both present in the protocol specification as well as in the correctness requirements. In the current version of the mCRL2 toolset, there is limited support for the analysis of timed specification. Hence, we have set up auxiliary processes which act as clocks and watchdogs for the actual processes participating in the protocols. There is an underlying clock synchronization mechanism, which enforces different clocks to proceed at the same speed. This is achieved by a multi-party synchronization among

clocks (using the so called multi-actions in mCRL2). Moreover, timeouts are implemented as watchdog processes that start ticking when they receive a message corresponding to their triggering event (e.g., sending a heartbeat by p[0] for the watchdog taking care of timeout at t in p[0]) and sending an un-delayable message to the process to be triggered after a certain amount of time (e.g., issuing a timeout message for p[0] after passing t time units from sending its heartbeat). A similar watchdog mechanism is used to enforce a maximum round-trip delay on channels; the heartbeats are time-stamped with their delay when delivered from p[0] to each p[i] and the corresponding watchdog will resume counting down from the point it has left when the replying heartbeat is set on the channel in the reverse direction. For broadcast messages delays are controlled by a separate watchdog for each process.

We give next an overview of the formal specification of main processes in our formalization of the binary heartbeat protocol and point out the changes that are made in order to obtain the other versions of the protocol.

3.2 The binary heartbeat protocol in mCRL2

Process p[0]

We define process p[0] by means of following five parameters. Here "Bool" and "Nat" are sorts (types) [GMvWU06] for boolean and natural numbers respectively.

- $active : Bool$. A flag that shows the state of process, i.e., $true$ if active otherwise $false$. The initial value for this parameter is $true$.
- $rcvd : Bool$. A flag that denotes the receiving a reply from p[1], i.e., $true$ if received, or otherwise $false$. The initial value for this parameter is $true$.
- $t : Nat$. Length of the time period to exchange the beat messages, of which the initial value is $tmax$.
- $tmin : Nat$. Lower bound for waiting time of each round.
- $tmax : Nat$. Upper bound for waiting time of each round.

```
1:  P0(t : Nat, active, rcvd : Bool, tmin, tmax : Pos) =
2:  tick_p0.P0(t, active, true, tmin, tmax)
3:  +
4:  active → inactivate_v_p0.P0(t, false, rcvd, tmin, tmax)
5:  +
6:  from_p1(hb1).(active → P0(tmax, active, true, tmin, tmax)
7:                              ◇
8:                               P0(tmax, active, false, tmin, tmax))
9:  +
10: active → (timeout_at_P0.
11:             (rcvd → for_p1(hb0).send_ticking_time(tmax).
12:                        P0(tmax, active, false, tmin, tmax)
13:                    ◇
14:                      t div 2 ≥ tmin → for_p1(hb0).
15:                          send_ticking_time(t div 2).
16:                          P0(t div 2, active, false, tmin, tmax)
17:                      ◇
18:                      inactivate_nv_p0.P0(tmax, false, rcvd, tmin, tmax)
19:              )
20:           );
```

Process p[0] comprises four summands separated by nondeterministic choice (denoted by +). In line 2 of the above specification for p[0], tick_p0 denotes the clock tick, which allows for time to pass and in line 4, an active process may non-deterministically voluntarily decide to become inactivated. (Notation $c \rightarrow p_0 \diamond p_1$ stands for "if c, then $p0$ else $p1$; when the else part is not mentioned, it is assumed to be deadlock. In case of mCRL2 processes without timed actions, deadlock is the unit element for non-deterministic choice.) Line 6 shows that, the process can receive the heartbeat of p[1] without any precondition. Line 10 presents the behavior of p[0] after the timeout; the timeout message is synchronized with a watchdog (described below). Namely, first it checks the reply from p[1] and if the reply is there within the given time t, then the waiting time is set to $tmax$, or otherwise it is decreased to its half. For readability, we have annotated each send and receive action in p[0] and p[1] with the type of heartbeat being sent and received: heartbeats of p[0] and p[1] are denoted by *hb0* and *hb1*, respectively. In the same step, the new value of waiting time is also compared with $tmin$ to decide wether to go for non-voluntarily inactivation or to continue running the protocol. The reduced transition system of this process (for $tmax = 2$ and $tmin = 1$, after hiding the action *send_ticking_time* and reducing modulo weak-trace equivalence) is depicted in Figure 1.

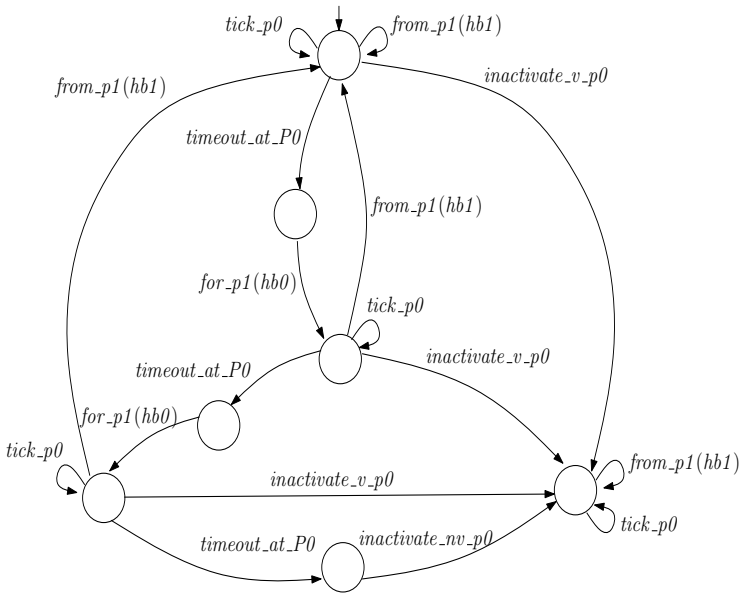


Figure 1: Reduced transition system for process p[0] with $tmax$=2 and $tmin$=1

Timeout stopwatch for p[0]

: The following stopwatch starts ticking by synchronizing on $rcv_ticking_time(i)$; this action synchronizes with $send_ticking_time(tmax)$ in the specification of p[0]. Notation $\sum_{i:Nat}$ is a generalization of nondeterministic choice, which allows for making a choice from a possibly infinite domain. The choice in this process represents the possibility of receiving any natural number as the time limit for the watchdog. The stopwatch subsequently counts up at each clock tick, until it either reaches the received time (i), in which case it sends a time-out message to p[0]. Initially this stopwatch receives $tmax$ from p[0] and it receives a value for i, every time p[0] sends a heartbeat to p[1].

1: $Stop_Watch_p0 =$
2: $tick_sw_p0.Stop_Watch_p0+$
3: $\sum_{i:Nat} rcv_ticking_time(i).Start_Ticking_p0(0,i);$
4:
5: $Start_Ticking_p0(t, time : Nat) =$
6: $(t \approx time) \rightarrow send_timeout_P0.Stop_Watch_p0$
7: $\diamond$
8: $tick_2.Start_Ticking_p0(t+1, time);$

Process for p[1]

The structure of p[1] is very similar to that of p[0]; it only replies to the heartbeat received from p[0] right away and has a timeout on $3tmax - tmin$, resulting in immediate non-voluntary inactivation. Process for p[1] is defined by means of three parameters; $tmin$, $tmax$ and active, which stand for the same intuition as in process p[0]. The transition system of this process (for $tmax = 2$ and $tmin = 1$) is depicted in Figure 2.

1: $P1(tmin, tmax : Pos, active : Bool) =$
2: $tick_p1.P1(tmin, tmax, active)$
3: $+$
4: $active \rightarrow inactivate_v_p1.P1(tmin, tmax, false)$
5: $+$
6: $from_p0(hb0).(active \rightarrow for_p0(hb1).snd_reset_sw_p1.P1(tmin, tmax, active)$
7: $\diamond$
8: $P1(tmin, tmax, active))$
9: $+$
10: $(active) \rightarrow timeout_at_P1.inactivate_nv_p1.P1(tmin, tmax, false);$

First four lines have the same purpose as in process for p[0] while in line 6, p[1] receives the beat from p[0] and if active then it sends an immediate reply. Line 10 shows the non-voluntarily inactivation of p[0] due to timeout, i.e., $3tmax - tmin$ units of time have passed without receiving message from p[0]. This timeout message is synchronized with the stopwatch given below.

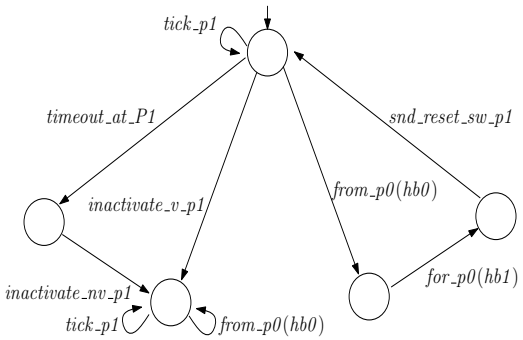


Figure 2: Transition system for process p[1] with $tmax = 2$ and $tmin = 1$

Stopwatch for p[1]

This stopwatch works as a monitor and sends timeout messages to p[1]. The timeout occurs at p[1] when it is active and doesn't receive any beat message during a period of $3tmax - tmin$ time units [GM98].

1: $Stop_Watch_p1(t, tmax, tmin : Nat) =$
2: $reset_sw_p1.tick_sw_p1.Stop_Watch_p1(0, tmax, tmin)$
3: $+$
4: $rcv_inactivate_v_p1.Idle_Ticking$
5: $+$
6: $(t \approx 3 \times tmax - tmin) \rightarrow for_p1_timeout.Stop_Watch_p1(0, tmax, tmin)$
7: $\diamond$
8: $tick_sw_p1.Stop_Watch_p1(t + 1, tmax, tmin)$;
9: $Idle_Ticking = tick_sw_p1.IdleTicking$;

The stopwatch gets reset by receiving *reset_sw_p1*, which synchronizes with the message from p[1] after it receives a heartbeat from p[0] (and replies to it). It also gets inactivated by receiving a message indicating that p[1] has been inactivated voluntarily. Otherwise, it ticks and counts up to its limit $3 \times tmax - tmin$ after which it sends a timeout message to p[1].

Communication channels

We define the following two processes, one for channel from p[0] to p[1] and other from p[1] to p[0] as shown below. Both processes synchronize on the clock ticks. The first process receives a heartbeat from p[0] and then non-deterministically either loses it or delivers it to p[1]. Losing a message is indicated by an action "lose_message". The functionality of the other process is identical but in the reverse direction. Both processes synchronize with a stopwatch in order to ensure the timely delivery of messages w.r.t. the round-trip maximum delay specified by *tmin*.

1: $Channel_p0_to_p1 =$
2: $tickp0p1.Channel_p0_to_p1$
3: $+$
4: $rcv_from_p0(hb0).(start_sw_ch(hb0) + lose_message).Channel_p0_to_p1$;
5:
6: $Channel_p1_to_p0 =$
7: $tickp1p0.Channel_p1_to_p0$
8: $+$
9: $rcv_from_p1(hb1).(update_sw_ch(hb1) + lose_message).Channel_p1_to_p0$;

Stopwatch for channel delay

This stopwatch measures the total delay in communication channels and ensures that the round trip between p[0] and p[1] will be completed within *tmin* units of time. According to [GM98], "*tmin* is upper bound on the round-trip delay between p[0] and p[1]". So this stopwatch starts counting the number of ticks when the heartbeat of p[0] is received at communication channel, i.e., Channel_p0_to_p1, and it gets reset when the heartbeat from p[1] is delivered to p[0] by the channel in the reverse direction.

```
1: Stop_Watch_Ch(t, tmin : Nat) =
2: tick_ch.Stop_Watch_Ch(t, tmin)
3: +
4: rcv_start_sw_ch(hb0).Start_Ticking_p0_to_p1(0, tmin)
5: +
6: rcv_update_sw_ch(hb1).Start_Ticking_p1_to_p0(t, tmin);
7:
8: Start_Ticking_p0_to_p1(t, tmin : Nat) =
9: (t < tmin − 1) → (tick_ch.Start_Ticking_p0_to_p1(t + 1, tmin)
10:                  +
11:                  send_to_p1(hb0).Stop_Watch4(t, tmin)
12:                  ◇
13:                  send_to_p1(hb0).Stop_Watch4(t, tmin);
14:
15: Start_Ticking_p1_to_p0(t, tmin : Nat) =
16: (t < tmin − 1) → (tick_ch.Start_Ticking_p1_to_p0(t + 1, tmin)
17:                  +
18:                  send_to_p0(hb1).Stop_Watch_Ch(0, tmin))
19:                  ◇
20:                  send_to_p0(hb1).Stop_Watch_Ch(0, tmin);
```

Line 2 contains the tick for synchronization and line 4 shows that this stopwatch starts counting when the channel from p[0] to p[1] receives the message. The other way round, when p[1] sends a reply, stopwatch 4 is updated with the already spent time so that the respective round may be completed within *tmin* time units. Process shown on line 8 and 15 are for counting the ticks when message travels from p[0] to p[1] and then back from p[1] to p[0], respectively. Both of these processes also ensure that the round-trip delay is at most *tmin*.

3.3 The static heartbeat protocol in mCRL2

As discussed in Section 2.2, there are a number of participants running the static heartbeat protocol with p[0]. Hence, the heartbeat of p[0] is broadcasted for all the participants. In our specification settings, the communication channel performs this function through a separate process, called "Broadcaster". We define these processes as:

```
1: Channel_p0_to_p(n : Nat) = tick_p0p1.Channel_p0_to_p(n)
2: +
3: rcv_from_p0(hb0).Broadcaster(hb0, 0, n);
4:
5: Broadcaster(msg : p0_to_p, np, n : Nat) =
6: (np < n) → (start_sw4(msg, np) + lose_message).
7:                Broadcaster(msg, np + 1, n)
8:           ◇
9:                Channel_p0_to_p(n);
10:
11: Channel_p_to_p0 = tick_p1p0.Channel_p_to_p0
12: +
13:     Σ_{message:p_to_p0} . Σ_{i:Nat} .rcv_from_p(message, i).
14:                  (update_sw4(message, i) + lose_message).Channel_p_to_p0;
```

The parameter 'n' denotes the number of participants. In line 3 the channel receives the heartbeat from p[0] and initiates the process of broadcasting, i.e., "Broadcaster". Line 6 presents

the non-deterministic choice between losing a message and delivering it (along-with starting the respective stopwatch to ensure the maximum round-trip delay). In lines 11-14, we present the channels that lose or deliver the message in the reverse direction.

Other processes are similar to their counterparts in the binary heartbeat protocol except that the heartbeat of p[0] also contains the identifier of the recipient. In this version we have also defined new functions for calculating with manipulating lists of time periods and boolean variables (indicating the receipt of a heartbeat). These functions are used in definition of process for p[0] in static heartbeat protocol as shown below:

1: $P0(t : Nat, active : Bool, rcvd : List(Bool), tmin, tmax : Nat, tm : List(Nat)) =$
2: $tick_p0.P0(t, active, rcvd, tmin, tmax, tm)$
3: $+$
4: $(active) \rightarrow (active) -> inactivate_v_p0.P0(t, false, rcvd, tmin, tmax, tm)$
5: $+$
6: $\sum_{i:Nat} from_p(hb1, i).reset_s w1.)$
7: $((active) \rightarrow P0(tmax, active, update(i, true, rcvd), tmin, tmax, tm)$
8: $\diamond$
9: $P0(tmax, active, rcvd, tmin, tmax, tm))$
10: $+$
11: $active \rightarrow (timeout_at_P0.$
12: $(minimum(updateTM(rcvd, tm, tmax)) \geq tmin)$
13: $\rightarrow$
14: $send_ticking_time(minimum(updateTM(rcvd, tm, tmax))).$
15: $broadcast(hb0).$
16: $P0(minimum(updateTM(rcvd, tm, tmax)), active, assignFalse(rcvd), tmin,$
17: $tmax, updateTM(rcvd, tm, tmax))$
18: $\diamond$
19: $inactivate_nv_p0.P0(tmax, false, rcvd, tmin, tmax, tm)$
20: $);$

There are two changes in parameters if we compare it with the binary heartbeat protocol. First, *rcvd* is a list instead of single value and secondly *tm* is list of time periods of all processes; functions *update*, *updateTM*, *assignFalse* and *minimum* operate on the aforementioned lists.

3.4 The expanding heartbeat protocol in mCRL2

In this version of the protocol, p[0] maintains one more list than the static heartbeat protocol, namely, the list of participants which have joined the protocol. Each participating process sends its heartbeat to join, waits for *tmin* units of time and continue sending till the response from p[0]. So we have introduced another stopwatch that is instantiated with each participating process. This stopwatch sends a timeout to its respective process, so that it it starts sending its heartbeat before *tmin* units of time.

3.5 The dynamic heartbeat protocol in mCRL2

As discussed in Section 2.4, the beat messages in this protocol carry a boolean parameter indicating the intention to join or to leave the protocol (denoted by *true* or $false$, respectively). So the process for p[0] is almost same as its counterpart in the expanding heartbeat protocol except for the arguments of the heartbeats and updating the joined list accordingly. However the process for p[i] has more choices in each iteration as described below:

1. Become inactive

2. If not joined then send a heartbeat with parameter *true* after every *tmin* units of time as a joining request.

3. If joined and active then send heartbeat with parameter $false$ for leaving or heartbeat with parameter *true* for remaining in the protocol.

4. Become non-voluntarily inactive, if there is no response from p[0] within $3tmax - tmin$.

5. Receive a beat from p[0] and send an immediate reply if active.

We modified process for p[i] to address these choices as specified below.

$P(n, tmin, tmax : Nat, active, join : Bool) =$
$tick_p.P(n, tmin, tmax, active, join)$
$+$
$active \rightarrow inactivate_v_p(n).P(n, tmin, tmax, false, join)$
$+$
$from_p0(hb0, n).resetSW3(n).$
$(active \rightarrow (for_p0(hb1, n, true).P(n, tmin, tmax, active, true)$
$+$
$for_p0(hb1, n, false).P(n, tmin, tmax, active, false))$
$\diamond$
$P(n, tmin, tmax, active, true))$
$+$
$active \rightarrow timeout_at_P(n).inactivate_nv_p(n).$
$P(n, tmin, tmax, false, join)$
$+$
$active \wedge join \rightarrow timeoutX(n).for_p0_2join(n).$
$tick_p.P(n, tmin, tmax, active, join);$

4 Formal specification in UPPAAL

4.1 Introduction to UPPAAL

UPPAAL [KPW97] is a tool-suit used for modeling, simulating and verification of real-time systems. The input language of UPPAAL allows for networks of timed-automata communicating via (handshaking or broadcast) communication channels. In UPPAAL one can define and exploit a number of clocks in order to specify timing constraints in the specification. All clocks increase at the same rate, can be reset when taking a transition and may guard transitions by checking their value (or the difference between two clocks) against constants. Moreover, one may specify state invariants in terms of clock values. Finite data types such as Booleans and bounded integers can also be used to define state variables and transition guards and effects in timed automata.

In this section, we briefly describe our formalization of accelerated heartbeat protocols in the timed-automata formalism of UPPAAL. We start with a brief description of the binary protocol and then explain incremental additions leading to each new version.

4.2 The binary heartbeat protocol in UPPAAL

Timed automaton for p[0]

The timed-automaton for process p[0] is depicted in Figure 3. The initial state is named Alive and is denoted by a double-circle. The only clock in this specification is *waiting*, which is initially reset to 0. Variable t, denoting the current waiting time is initially set to 0. Once a process is alive, its waiting period should be at least *tmin*, and its waiting time can grow to the time bound specified by variable t (initially set to *tmax*. Afterwards, a time-out is issued and the transition to

the state called *time-out* has to be taken. Alternatively, and at any moment of time, the process can move to the state *V_Inactivated* and as a result, become inactivated voluntarily.

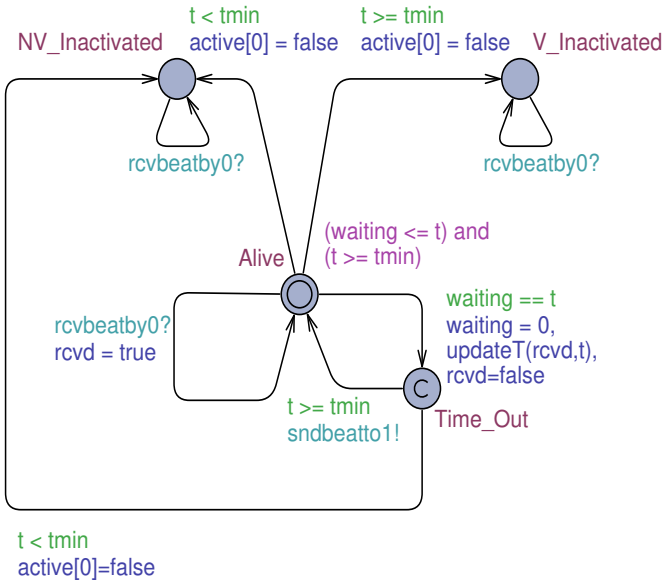


Figure 3: Timed-automaton for p[0] in the binary heartbeat protocol

Time_Out state is a committed state, denoted by an encircled C, meaning that when the network of timed-automata reaches a combination containing one such state, time cannot pass, and an outgoing transition of a committed state must be taken immediately. This ensures that p[0] resets the waiting time and immediately computes the new waiting period, and makes a decision as to become non-voluntarily inactivated (move to the state *NV_Inactivated* or continue running the protocol instantaneously.

Note that p[0] can receive messages from p[1] regardless of being alive or inactive.

Timed-automaton for p[1]

The timed-automaton for process p[1] is depicted in Figure 4. The initial state of p[1] is denoted by *Alive*. The only clock used in the timed-automaton for p[1] is *waitingforbeat* which measures the amount of time since the last received heartbeat. Upon receiving a heartbeat, the automaton moves to the committed state *Rcvd* from which it should instantaneously move to the alive state by resetting the clock and sending its heartbeat to p[0].

Depending on the total waiting time, the process may be forced to become inactivated non-voluntarily, or choose to become inactivated voluntarily.

Timed-automaton for communication channels

The timed-automaton for the communication channels in the binary heartbeat protocol is depicted in Figure 5. It simply receives the first message from p[0], either decides to communicate it or loses it. Upon losing a message a boolean variable will be set to true which will be later used for verifying correctness properties. Moreover, the total round-trip delay is enforced by means of the clock *delay*, which is checked against the constant *tmin*. In case a process is inactivated then the communication channel will stop waiting for heartbeats from that process.

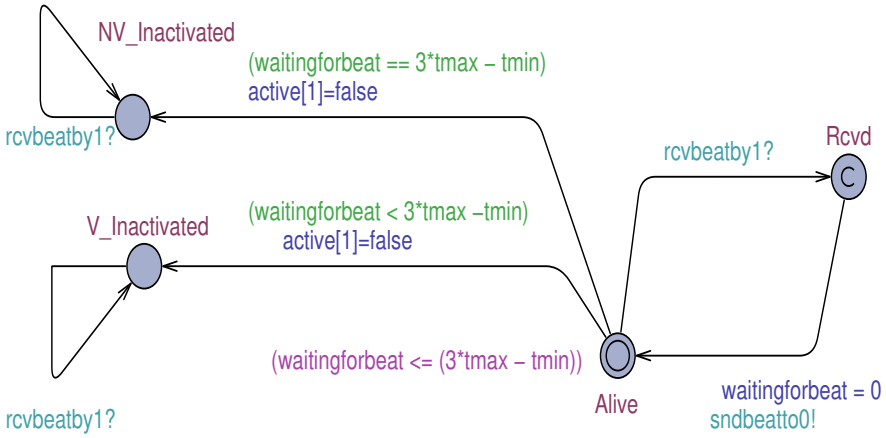


Figure 4: Timed-automaton for p[1] in the binary heartbeat protocol

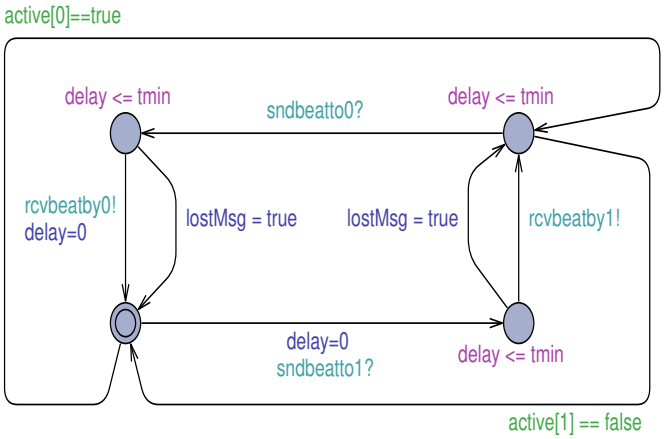


Figure 5: Timed-automaton for communication channels in the binary heartbeat protocol

4.3 The static heartbeat protocol in UPPAAL

Process p[0] in the static version of the protocol is very similar to the one given in Figure 3. The only differences are that firstly, broadcast channels, which are built-in primitives in UPPAAL are used to sent the heartbeat of p[0] to the communication channels between p[0] and the other participant and secondly, lists and operations thereon replace the single variables storing the waiting times and the receipt of heartbeats. We decided to a separate communication channels between p[0] (rather than a single channel broadcasting the heartbeat of p[0] simultaneously to all participants and vice versa) and each participant in order to allow for different communication delays in each direction.

The structure of each p[i] process is identical to the process p[1] depicted in Figure 4.

Finally, several instances of a communication channel, identical to the specification given in Figure 5, communicate the messages from p[0] to p[i] and vice versa.

4.4 The expanding heartbeat protocol in UPPAAL

There are a few changes in the specification of expanding heartbeat protocol, when compared to the static version.

Firstly, process p[0] takes note of processes that have joined the protocol (by sending a heart-

beat) and only takes them into account when calculating the new waiting time. Otherwise, the structure of p[0] is identical to its counterpart in the static version.

Secondly, each process p[i] starts off by sending out its join request (by sending a heartbeat) and then keeps on sending this request until it receives a heartbeat with parameter *true* from p[0] or becomes inactivated. Sending the first join request cannot happen later than *tmin* units of time. This is guaranteed by using a new clock called *waitingtojoin*. The specification of process p[i] in this protocol is given in Figure 6. In this figure, the initial state of the timed-automaton is an *urgent* state, meaning that time cannot pass before leaving this state. Intuitively, this means that the process can not abstain from running the protocol by remaining in the initial state.

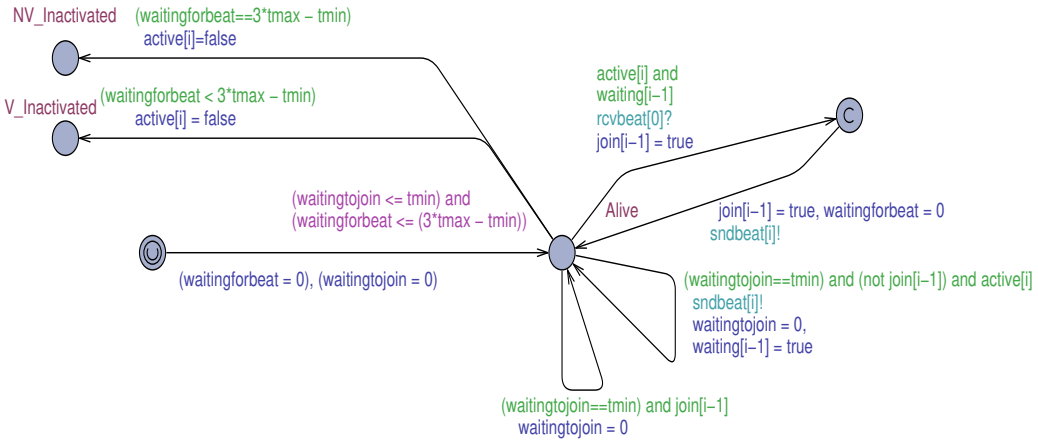


Figure 6: Timed-automaton for p[i] in the expanding heartbeat protocol

Finally, p[i] processes initiate the protocol and thus, the communication channel should allow for the delivery of joint requests. We model this by adding an extra channel per participating process which is only active before that the process has joined the protocol. Afterwards, the same communication channel as in the static protocol will take care of communicating messages.

4.5 The dynamic heartbeat protocol in UPPAAL

Process p[0] in the dynamic version has the extra possibility of receiving leave requests. We denote receiving join and leave requests from process p[i] by messages *rcvfalsebeat[i]* and *rcvtrueebeat[i]*, respectively. The rest of the structure and the logic behind p[0] is identical to its expanding counterpart. For sake of completeness, the timed-automaton for p[0] is given in Figure 7.

Similarly, process p[1] has the extra option of leaving the protocol by replying *sndfalsebeat[i]* to the heartbeat of p[0]. The timed-automaton for p[i] is depicted in Figure 8.

5 Verifying protocol requirements

5.1 General requirements

In [GM98, p. 2], we read the following requirement:

> ... if one or more processes ever choose to become inactive, then all processes in the network eventually become inactive.

This progress (eventuality) requirement has been further specified in [GM98, p. 3] and [MG04, p. 97] as follows:

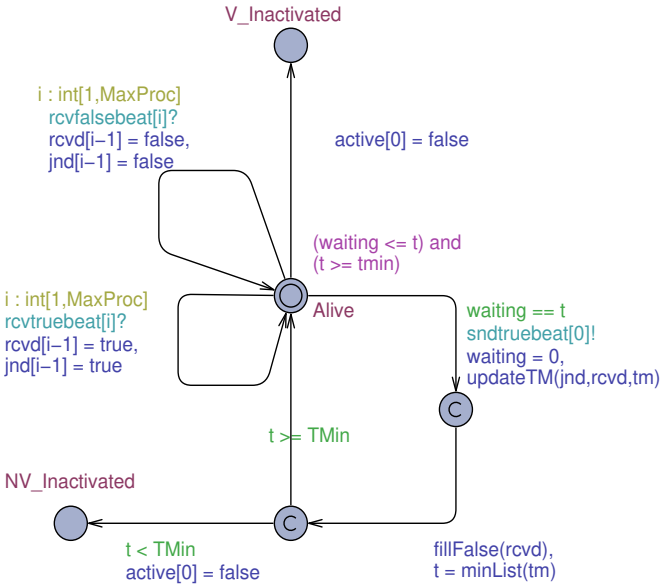


Figure 7: Timed-automaton for p[0] in the dynamic heartbeat protocol

> ... if p[0] does not receive any beat message for a period of $2tmax$, then p[0] becomes inactive. [1]

We thus define our first requirement as follows:

> **(R1)** For each $i > 0$, if p[0] does not receive a heartbeat from p[i] for a period of $2tmax$, then p[0] becomes inactive non-voluntarily.

The following symmetric requirement is given in [GM98] about the inactivation of p[0] and its effect on the other participants:

> If process p[0] becomes inactive voluntarily, then all p[i] will become inactive non-voluntarily after at most $3tmax$ time units.

However, this requirement is enforced trivially by accommodating a time-out mechanism in all p[i] processes which forces each p[i] to be inactivated if it does not receive a heartbeat from p[0] within $3tmax - tmin$ units of time. Hence, we do not discuss this requirement in the remainder of this paper.
In [GM98, p. 2], it is stated that:

> If every process in the network, continues to choose to remain active [and no message is lost or delayed beyond the limit], then all processes remain active indefinitely.

We added to the premises of the above requirement that no heartbeat message is lost or delayed beyond the specified limit on the delays; otherwise, the above requirement is vacuously violated by all heartbeat protocols.

> **(R2,R3)** For each $i \geq 0$, if there has been no voluntary inactivation of any p[j] (for each $i \neq j$) and no message is lost or delayed beyond its limit, then p[i] is not inactivated non-voluntarily.

[1] This requirement is stated for the binary heartbeat protocol in [GM98]. But the same constants are used for the other versions of the protocol and there is no further mention of a different upper bound for the other versions. Thus, we assume that the same upper bound should hold for the other versions, as well.

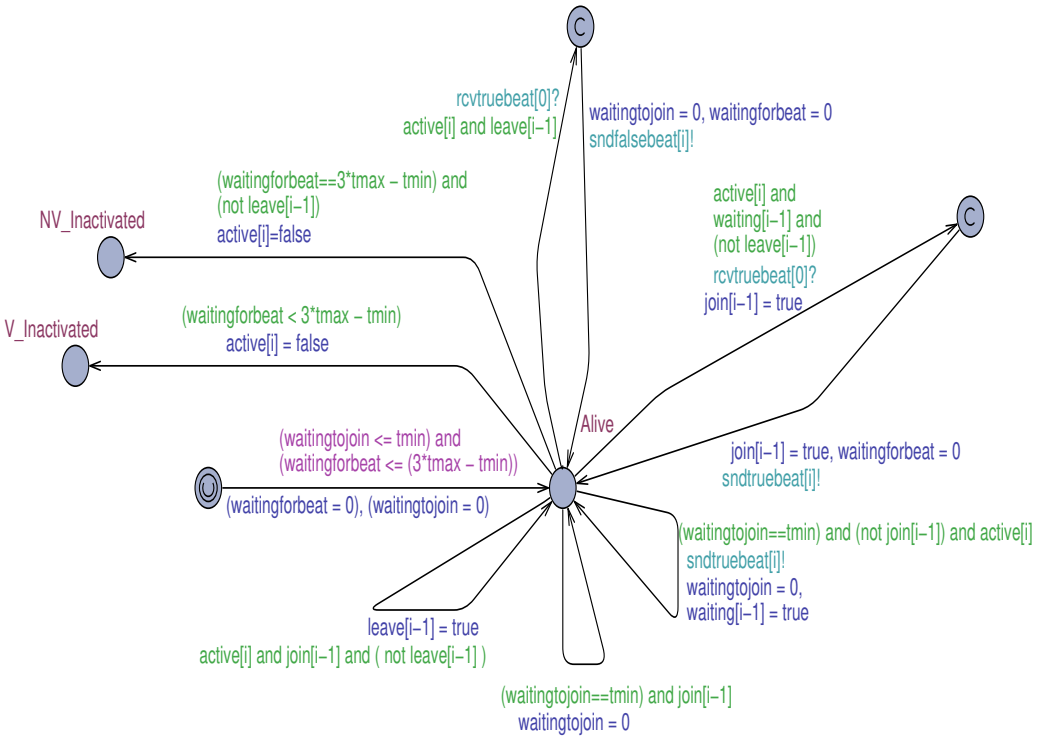


Figure 8: Timed-automaton for p[i] in the dynamic heartbeat protocol

We split the above statement into two requirement **(R2)** and **(R3)**. For each $i > 0$, **(R2)** requires that as long as p[0] and all p[j], where $j \geq 0$ and $j \neq i$, are active, and all channels are up, the protocol should not inactivate p[i] non-voluntarily. Requirement **(R3)** specifies that if all p[j] processes, where $j > 0$, are active and all channels are up, then the protocol should not inactivate p[0] non-voluntarily.

5.2 Formalizing the requirements in the modal μ-calculus

R1. To formalize the progress requirement (R1), we devised a watchdog for each process p[i], which starts counting down from $2tmax$ and is reset by each heartbeat of p[i] received at p[0]. All watchdogs are inactivated when p[0] is non-voluntarily inactivated. If a watchdog reaches 0 and does not receive an inactivation message from p[0] before the next time unit, it will issue a special message called "*error*". (In case of expanding and dynamic protocols, the watchdog of p[i] is only active after that the first joining request, i.e., a heartbeat with parameter *true*, is sent by p[i] and in the dynamic protocol, it remains active until a leave request is sent by p[i].) Then, we use the following simple formula in the modal μ-calculus, for checking the reachability of a trace containing an action *error*.

$$[true^*.error]false$$

The notation $[r]\phi$, where r is a regular expression over actions and ϕ is a formula, specifies that after all traces satisfying r, ϕ should hold. Particularly, $[r]false$ specifies that no trace satisfying r is reachable (since otherwise *false* should be satisfied, which is impossible).

R2. For the binary and static versions of this protocol, the formalization of this requirement is straightforward. Namely, for each $i > 0$, the following formula formalizes requirement (R2).

$$[(\bigwedge_{j\geq 0, i\neq j} \overline{inactivate_v_p_j} \wedge \overline{lose_msg})^*.inactivate_nv_pi]false$$

The above formula states that non-voluntary inactivation of p[i], denoted by action $inactivate_nv_pi$, is always preceded by a voluntary inactivation of another process p[j], denoted by action $inactivate_v_pj$, or a message loss, denoted by action $lose_msg$.

However, for the expanding and dynamic protocols, the above formula is too weak. It disregards joining and leaving requests and for example, allows for a process which has not joined, or joined but then left the protocol to become non-voluntarily inactive. Next, we give the formalization of this requirement in modal μ-calculus for the dynamic protocol and for three participants:

$$\begin{array}{l}[\overline{p_joined(1)}^*.inactivate_nv_p(1)]false\wedge\\ [nofault(\{0\})^*.p_left(2).nofault(\{0\})^*.inactivate_nv_p(1)]false\wedge\\ [nofault(\{0,2\})^*.inactivate_nv_p(1)]false,\end{array}$$

where $nofault(I)$ stands for $\overline{lose_msg}\wedge\bigwedge_{i\in I}(\overline{inactivate_v_p0}\wedge\overline{inactivate_nv_p0})$ and $\overline{p_left(2)}$ stands for $sent_by_p[2](false).\ true^*.\ rcv_from_p[0](true)$. This formula states that p[1] is not allowed to be non-voluntarily inactivated, if it has not joined the protocol, or if it has joined, no other process has joined and no fault has occurred in p[0] or the channel, or if no fault has occurred in p[0], the participants and the channel.

R3. The following formula formalizes requirement (R3), which states that non-voluntary inactivation of p[0] must be preceded with the voluntary inactivation of some p[i] (for some $i > 0$) or a message loss.

$$[(\bigwedge_{i>0} \overline{inactivate_v_pi} \wedge \overline{lose_msg})^*.inactivate_nv_p0]false$$

As in requirement R2, the formalizations of R3 for the expanding and the dynamic protocols are more involved.

$$\begin{array}{l}[(\bigwedge_{i>0} \overline{p_joined(i)})^*.inactivate_nv_p0]false\wedge\\ [noinact(\{1\})^*.p_left(2).noinact(\{1\})^*.inactivate_nv_p(1)]false\wedge\\ [noinact(\{0,1\})^*.inactivate_nv_p(1)]false,\end{array}$$

where $noinact(I)$ stands for $\overline{lose_msg} \wedge \bigwedge_{i\in I} \overline{inactivate_v_p0}$.

5.3 Formalizing the requirements in UPPAAL

R1. To formalize requirement R1, we devise a timed-automaton that runs in parallel with the protocol and observes the receipt of heartbeats and the the corresponding inactivation. If it observes that despite not receiving a heartbeat for $2 * tmax$, the monitor processes for the binary and dynamic protocols are given, respectively, in Figures 9.(a) and 9.(b).

Subsequently, requirement R1 for the binary protocol is formalized in UPPAAL in terms of (the negation of) the following reachability property:

$$E\Diamond\ M1.ErrorR1$$

For the static, expanding and dynamic protocols, for each participant, one monitor automaton is instantiated and requirement R1 has the following form:

$$E\Diamond\ (M1.ErrorR1 or M2.ErrorR1)$$

Here M1 and M2 are instances of type Monitor, given in Figure 9.(b).

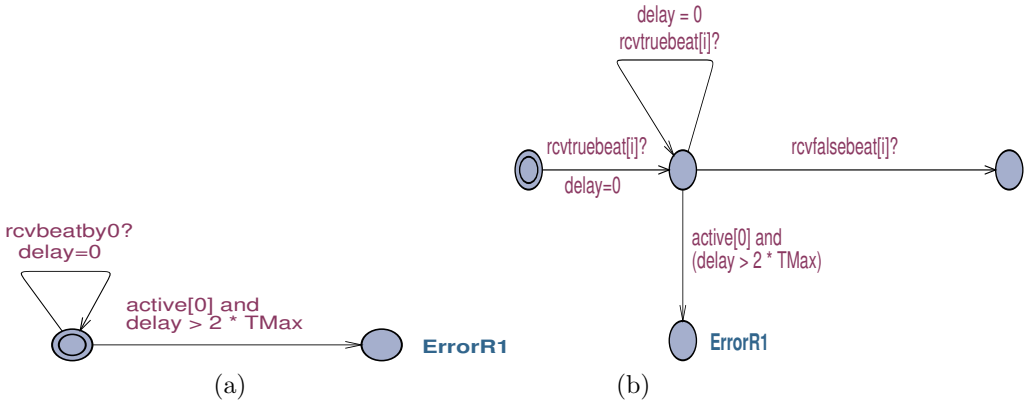


Figure 9: Monitor timed-automaton for R1 in (a) the binary and (b) the dynamic heartbeat protocol

R2. Requirement R2 for the binary protocol is formalized by the following reachability formula:

$$E\Diamond((notCh.lostMsg)andP0.NV_Inactivatedand P1.Alive)$$

For the dynamic protocols, the following property captures R2:

$$E\Diamond \quad ((not\ C11.lostMsg)\ and\ (not\ C12.lostMsg)\ and\ Process0.NV_Inactivated\ and\ ((P1.Alive\ or\ (not\ jnd[0])\ or\ leave[1])\ and\ (P2.Alive\ or\ (not\ jnd[1])\ or\ leave[1])))$$

R3. Symmetrically, requirement R3 is captured in the specification of binary protocol by the following formula:

$$E\Diamond((not\ Ch.lostMsg)\ and\ P1.NV_Inactivated\ and\ P0.Alive)$$

For the dynamic protocol, the requirement is specified in terms of the following formula:

$$E\Diamond \quad ((not\ C11.lostMsg)\ and\ (not\ C12.lostMsg)\ and\ P1.NV_Inactivated\ and\ (Process0.Alive\ and\ (P2.Alive\ or\ (not\ jnd[1]))))$$

5.4 Verification techniques

We applied model checking techniques for the verification of R1, R2 and R3 (discussed in Section 5.2) with respect to the different versions of accelerated heartbeat protocols. We used the process algebra mCRL2 [GMvWU06] for modeling and evaluated the formulae with model checker CADP [GMLS07]. In the process algebraic approach, a number of steps should be taken between modeling (with mCRL2) and model checking (with CADP). Namely, we translated the models to the respective linear process specifications (LPS) [GPU01] (a simple format used for storing and manipulating recursive process definitions). From the LPS, we generated the state space after applying different state space reduction techniques, such as minimizing modulo strong bisimilarity and eliminating constants, superfluous summands and inconsequential parameters. We also modeled the same protocols in timed-automata and used UPPAAL [KPW97] to verify the properties specified in Section 5.3. Both model checkers produced similar results.

5.5 Verification results

We used different data sets for *tmin* and *tmax* for each protocol presented in [GM98] and [MG04]. (Note that the only constraint on *tmin* and *tmax* according to [GM98] is that $0 < tmin \leq tmax$.) Since the counter-examples reported for the (revised) binary, two-phase and static heartbeat protocols are identical, we report about them once.[2]

5.5.1 The (revised) binary, two-phase and static heartbeat protocols

tmin	1	4	5	9	10
tmax	10	10	10	10	10
R1	F	F	F	T	T
R2	T	T	T	T	F
R3	T	T	T	T	F

Table 1: Verification results for (revised) binary, and static protocols

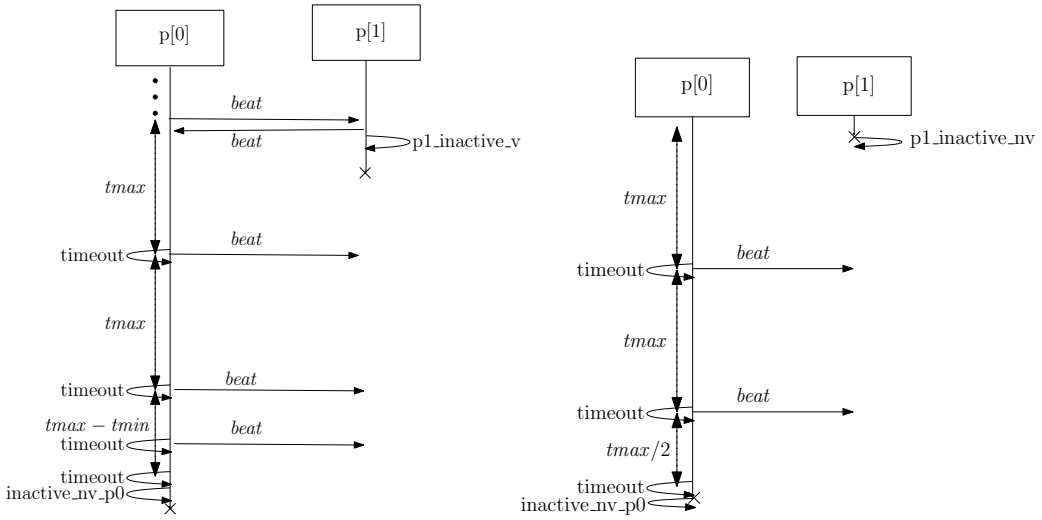
- **R1:** This property is violated in the (revised) binary and static protocols provided that *tmin* is relatively small compared to *tmax*. A counter-example for this property is depicted by the sequence diagram in Figure 10. In this trace p[0] sends a heartbeat to p[1], p[1] receives it and replies to it and is voluntarily inactivated right away. Then, p[0] receives the heartbeat of p[1] and after a period of at most *tmax* time units, it receives a time-out. At this point, p[0] observes that a reply has been received from p[1] and hence, it sets the waiting time t to *tmax*. From that point on, the total time to non-voluntary inactivation of p[0] takes at most $2tmax - tmin$ time units. Thus, the total time to inactivation of p[0] can grow up to $3tmax - tmin$, which is greater than $2tmax + tmin$, if $tmax > 2tmin$. This was illustrated by the counter-examples generated for $tmin = 1$ and $tmin = 4$ by the model-checker. (Note that [GM98] considers $tmax > 2tmin$ to be the "usual situation"; the authors write in p. 4 that $tmax + tmin$ is usually less than $tmax + tmax/2$. Also in [MG04], the authors choose 1000 and 10000 as typical values for *tmin* and *tmax*, respectively.)

  For the case where $2tmin = tmax$, we found a different, yet very simple, counter-example, depicted in Figure 10.(b). (This counter-example also holds for the same protocols in case $2tmin < tmax$ and represents a different phenomenon in the protocols.)

- **R2:** Requirement R2 is violated in the (revised) binary, two-phase and static protocols, when *tmin* and *tmax* have the same value, e.g., 10. The counter-example is illustrated in Figure 11, which represents the following scenario. Consider a trace, where p[0] sends its first heartbeat after its first timeout (after that $t = tmax$ units of time pass) and round-trip delay is equal to its upper-bound, i.e., *tmin*. The total time spent at the delivery of p[0]'s heartbeat will thus be $tmax + tmin$, which is equal to $3tmax - tmin$, if $tmax = tmin$ and the allowed delay limit is consumed in the channel from p[0] to p[1]. According to specification, a timeout occurs at p[1] when $3tmax - tmin$ time is reached without receiving a beat message. Hence, the timeout and receiving the heartbeat occur simultaneously at p[1] and if the former is processed first, it causes non-voluntarily inactivation of p[1], whereas p[0] has not been voluntarily inactivated and the communication channel is also up. For the revised binary protocol, only the initial delay of *tmax* is not present, but the essence of the counter-example remains the same.

- **R3:** This property is also violated in the (revised) binary, two-phase and static protocols, when the values of *tmin* and *tmax* are equal. The same counter-example, depicted in Figure

[2]For the two-phase heartbeat protocol the condition for non-voluntary inactivation of p[0] is not specified in [GM98]. Hence, we could not verify properties (R1) and (R3) for this version of the protocol.



(a) $2tmin < tmax$ (b) $2tmin \leq tmax$

Figure 10: Counter-examples for (R1) when $2tmin \leq tmax$

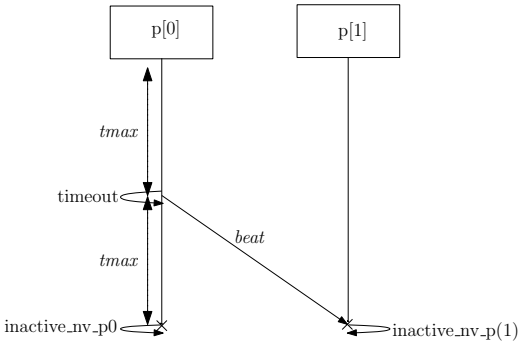


Figure 11: Counter-example for (R2) when $tmin = tmax$

11, illustrates this fact. In the same trace, p[0] is inactivated non-voluntarily without a voluntary inactivation of p[1]. In another counter-example, depicted in Figure 12, p[1] may remain alive and respond to the heartbeat of p[0] but its hearbeat is received at p[0] exactly after $tmin = tmax$ units of time. Then, both timeout and heartbeat arrive simultaneously at p[0] and if the former is processed first, p[0] will non-voluntarily be inactivated while p[1] is still active.

5.5.2 The expanding and dynamic heartbeat protocols

In cases where we could find a counter-example for the revised binary protocol, an almost identical counter-example is also reported for the expanding and dynamic protocols. (Namely, the expanding and dynamic protocol with one participant behave exactly the same as the revised binary protocol after the participant has sent a joining request at time 0 and the request is received at p[0] immediately.) However, expanding and dynamic heartbeat protocols contain many new traces, which result in more counter-examples. Next, we only give discovered counter-examples regarding the requirement (R2) that are not in common with the aforementioned protocol.

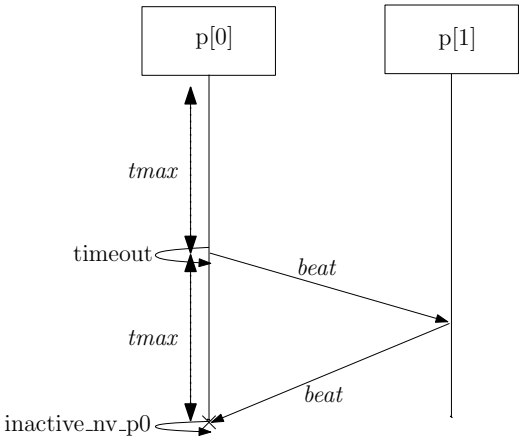


Figure 12: Counter-example for (R3) when $tmin = tmax$

tmin	1	4	5	9	10
tmax	10	10	10	10	10
R1	F	F	F	T	T
R2	T	T	F	F	F
R3	T	T	T	T	F

Table 2: Verification results for expanding and dynamic protocols

In addition to the counter-examples reported before, this property is violated in the expanding and dynamic protocols when $2tmin \geq tmax$. That is why in case $tmin = 5$, 9 or 10, this property is not satisfied. The counter-example for this case is depicted in Figure 13. In this trace, p[1] sends its hearbeat to join the protocol, but its heartbeat is received at p[0] right after the first time-out at p[0], and thus, p[0] does not send its heartbeat to p[1] before its next time-out. The heartbeat of p[0] may take at most *tmin* units of time before it reaches p[1]. Hence, p[1] only receives a beat from p[1] after $2tmax + tmin$ which is too late if $3tmax - tmin \leq 2tmax + tmin$, or in other words, $2tmin \geq tmax$.

6 Correcting the protocols

As observed in Section 5.5, all requirements of the protocols are violated under certain circumstances. These violations can be traced back to two main causes: inappropriate handling of simultaneous events and incorrect time-bounds for the inactivation of processes. In the remainder of this section, we explain the nature of these causes and propose fixes that can fix the discovered problems and even improve the performance of accelerated heartbeat protocols.

6.1 Simultaneous events

One clear source of problem in all heartbeat protocols is the possibility of simultaneous events and lack of appropriate treatment thereof. Particularly, if a heartbeat is received simultaneously with the occurrence of a timeout, the timeout may get precedence and thus, the receiving process may become non-voluntarily inactive while the sending process is still alive and in fact has sent its heartbeat on time. This results in a violation of properties R2 and R3 (see Figures 11 and 12). To solve this problem, receive operations must be given precedence over timeouts, i.e., before processing timeouts, it has to be checked whether the communication channels offer messages that

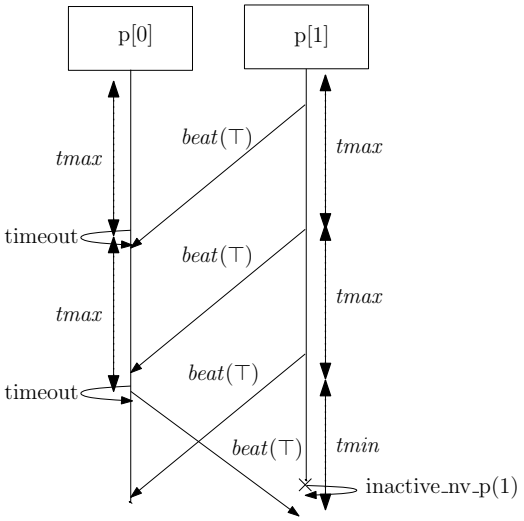


Figure 13: Counter-example for (R2) when $2tmin \geq tmax$

have to be delivered or not. In the former case, the pending messages are first processed and then timeouts are issued.

Adopting the priorities specified above removes all the counter-examples reported for R2 and R3 for binary and static heartbeat protocols. This fix is essential for solving the problems regarding the same properties for the expanding and dynamic protocols, but it is not sufficient as explained below.

6.2 Incorrect time-bounds

All heartbeat protocols assume a total waiting time of $2 * tmax$ for p[0] and $3 * tmax - tmin$ for p[i] process(es), respectively. We argue below that both of the above-mentioned time-bounds are either incorrect or imprecise (depending on the type of the protocol). Incorrectness of time-bounds leads to violation of properties R1 and R2 and imprecision leads to inefficiency of the protocol (i.e., unnecessarily long delays before detecting a process or channel failure).

Time bounds for the binary and static heartbeat protocols

As for the total waiting time for p[0], the maximum time between between receiving the last heartbeat from p[1] to the non-voluntary inactivation of p[0] is achieved when p[1] crashes at the beginning of the first round right after sending its heartbeat to p[0]. To maximize the total waiting time, assume that the heartbeat of p[1] is received at p[0] instantaneously. We distinguish the following two cases:

- $2 * tmin > tmax$: In this case, the maximal waiting time of p[0] is indeed $2 * tmax$ since after the second round, the waiting time is reduced to $tmax/2$ and since it holds that $tmax/2 < tmin$, p[0] will be non-voluntarily inactivated.
- $2 * tmin \leq tmax$: In this case, the maximal waiting time of p[0] is $3 * tmax - tmin$ according to the calculation given below.

$2 * tmax + \sum_{i=1}^{j} tmax/2^i =$ for a j s.t. $tmax/2^{j+1} < tmin \leq tmax/2^j$
$2 * tmax + \sum_{i=1}^{\infty} tmax/2^i - \sum_{i=j+1}^{\infty} tmax/2^i =$
$2 * tmax + tmax - 2 * tmax/2^{j+1} =$
$2 * tmax + tmax - tmax/2^j \leq$ (since $tmin \leq tmax/2^j$)
$3 * tmax - tmin$

Fixing the maximal waiting time for p[0] to the one given above removes all the counter-examples concerning the requirement R1 as reported in Table 1 (and does not introduce any new counter-examples).

Concerning the time bound for p[1], the maximal waiting time for p[1] is achieved when in the previous round, the heartbeat of p[0] was received at the beginning of the round, and the receipt of the heartbeat of p[0] in the current round is delayed till the end of the round(see Figure 12). Thus, the maximal waiting time in the binary and static heartbeat protocol is $2 * tmax$, which is a tighter bound than $3 * tmax - tmin$. This lower bound does not solve any correctness problem but adds to the efficiency of the protocol in that channel failures and process crashes are detected earlier by the participating processes.

We have implemented all the proposed fixes and model-checked the changed protocols; the model-checking shows that all the reported problems for the binary and static protocols are removed.

Time bounds for the expanding and dynamic heartbeat protocols

In the expanding and dynamic protocols, the time-bound for p[0] is identical to the binary and static protocols. However, the time-bound for p[i] processes is different due to the initial phase before joining the protocol. The maximal time-bound for p[i] is achieved when p[i]'s join request is received right after starting a new round and moreover, the reply from p[0] at the beginning of the next round takes *tmin* units of time before it reaches p[i]. This way, the delay between the start-up of p[i] and the first heartbeat from p[0] goes up to $2 * tmax + tmin$ (see Figure 13). Note that the time-bound proposed in [GM98], i.e., $3 * tmax - tmin$, is incorrect in case $2 * tmin \geq tmax$ and is inefficient otherwise.

Fixing the time-bounds as given above and adopting the fix proposed in Section 6.1 removes all of the counter-examples reported in Table 6.1. We have applied the fixes to our automata-theoretic models and model-checked the corrected versions (with all the different data-sets) of the protocols; model-checking these fixed models does not result in any counter-example for any requirement.

7 Conclusions

We formalized different versions of heartbeat protocols as specified in [GM98, MG04] in the process algebra mCRL2 and timed-automata-theoretic formalism of UPPAAL. We then formalized some natural properties on these protocols and verified them using the CADP tool-set and UPPAAL. We reported several counter-examples that were discovered during our formal analysis. The properties that are not satisfied by the accelerated heartbeat protocols are quite natural and essential. Hence, we proposed subsequent improvements on the protocols in order to meet these requirements seem inevitable. We model-checked the improved versions of the protocols and showed that they indeed satisfy our requirements.

We believe that the specifications developed in the course of researching heartbeat protocol can be readily used to verify similar protocols and protocols that build upon them, e.g., protocols for failure detectors. We are currently following this line and are applying our method to verify the correctness of failure detector protocols. Moreover, in the present version of the dynamic heartbeat protocol, an upper bound on the number of processes should be a priori known and additionally, a process can never join the protocol once it has left it. An improved version of the

dynamic heartbeat protocol allowing for an unbounded number of processes, which can join and leave at any time and proving the correctness of the extended protocol is also a future research goal.

Acknowledgments.

Jan Friso Groote and Michel Reniers provided valuable comments on the earlier versions of this report.

References

[BDL04] Gerd Behrmann, Alexandre David and Kim G. Larsen. A Tutorial on Uppaal. In *Proc. of SFM-RT'04*, vol. 3185 of *LNCS*, pp. 200–236, Springer, 2004.

[GMLS07] Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Proc. of CAV'07*, vol. 4590 of *LNCS*, pp. 158–163. Springer, 2007.

[GM98] Mohamed G. Gouda and Tommy M. McGuire. Accelerated heartbeat protocols. In *Proc. of ICDCS'98*, pp. 202–209, IEEE, 1998.

[GM00] Mohamed G. Gouda and Tommy M. McGuire. Alert communication primitives above TCP. *J. High Speed Netw.*, 9(2):139–150, 2000.

[GMvWU06] Jan Friso Groote, Aad Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck van Weerdenburg Analysis of Distributed Systems with mCRL2. Chapter 4 of Michael Alexander and William Gardner Eds., Process Algebra for Parallel and Distributed Processing, pp. 99–128, CRC Press, 2009.

[HSC95] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *SIGCOMM Comput. Commun. Rev.*, 25(4):328–341, 1995.

[MG04] Tommy M. McGuire and Mohamed G. Gouda. *The Austin Protocol Compiler.* Springer, 2004.

[TSLC02] Y. Ting, F. M. Shan, W. B. Lu, and C. H. Chen. Implementation and evaluation of failsafe computer-controlled systems. *Comput. Ind. Eng.*, 42(2-4):401–415, 2002.

[Vog96] Werner Vogels. World wide failures. In *Proc of ACM SIGOPS European Workshop*, pp. 115–120. ACM, 1996.

[WGZC05] Guojun Wang, Zhongshan Gao, Lifan Zhang, and Jiannong Cao. Prediction-based multicast mobility management in mobile internet. In *Proc. of ISPA'05*, vol. 3758 of *LNCS*, pp. 1024–1035. Springer, 2005.

[KPW97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *J. International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[GPU01] Jan Friso Groote, Alban Ponse and Yaroslav S. Usenko Linearization in parallel pCRL. *J. Log. Algebr. Program.*, 48(1-2):39–70, 2001.