Formal Verification of Unreliable Failure Detectors in Partially Synchronous Systems

M. Atif
TU/Eindhoven
Dept. of Computer Science
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
m.atif@tue.nl

M.R. Mousavi
TU/Eindhoven
Dept. of Computer Science
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
m.r.mousavi@tue.nl

A. Osaiweran
TU/Eindhoven
Dept. of Computer Science
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
a.a.h.osaiweran@tue.nl

ABSTRACT

We formally verify four algorithms proposed in [M. Larrea, S. Arévalo and A. Fernández, Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems, 1999]. Each algorithm is specified as a network of timed automata and is verified with respect to completeness and accuracy properties. Using the model-checking tool UPPAAL, we detect and report the occurrences of deadlock (for all algorithms) between each pair of non-faulty nodes due to buffer overflow in communication channels with arbitrarily large buffers and we propose a solution. Moreover, we use one of the algorithms as a measure to compare three model-checking tools, namely, UPPAAL, mCRL2 and FDR2.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification; C.2.4 [**Distributed Systems**]

General Terms

Verification, Reliability

Keywords

Distributed Algorithms, Formal Verification, Failure Detectors, Model Checking

1. INTRODUCTION

Distributed systems are vulnerable to faults such as a crash of the participating processes or the communication media among them. A key challenge is to design distributed failure detectors that allow processes to distinguish slow processes from those which have crashed. It is important that these detectors are accurate, i.e., do not suspect correct processes, and complete, i.e., do suspect crashed ones. Given their non-trivial design, it is highly desirable to validate that these protocols satisfy their required or claimed properties. M. Larrea et al. introduce "efficient algorithms to implement failure detectors in partially synchronous systems" in [6], whose formal verification forms the subject matter of this paper.

1.1 Types of unreliable failure detectors

A failure detector is called unreliable, if it can mistakenly report a correct process (a process that remains operational during the protocol) as faulty (also known as suspected or crashed). Chandra and Toueg proposed unreliable failure detectors in [2] to guarantee two essential properties, namely, *completeness* and *accuracy*. *Completeness* is about suspecting each faulty process and *accuracy* concerns not suspecting any correct process. These properties are further classified into weak and strong as follows:

1 *Strong completeness*: Eventually every faulty process is permanently suspected by *every* non-faulty process.
2 *Weak completeness*: Eventually every faulty process is permanently suspected by *some* non-faulty process.
3 *Eventual strong accuracy*: Eventually no correct process is suspected by any correct process.
4 *Eventual weak accuracy*: Eventually *some* correct process is not suspected by any correct process.

1.2 Partial synchrony

In distributed systems, upper bounds on message delivery times (across communication channels) and message processing times play an important role in fault detection. For example, it is impossible to distinguish a slow process from a faulty one when there are no such upper bounds, i.e., in a totally asynchronous system [4]. In [2], a system is designated as *partially synchronous*, if there exist upper bounds for message delivery; such upper bounds are assumed to be unknown and hold only after an unknown stabilization interval. It is assumed that after the stabilization interval, every sent message is eventually received within the upper bound on the channel and process delays, provided that their communication channel is up and both the sender and the receiver are correct. The protocols described in [6] and analyzed here are supposed to guarantee their properties only in the partially synchronous setting.

1.3 Purpose of the study

The purpose of this study is to specify and formally verify the algorithms given in [6] using the formal verification tool UPPAAL [11]. Each algorithm includes a number of participants which have symmetric behavior and, as will be demonstrated below, this allows us to exploit symmetry reduction [5] supplied by UPPAAL to overcome the state-space explosion problem.

The results of our verification show that all algorithms of [6] contain a deadlock if there is a bounded (yet arbitrarily large) buffer in the communication channel between a pair of participants. We implement a fix for this problem and show that in the fixed setting, the other claimed properties regarding accuracy and completeness are indeed satisfied by the respective algorithms.

We also used the first algorithm in [6] as a case study to compare the performance of three model-checking tools, namely, UPPAAL, mCRL2 [8] and FDR2 [3]. Our case study shows that UPPAAL is best suited for larger state space, e.g., 700 millions states and more, whereas for relatively smaller state spaces, e.g., 300 millions or less, the performance of FDR2 outperforms the other two. The performance of mCRL2 remains closer to UPPAAL than FDR2 in both cases.

Structure of the paper. The algorithms under study are presented informally in Section 2 and formally in Section 3. In Section 4 we describe the functional requirements of the algorithms and Section 5 is devoted to the results.

2. ALGORITHMS

For fault detection, all the participants of the algorithms in [6] make a logical ring and every participant monitors its successor, called its *target*. This is achieved by sending periodic messages of the form "ARE-YOU-ALIVE?" and expecting timely response of the form "I-AM-ALIVE". If the target is unresponsive then it is suspected and the successor of the current target becomes the new target. Otherwise, if the target replies "I-AM-ALIVE" in time then it is pinged again after a period of Δ, which is a waiting time specific to that target. Each algorithm has two tasks, *Task1* and *Task2*, where the former is responsible for sending "ARE-YOU-ALIVE?" messages and the latter receives both "I-AM-ALIVE" from the successors and "ARE-YOU-ALIVE?" messages from the predecessors. Upon receiving "ARE-YOU-ALIVE?", *Task2* immediately replies with "I-AM-ALIVE".

2.1 Assumptions

The family of algorithms presented in [6] and analyzed in this paper are based on the following assumptions.

1. After stabilization, communication channels between any two processes are reliable, i.e., no message is lost.
2. A crashed process is permanently halted.
3. Π is a set of n processes or participants and every process is aware of the formation of the initial logical ring. Members of Π are fixed and hence, no process can join the protocol.
4. For fault detection, one process monitors at most one process at a time.
5. Every process is correct at the start and initially does not suspect any other process.
6. All the participants have symmetric behavior.
7. A process does not send any message to itself.
8. A message sent later can reach the destination earlier than a message sent earlier to the same destination.
9. The initial waiting time (Δ), the period in between each two rounds of monitoring for every process, is fixed and a priori known to each participant. For example if a process p monitors another process q, then $\Delta_{p,q}$ denotes the time interval for which p has to wait for the reply from q.

In the remainder of this section, we briefly explain the 4 algorithms proposed in [6].

2.2 An algorithm for weak completeness

This algorithm, given in Figure 1, forms the basis for the other algorithms in [6]. As mentioned at the outset of this section, the functionality of the process p is divided into two concurrent tasks; *Task1* is in charge of sending out "ARE-YOU-ALIVE?" messages and suspecting processes that have not replied within a certain time and *Task2* is in charge of receiving messages and processing (responding to them), if needed. *Task1* waits for the mutex and sends an "ARE-YOU-ALIVE?" message to the current target and signals the mutex. Subsequently, *Task1* sets the variable *received* to *false* and waits for its toggling by *Task2*. *Task1* waits for a fixed amount of time (initially set to the corresponding Δ for its target), and if it does not receive a response after the timeout, it suspects its target and moves to monitor the successor of its current target. *Task2* sets the variable *received* to *true* upon receiving any message either from the current target or from any of the already suspected processes. Upon receiving a message from a process q suspected by another process p, the process(es) in $\{q, \ldots, pred(target_p)\}$ are no more suspected by p and then q becomes the next target. All the messages of the type "I-AM-ALIVE" are discarded if they are neither from the current target, nor from the suspects.

2.3 An algorithm for eventual weak accuracy

This algorithm, given in Figure 2, is an extension of the algorithm presented in Section 2.2. To provide weak accuracy, the waiting time is adjusted according to the response time of a particular process which is supposed to be correct. Such a process is called *leader*. In the initialization phase of the protocol, an arbitrary process is named as *initial-cand* (initial candidate) to become the leader. Eventually the leader is either *initial-cand* or its immediate correct successor. If some process p is unresponsive to an "ARE-YOU-ALIVE?" message and $initial\_cand \in \{succ(p), \ldots, target_p\}$ then the waiting time for the current target is incremented by one unit of time, i.e., p increments its timeout value $\Delta_{p,target_p}$.

2.4 An algorithm for strong accuracy

This algorithm, given in Figure 3, is also an extension to the basic algorithm given in Section 2.2. In this algorithm, there is no leader; hence, each process increases the timeout value for its target when suspected. Using such a scheme, each process makes the timeout value sufficiently large so that it eventually stops suspecting its correct target.

2.5 An algorithm for strong completeness

In this algorithm, given in Figure 4, each participant p maintains a global list G_p of suspected processes along with its local view L_p of suspected processes (the former is particular to this algorithm, while the latter is common to all algorithms). Upon sending and receiving each message of

```
PROCESS(p)

$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

cobegin
|| Task1:
 loop
  wait($mutex_p$)
  send ARE-YOU-ALIVE? to $target_p$
  $t_{out} \leftarrow \Delta_{p,target_p}$
  $received \leftarrow false$
  signal($mutex_p$)
  delay $t_{out}$
  wait($mutex_p$)
  if not $received$
   $L_p \leftarrow L_p \cup \{target_p\}$
   $target_p \leftarrow succ(target_p)$
  end if
  signal($mutex_p$)
 end loop

|| Task2:
 loop
 receive message $m$ from a process $q$
 wait($mutex_p$)
 case
  $m$=ARE-YOU-ALIVE?:
   send I-AM-ALIVE to $q$
   if $q \in L_p$
  $L_p \leftarrow L_p - \{q, \ldots, pred(target_p)\}$
    $target_p \leftarrow q$
    $received \leftarrow true$
   end if
  $m$ = I-AM-ALIVE:
   case
    $q = target_p$:
     $received \leftarrow true$
    $q \in L_p$:
  $L_p \leftarrow L_p - \{q, \ldots, pred(target_p)\}$
     $target_p \leftarrow q$
     $received \leftarrow true$
    else discard $m$
   end case
  end case
  signal ($mutex_p$)
 end loop
coend
```

Figure 1: Weak completeness [6]

```
PROCESS(p)

$initial\_cand_p \leftarrow$ pre-agreed process
$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

cobegin
|| Task1:
 loop
  wait($mutex_p$)
  send ARE-YOU-ALIVE? to $target_p$
  $t_{out} \leftarrow \Delta_{p,target_p}$
  $received \leftarrow false$
  signal($mutex_p$)
  delay $t_{out}$
  wait($mutex_p$)
  if not $received$
  if  $initial\_cand_p \in$
             $\{succ(p), \ldots, target_p\}$
     $\Delta_{p,target_p} \leftarrow \Delta_{p,target_p} + 1$
   $L_p \leftarrow L_p \cup \{target_p\}$
   $target_p \leftarrow succ(target_p)$
  end if
  signal($mutex_p$)
 end loop

|| Task2:
 ...  { Same as algorithm in Fig. 1}
coend
```

Figure 2: Weak accuracy [6]

```
PROCESS(p)

$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

cobegin
|| Task1:
 loop
  wait($mutex_p$)
  send ARE-YOU-ALIVE? to $target_p$
  $t_{out} \leftarrow \Delta_{p,target_p}$
  $received \leftarrow false$
  signal($mutex_p$)
  delay $t_{out}$
  wait($mutex_p$)
  if not $received$
   $\Delta_{p,target_p} \leftarrow \Delta_{p,target_p} + 1$
   $L_p \leftarrow L_p \cup \{target_p\}$
   $target_p \leftarrow succ(target_p)$
  end if
  signal($mutex_p$)
 end loop

|| Task2:
 ...  { Same as task 2 of Fig. 1}
coend
```

Figure 3: Strong accuracy [6]

```
PROCESS(p)

{if the algorithm needs it:
$initial\_cand_p \leftarrow$ pre-agreed process}
$target_p \leftarrow succ(p)$
$L_p \leftarrow \emptyset$
$G_p \leftarrow \emptyset$
$\forall q \in \Pi : \Delta_{p,q} \leftarrow$ default timeout

cobegin
|| Task1:
 loop
  wait($mutex_p$)
  send ARE-YOU-ALIVE? to $target_p$
      —with $G_P$ — to $target_p$
  $t_{out} \leftarrow \Delta_{p,target_p}$
  $received \leftarrow false$
  signal($mutex_p$)
  delay $t_{out}$
  wait($mutex_p$)
  if not $received$
   {Update $\Delta_p, target_p$ if required}
   $G_p \leftarrow G_p \cup \{target_p\}$
   $L_p \leftarrow L_p \cup \{target_p\}$
   $target_p \leftarrow succ(target_p)$
  end if
  signal($mutex_p$)
 end loop

|| Task2:
 loop
  receive message $m$ from a process $q$
  wait($mutex_p$)
  case
   $m$=ARE-YOU-ALIVE?:
    send I-AM-ALIVE to $q$
    if $q \in L_p$
     $L_p \leftarrow L_p - \{q, \ldots,$
               $pred(target_p)\}$
     $target_p \leftarrow q$
     $received \leftarrow true$
    end if
    $G_p \leftarrow G_p \cup L_p - \{p, q\}$
   $m$ = I-AM-ALIVE:
    case
     $q = target_p$:
      $received \leftarrow true$
     $q \in L_p$:
      $L_p \leftarrow L_p - \{q, \ldots,$
               $pred(target_p)\}$
      $G_p \leftarrow G_p - \{q\}$
      $target_p \leftarrow q$
      $received \leftarrow true$
     else discard $m$
    end case
   end case
   signal ($mutex_p$)
  end loop
coend
```

Figure 4: Strong completeness [6]

types “ARE-YOU-ALIVE?” and “I-AM-ALIVE”, the global list is sent along and is updated, respectively, i.e., suspected processes are added while the correct ones are removed. In this way, eventually all crashed processes will be aggregated in list G and correct processes will be removed from G, realizing the goals of the algorithm.

3. FORMAL SPECIFICATION IN UPPAAL

We specify *Task1*, *Task2*, the communication channels and the monitor processes (explained in Section 5) in terms of timed automata in UPPAAL [7]. Timed automata as used in UPPAAL are extensions of finite state machines with real-valued clocks, as well as features such as invariants (in states), guards (on transitions) and synchronization (denoted by ! for sending and ? for receiving messages, respectively). We refer to [7] for further details. Parallel composition of these timed automata forms the system model. To alleviate the state-space explosion problem, we apply symmetry reduction [5] to our models, because all participants have symmetrical behavior. To this end, we exploit scalar set to specify this symmetric behavior as shown in Figure 5. In this figure, *initAll* is a function for assigning default values (to global declarations) and forming a logical ring of participants. There is a twist in our use of scalar sets [5], however, for specifying symmetry in our models. UPPAAL’s scalar sets are suitable for specifying fully symmetric structures; in order to specify symmetry in a ring, we need to identify the next process for each process while not exposing the exact identity of the process, to prevent breaking symmetry. This is the main technical difficulty dealt with in Figure 5. The loops (of type *for*) in the beginning of the *initAll* function are used to make the elements of Π dissimilar to each other.

```
// Global declarations
typedef scalar[3] id_t; // type declaration
id_t p0, p1, p2; // process identifiers
bool mutex[id_t]; // the mutex of each process
id_t target[id_t]; // target of each process
bool L[id_t][id_t]; // list of suspects for each process
void initAll(){
    //To assign p2 a different value from p0
    for (i:id_t ){
      if (i!=p0)
      p2=i;
    }
    // To assign p1 a different value from p0 and p2
    for (i:id_t ) {
       if (i!=p0 && i!=p2)
         p1=i;
       }
    // To form a logical ring of participants
    target[p0]=p1;
    target[p1]=p2;
    target[p2]=p0;
    //To assign default value to the mutex of every participant
    mutex[p0]=mutex[p1]=mutex[p2]=true;
    // initialization of L
    for (i:id_t)
       for (j:id_t)
         L[i][j]=false;
}
// initAll ends
```

Figure 5: Ring Symmetry in UPPAAL

In the following sections we discuss the formal specification of processes for each algorithm.

3.1 Weak completeness

3.1.1 Task1

The automaton for *Task1* is shown in Figure 6 where p is the identifier of the process executing *Task1.* In the initial state, *Task1* waits for the mutex and upon its availability, it assigns $target_p$ to $MyTarget$ (a temporary variable). This temporary variable is used because $target_p$ can change, while $MyTarget$ remains constant throughout each given round. Then at the *wait* state, *Task1* synchronizes with the channels dedicated to $target_p$ and sends an “ARE-YOU-ALIVE?” message. During this synchronization, the mutex is signaled (i.e., released) and $received_p$ is reset in accordance with the discussion in Section 2.2. At the *delay* state, *Task1* either notices non-deterministically the receipt of an “I-AM-ALIVE” message by *Task2* or starts suspecting the current target after reaching the *noReply* state. The function *succ* computes the successor of the current target. The process may crash at any state as shown in Figure 6. We assume that a crashed process can only receive messages but will not respond to them; the former assumption is essential, because otherwise messages to crashed processes would not be removed from the channels.

In this protocol, UPPAAL’s built-in support for time is not used, because time plays a role only in determining whether a received message is in time or not (i.e., in distinguishing no reply from late reply), which we model as a non-deterministic choice at the state *delay* in Figure 6.

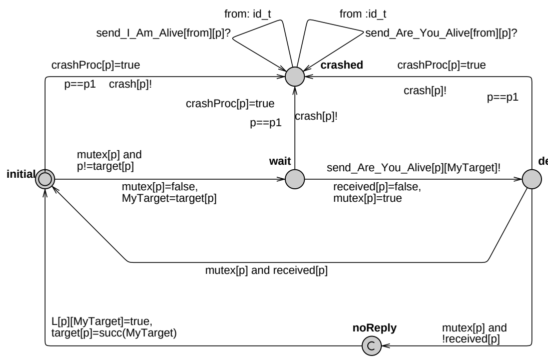

Figure 6: *Task1* of weak completeness algorithm

In our modeling, only one process is allowed to crash, which without loss of generality can be called p_1 (note that p_1 is not a particular identifier, but is just one of the scalars used in the ring). Hence, every transition going towards the *crashed* state is guarded with p_1. Crash of *Task1* is synchronized with *Task2* to halt both tasks at the same time.

3.1.2 Task2

Task2 receives either an “ARE-YOU-ALIVE?” (called message type 0) or an “I-AM-ALIVE” (called message type 1) from some process q at its initial state and then waits for the mutex at the *wait* state as shown in Figure 7. If the mutex is available it reaches the *case* state where, according to the message type, it either replies by sending an “I-AM-ALIVE” message or updates the suspicion status for process q. If process q is suspected by a process p, i.e., L[p][q] is *true* then all the processes in $\{q, \ldots, pred(target_p)\}$ are removed from the list of p's suspects (using *stopSuspect* function) and its *received* variable is set to *true* as shown at the *stopSus* state in Figure 7. A message of type "I-AM-ALIVE" is discarded if it is neither from the current target nor from an already suspected process.

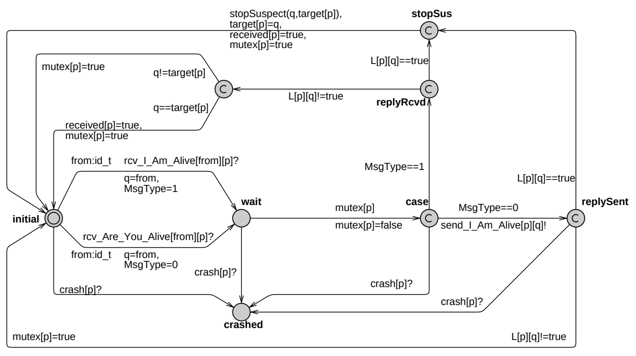

Figure 7: *Task2* of weak completeness algorithm

3.1.3 Communication channels

There are two communication channels between each pair of processes for the messages "ARE-YOU-ALIVE?" and "I-AM-ALIVE". Each channel has a limited buffer size, globally defined for all channels in the system. Source and destination processes are denoted by *from* and *to*, respectively as shown in Figure 8. Upon receiving a message every channel increases its local counter, i.e., *msgCounter* and decreases when a message is delivered. Channels can receive messages until *msgCounter* reaches the maximum buffer-size (denoted by *BufferSize*). A message is delivered only if there exists some message in the buffer of that channel.

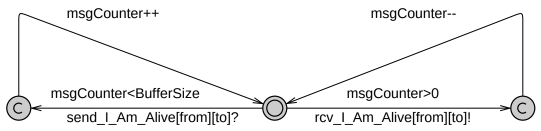

Figure 8: Channel process specific to *I-AM-ALIVE*

An identical channel is used for the messages of type "ARE-YOU-ALIVE?" for each process.

3.2 Weak accuracy

We model this protocol after its stabilization phase, i.e., when there is an upper bound on the maximum round-trip delay for messages both in the channels and processes, denoted by *maxDelta*.

3.2.1 Task1

The process for *Task1* in this protocol is similar to the one discussed in Section 3.1.1. The variable p is the identifier of the process executing this task. However, unlike in Section 3.1.1, here we do use the built-in support of UPPAAL for clocks. Every process uses a separate clock for each of its target processes. When sending an "ARE-YOU-ALIVE?" message, the variable *t_out* (for timeout) and the clock linked to the current target are initialized. At the *delay* state, delay is exactly up to *t_out*. This is why all outgoing edges are guarded with $waiting[p][MyTarget] = t\_out$, except for those modeling process crashes.

After a timeout, if the received flag is not marked as *true* by *Task2* then *Task1* reaches a state, named as *noReply* where it is determined whether the *initial_cand* belongs to $\{succ(p), \ldots, target_p\}$ or not as shown in Figure 9. If the *initial_cand* is in the aforementioned set, then Δ for the current target is incremented by 1 unit of time (provided that $\Delta < maxDelta$).

The other two edges from the *delay* states are for going to the *initial* state after a timeout, when either a reply has been received or the target is crashed.

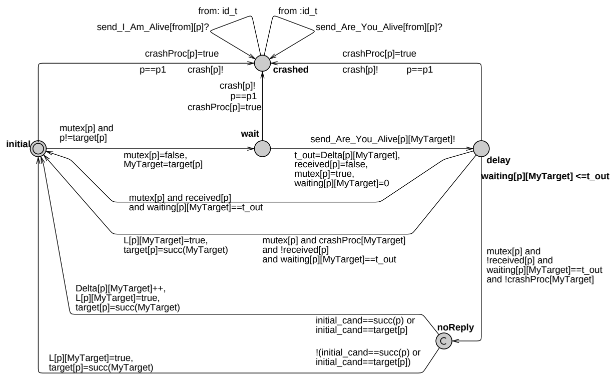

Figure 9: Timed-automata for *Task1* in the algorithm that provides weak accuracy

3.2.2 Task2

Task2 is exactly the same as the corresponding task discussed in Section 3.1.2 except for the added invariant to make sure that the processing time remains within *maxDelta*. This invariant checks the amount of time spent for a received message so that in the remaining time (maximum delay=*maxDelta*) the received message is processed.

3.3 Strong accuracy

For this algorithm, *Task1* discussed in Section 3.2.1 is slightly modified while *Task2* remains intact. The only difference is at the *noReply* state in Figure 9. There is only one outgoing transition from the *noReply* state to the initial state. This transition has no guard and updates $\Delta_{p,target_p}$, L_p and $target_p$ as follows:

- $\Delta_{p,target_p} = \Delta_{p,target_p} + 1$,
- $L[p][MyTarget] = true$, and
- $target_p = succ(MyTarget)$.

3.4 Strong completeness

In the specification of this protocol, we declare a global list G for all suspected processes. Hence, *Task1* of each process adds its target to G if the target is suspected and *Task2* removes the process if a process from this list communicates with its monitoring process. *Task1* discussed in Section 3.3 is modified to add the target in G to the only outgoing transition from the *noReply* state. There is no other change in *Task1*.

Task2 is also slightly modified by adding the list G, i.e., if an already suspected process q sends an "I-AM-ALIVE" message to a process p, then p removes q from G and likewise, if the received message is of the type "ARE-YOU-ALIVE?" then both p and q are removed from G.

4. GENERAL REQUIREMENTS

The algorithms to implement unreliable failure detectors in partially synchronous systems given in [6] are supposed to satisfy the following requirements.

1. *Deadlock freedom*: There must not be a deadlock in any protocol provided that at least two processes are correct.
2. *Weak completeness*: For the protocol discussed in Section 2.2.
3. *Eventual weak accuracy*: For the protocol discussed in Section 2.3.
4. *Strong accuracy*: For the protocol discussed in Section 2.4.
5. *Strong completeness*: For the protocol discussed in Section 2.5.

5. RESULTS

In this section, we report on our analysis results for all four algorithms presented earlier in this paper. For each algorithm, we first discuss the result of deadlock checking in UPPAAL. In order to compare the effectiveness of UPPAAL, we compare its performance with two other model-checking tools, namely, FDR2 [9, 10] and mCRL2[8]. Then, we propose a slight correction of the algorithms to remove the detected deadlock. Finally, we report on the verification of other properties on the corrected algorithms.

5.1 Results for weak completeness

5.1.1 Detecting deadlocks in UPPAAL

In UPPAAL, we specify the absence of deadlock throughout the state space by the following formula:

$$A[]\ \textit{not deadlock}$$

We have used client and server components of UPPAAL 4.1.4 (64 bit, release July 11, 2011) on different machines, i.e., a client on a Windows-based machine and the server on a Unix-based server machine (4 × 2.5 Ghz processor and 64 GB RAM).

To express eventuality while not breaking symmetry, we devise monitor processes for liveness properties, which we discuss in detail in the following sections. In the remainder of this section, we assume $\Pi = \{p_0, p_1, p_2\}$ and p_0, p_1, and p_2, respectively, form a logical ring.

Figure 10 shows a counter-example where a finite buffer (of an arbitrary size) overflows and as a consequence the protocol encounters a deadlock. Particularly, the buffer used to store "ARE-YOU-ALIVE?" messages overflows due to sending more "ARE-YOU-ALIVE?" messages and receiving less "I-AM-ALIVE" messages. In this deadlock scenario, the process p_2 sends "ARE-YOU-ALIVE?" to its target p_0 and after a timeout suspects p_0. Then p_2 receives "ARE-YOU-ALIVE?" from p_0, replies with "I-AM-ALIVE" and stops suspecting p_0. *Task2* of p_2 receives "I-AM-ALIVE" but at that time the mutex is taken by *Task1* which sends "ARE-YOU-ALIVE?" to p_0 and releases the mutex. *Task2* takes the mutex and processes the recently received "I-AM-ALIVE" considering it the reply of the last polling. Up to this point, the process p_2 has sent two "ARE-YOU-ALIVE?" messages and received only one reply whereas it is not waiting for any further reply. Rather it is going to send another "ARE-YOU-ALIVE?" for p_0. So, due to repeating the above message sequence at p_2's end, the buffer of size n overflows on $n + 1$ iterations as shown in Figure 10. When the buffer is full, *Task1* cannot synchronize with the channel after holding the mutex, and due to the unavailability of the mutex, *Task2* is also halted, which results in a deadlock for process p_2. Process p_1 has already crashed, hence the protocol faces a general deadlock.

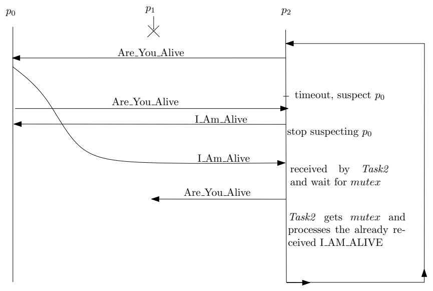

Figure 10: Example of buffer overflow

5.1.2 Detecting deadlocks in FDR2 and mCRL2

Besides UPPAAL, we model checked the algorithm that provides weak completeness in both FDR2 and mCRL2, and came up with the same counterexample shown in Figure 10 for the occurrence of a deadlock due to buffer overflow. We refer to [1] for the complete formal specifications in mCRL2 and CSP. The reason for modeling this algorithm in FDR2 and mCRL2 is to compare the performance of the three model checkers, i.e., UPPAAL, FDR2 and mCRL2. The reason for not modeling other algorithms in FDR2 and mCRL2 is that built-in support of time is available only in UPPAAL and time-based events in the other three algorithms play a crucial role in their functionality (see sections 2.3, 2.4 and 2.5).

To fix this deadlock, one solution is to use FIFO channels (instead of arbitrary channels allowing for message overtaking), and another solution is to ignore all incoming messages to a channel when its buffer is full.[1] We categorize the behavior of a channel as follows when its message buffer is full.

1. *Channel type* B: Sender waits until there is space for one message.
2. *Channel type* R: Full channel reports "error" message when another message is received.
3. *Channel type* I: Full channel receives and ignores further messages.

In Table 1, we give the results with respect to channel types R and B.

We performed model-checking on a server machine having 32 × 16 × 2 Ghz processor and 32 × 12 GB memory.

[1]In a recent personal communication, the first author of [6] suggested another possible fix, namely to remove lines 7 to 11 from Task 2 in Figure 1. We could formally verify that this fix is deadlock free, if we also assume that messages to crashed processes are always lost.

Tool	Buffer Size	Channel type	time	state-space	states/sec
mCRL2	1	B	44sec	588641	13378
		R	0m5.9	6630	1105
	2	B	42m43	30182443	11776
		R	33sec	314951	9544
UPPAAL	1	B	52sec	2431674	46762
		R	0sec	46608	–
	2	B	71m8	184493193	43227
		R	1m17	5654365	73433
FDR2	1	B	1sec	167388	167388
		R	0sec	2401	–
	2	B	1m10	6230100	89001
		R	0sec	83437	–

Table 1: Comparison of mCRL2, UPPAAL and FDR2

For channel type I, there is no deadlock regardless of buffer size. For buffer size 1, UPPAAL explored 711410029 states in 404 minutes, FDR2 explored 385861073 states in 887 minutes and mCRL2 explored 168491893 states in 451 minutes. Thus, for this case study, UPPAAL is the most effective for larger state spaces, while for smaller state spaces, i.e., for the case of channel types B and R, FDR2 takes the lead as shown in Table 1.

In the remainder of this paper, we first remove the above-mentioned deadlock, using a bounded FIFO channel and then proceed to verify the functional properties of the protocols. (The authors of [6] indicated in a personal communication that general channels with the possibility of overtaking are the ones used when designing the protocol, but as demonstrated above this assumption appears to be too general for the protocol to work correctly.)

5.1.3 Weak completeness

To verify weak completeness for the algorithm discussed in Section 2.2, we devised a monitor process shown in Figure 11. This process moves to the *error* state when the system oscillates more than a specified number of times (e.g., three times in Figure 11) between suspecting and not suspecting an already crashed process by its correct predecessor. (A more general monitor can be constructed by counting the number of oscillations and checking it against a fixed constant as the guard for the transition to the state labeled *error*.) The initial state is marked as committed to give it a higher priority over functional steps of the protocol, because using the *initAll* function, global declarations are initialized and a logical ring of the participants is formed.

After verification, it turns out that the monitor process does detect a counterexample if the number of oscillations is set to one or two. We depict the counter-examples in Figure 12. According to [6], suspecting a crashed process by its correct predecessor must be permanent and hence the reported counter-examples apparently do violate the intuition stated in [6].

The property of *weak completeness* is satisfied, i.e., eventually a crashed process is suspected by its correct predecessor, but the proof of Theorem 1 in [6] does not appear to be correct; it is claimed there:

$$\exists t_0 \ : \ \forall p \in \mathit{crashed}, p \ \mathit{has\ failed\ at\ time}\ t_0 \ \mathit{and}\ \forall t \geq t_0, p \in L_{corr\_pred(p)}(t).$$

A counter-example to this claim is shown in the message-

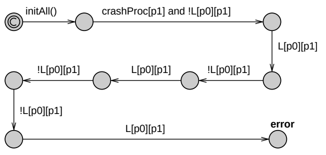

Figure 11: A monitor to check weak completeness

sequence charts of Figure 12, where (a) shows the scenario given as the proof of Theorem 1 while (b) and (c) depict the counterexamples to this proof, i.e., exclusion of a crashed process from the list of suspects. Although this exclusion is eventually stopped, oscillating between suspecting and not-suspecting a crashed process is not addressed in the proof.

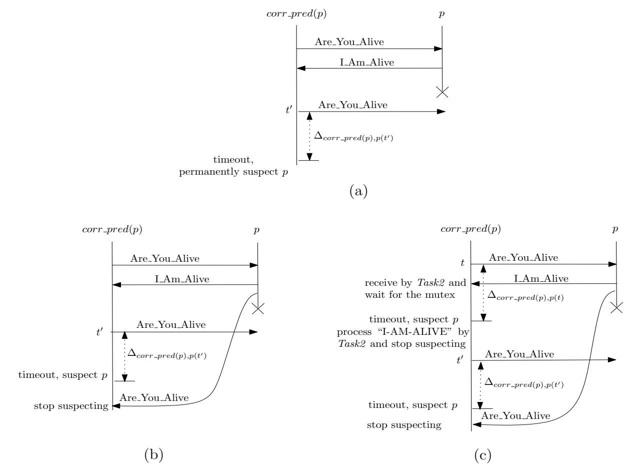

Figure 12: Counterexamples contradicting Theorem 1 given in [6]

In Figure 12, a correct predecessor of a process p sends an "ARE-YOU-ALIVE?" message to p and receives its reply after which p crashes. At time t', another "ARE-YOU-ALIVE?" message is sent and because of p's crash failure, it is assumed that there will be no further message from p. Hence, p is permanently suspected as shown in Figure 12(a) after $\Delta_{correct\_pred(p),p(t')}$. However, Figure 12(b) shows receiving of a "ARE-YOU-ALIVE?" message (late, due to unbounded delay in channels) from p which causes it to stop suspecting process p. Figure 12(c) shows a different situation when *Task2* of *corr_prd(p)* receives a "I-AM-ALIVE" message and waits for the mutex but at the same time, timeout occurs at *Task1* which adds the process p in suspects and releases the mutex. *Task2* takes the mutex and processes the reply, due to which p is again excluded from the list of suspects.

5.2 Results for weak accuracy

5.2.1 Deadlock

The counterexample shown in Figure 13 exhibits the deadlock scenario which is different from the one discussed in Section 5.1.1 but it resembles it in the sense that it is also due to not-suspecting by receiving an "ARE-YOU-ALIVE?" message when an "I-AM-ALIVE" message is expected. An explanation of reaching deadlock in process p_0 is given below.

1 Send "ARE-YOU-ALIVE?" to p_1, but p_1 is already crashed.
2 Suspect p_1 and change target to p_2.
3 Send "ARE-YOU-ALIVE?" to p_2.
4 Timeout and suspect p_2.
5 Receive "ARE-YOU-ALIVE?" from p_2.
6 Send "I-AM-ALIVE" and stop suspecting p_2.
7 Send "ARE-YOU-ALIVE?" to p_2.
8 *Task2* receives "ARE-YOU-ALIVE?" from p_2 and gets the mutex but cannot send "I-AM-ALIVE" because the same message (mentioned at step 6) is already there to be delivered. Now *Task2* continues to wait for a free channel while holding the mutex. Because of the mutex, *Task1* also stops and as a result the whole process p_0 is halted even though it is non-faulty. A similar reason for deadlock is there for p_2 as well, which is shown in Figure 13.

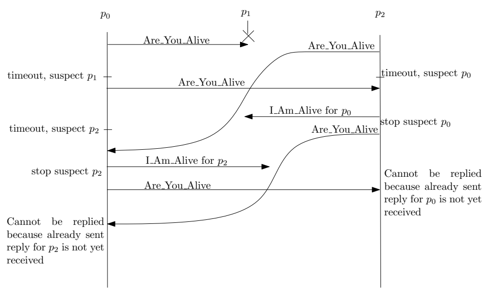

Figure 13: Message sequence chart to show deadlock

Again, the deadlock is removed when replacing the communication channel with a FIFO channel and we verify the rest of the properties in the fixed setting.

5.2.2 *Weak accuracy*

To verify weak accuracy, we devised a monitor process shown in Figure 14 and found that this property is satisfied. The initial state is marked committed, so that the first transition in the system model is by this monitor process which uses the *initAll* function to initialize global variables. Then, it takes the next transition only when the waiting time at some process p_0 for process p_2 reaches *maxDelta* whereas in the logical ring of processes p_0, p_1 and p_2 only p_1 can crash. So this monitor process reaches the *error* state when the process p_2 is correct, replying within *maxDelta* and its correct predecessor p_0 suspects it.

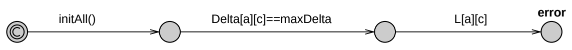

Figure 14: A monitor process for weak accuracy

5.3 Results for strong accuracy and strong completeness

We found the same deadlock reported in Section 5.2 for both of these algorithms but their concerning properties of *strong accuracy* and *strong completeness* are satisfied.

For *strong accuracy*, we devised a monitor process shown in Figure 15. In the logical ring of the processes p_0, p_1 and p_2, only p_1 is allowed to crash. In other words the processes p_0 and p_2 are correct and according to *strong accuracy* [6] they are supposed to be not suspected if they continue responding within a certain amount of time. So, the monitor process monitors the suspicion status of both p_0 and p_1 when the waiting time of one for the other is augmented to *maxDelta* but the other is still unresponsive. So, Δ becomes more than *maxDelta*, which is a violation of *strong accuracy* because the upper bound on the round-trip communication between each pair of processes is *maxDelta*. So, $\forall p, q \in Correct$ when $\Delta_{p,q}$ becomes greater than *maxDelta* then it causes reaching to the *error* state.

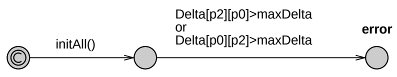

Figure 15: Monitor process for strong accuracy

For *strong completeness*, we devised the monitor process shown in Figure 16. As discussed before, only the process p_1 is allowed to crash. So the monitor process shown in Figure 16 reaches to *error* state when p_1 is crashed but not suspected (i.e., not part of the list G) while the other processes p_0 and p_2 have augmented their waiting time to *maxDelta*.

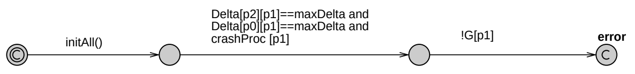

Figure 16: Monitor process for strong completeness

6. REFERENCES

[1] M. Atif, M. R. Mousavi, and A. Osaiweran. Formal verification of unreliable failure detectors in partially synchronous systems. Technical Report CSR-11-12, Dept. of Computer Science, TU/Eindhoven, 2011.
[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
[3] FDR homepage. *http://www.fsel.com.*
[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, 1985.
[5] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding symmetry reduction to UPPAAL. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.
[6] M. Larrea, S. Arévalo, and A. Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In P. Jayanti, editor, *DISC*, volume 1693 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 1999.
[7] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.
[8] mCRL2 toolset homepage. *http://www.mcrl2.org/.*
[9] B. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.
[10] P. Y. A. Ryan and S. A. Schneider. *The modelling and analysis of security protocols: the CSP approach.* Addison-Wesley Professional, first edition, 2000.
[11] UPPAAL homepage. *http://www.uppaal.org/.*