Formal Analysis of Consensus Protocols in Asynchronous Distributed Systems

Muhammad Atif

16th October 2009

Abstract

This paper presents a formal verification of two consensus protocols for distributed systems presented in [T. Deepak Chandra and S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM, 1996]. These two protocols rely on two underlying failure detection protocols. We formalize an abstract model of the underlying failure detection protocols and building upon this abstract model, formalize the two consensus protocols. We prove that both algorithms satisfy the properties of "uniform agreement", "uniform integrity", "termination" and "uniform validity" assuming the correctness of their corresponding failure detectors.

1 Introduction

In a consensus protocol, each participating process proposes a value and eventually all (non-crashed) processes should reach a state in which they decide upon the same value. The decided value has to be chosen from the set of proposed values by the participating processes [3]. In an asynchronous environment, there is no upper bound on the delay of (reliable) communication channels; hence, a process cannot distinguish between a crashed process, for whose proposed value it does not have to wait, and a process connected to a very slow communication channel, whose proposed value has to be taken into account in the final result of the consensus. This forms the basic argument behind the impossibility of solving the consensus problem in an asynchronous environment in the presence of crash failures [4].

To circumvent this problem, the consensus protocols are built upon failure detectors, which by a synchronization mechanism can provide us with information about crashed (i.e., permanently halted) and correct processes. Upon query at any given time, the failure detector of each process outputs the list of its suspected processes. The information provided by a failure detector is not necessarily accurate and hence, failure detectors can only *suspect* other processes. The unreliable failure detectors are in turn the

result of unbounded delays in the asynchronous communication channels. Hence, at each moment of time, the output of any two failure detectors can be different.

We formalize and verify two algorithms (also called protocols) for solving the consensus problem proposed by [1]; one uses strong completeness with weak accuracy and the other uses strong completeness with eventual weak accuracy. Strong completeness refers to suspecting all crashed processes, i.e., after a certain amount of time every correct process permanently suspects each crashed process. Weak accuracy means that some correct process is never suspected. Eventual weak accuracy means that after a certain amount of time, some correct process is never suspected. The first consensus protocol, relying on strongly complete and weakly accurate failure detectors, tolerates $N-1$ number of process-failures (N is the total number of processes in asynchronous systems) whereas the one, relying on a strongly complete and eventually weakly accurate failure detector, requires a majority of processes to be correct [1]. If the network guarantees the said number of processes to be correct, we prove that both consensus algorithms satisfy functional requirements of uniform agreement, uniform integrity, termination and uniform validity, to be defined precisely in the remainder of this report.

Structure of the paper. We give an informal description of two consensus protocols in Sections 2.2 and 2.3 and process-algebraic specifications of them in Sections 3.2 and 3.3, respectively. The requirements of the protocols and their results are presented in Section 4. The paper is concluded in Section 5.

2 Consensus Protocols

Consensus protocols ensure that all correct processes eventually reach a consensus on one value, called the decided value. The decided value is always selected from a set of values, to which every process (at the beginning of the protocol) contributes one value, called the proposed value, to this set. The process will not come to a decision if it fails by crashing, i.e., permanently halting. A failure pattern, denoted by F in the remaining text, is a function from $\mathfrak{T}$ to 2^{π} where $\mathfrak{T}$ is the set of natural numbers, denoting discrete time, and $\pi = \{p_1, p_2, \ldots, p_n\}$ is the set of participating processes. During the execution of the protocols, a failure detector D makes (possibly unreliable) information available about the failure pattern F. Next we explain the general assumptions on which the forthcoming algorithms rely.

2.1 General assumptions

1. If a process is crashed, it will never recover. Assume that $F(t)$ denotes the set of crashed processes up to time t then $F(t) \subseteq F(t+1)$.

2. All failure detectors are unreliable. This means that they can suspect correct processes or unsuspect crashed processes at any time. Hence, in general for each process p, $H(p,t)$ is unrelated to $H(p,t+1)$ where H is a function from $\pi \times \mathfrak{T}$ to 2^{π} for failure detector history and it provides the history of a failure detector D_p up to time t, i.e., a timed trace of lists of processes suspected by p_i up to time t. It is assumed that there is a discrete global clock that acts as a fictional device and the processes do not have access to it. Due to unreliability of failure detectors, it is also possible for two distinct processes p and q that $H(p,t) \neq H(q,t)$ at some time t.

3. A solution for the consensus problem is proposed in the setting of asynchronous distributed systems in which there is no upper bound on:

   (a) message delays,

   (b) clock drifts, and

   (c) the amount of time necessary to execute a step.

4. The failure detectors of all correct process participants satisfy *strong completeness*, i.e., eventually every crashed process is permanently suspected by their failure detectors. Due to [1], the following formula formalizes this description.

$$\forall F,\ \forall H \in D(F),\ \exists t \in \mathfrak{T},\ \forall p \in \mathit{crashed}(F),$$
$$\forall q \in \mathit{correct}(F), \forall t' \geq t : p \in H(q,t')$$

   $D(F)$ is a set of failure detector histories and $\mathit{correct}(F) = \pi - \mathit{crashed}(F)$ where $\mathit{crashed}(F) = \bigcup_{t \in \mathfrak{T}} F(t)$.

5. Although the failure detectors are unreliable, they are assumed to satisfy some notion of *accuracy*. A failure detector is *weakly accurate* when some correct process is never suspected; it is *eventually weakly accurate*, if it eventually never suspects some correct process. The following formula, due to [1], formalizes this description.

$$\forall F,\ \forall H \in D(F),\ \exists p \in \mathit{correct}(F),\ \forall t \in \mathfrak{T},\ \forall q \in \pi - F(t) : p \notin H(q,t)$$

6. The consensus algorithm that relies on strong completeness with weak accuracy can tolerate any number of process failures whereas the other consensus algorithm requiring strong completeness and eventual weak accuracy, requires the majority of the process to be correct.

7. The communication channel between each pair of processes is reliable.

Along with the property of strong completeness, the algorithms discussed in Sections 2.2 and 2.3 rely on the above assumptions together with the properties of weak accuracy and eventual weak accuracy, respectively.

2.2 Solving consensus using strong completeness and weak accuracy

This algorithm assumes the properties of strong completeness and weak accuracy and solves the consensus problem in an asynchronous system provided that at least one correct process is never suspected by any failure detector. The algorithm has three phases and each process, if it remains operational, is supposed to go through all phases (from the first to the last). Suppose that n is the total number of processes in the network. In the first phase, each (non-crashed) process p executes $n-1$ rounds. In every round each process broadcasts a message that contains its proposed value v_p and then receives the same type of message from other non-suspected processes. At the end of this phase, every process updates its set of proposed values. These values are obtained either directly from other processes or indirectly in that some processes are correct but erroneously suspected.
In the second phase, all correct processes exchange their sets of values and make them identical to each other by dropping values that are not part of some received set. In the third and last phase, each process decides the first available value in its set. The algorithm for solving the consensus problem using strong completeness and weak accuracy, due to [1], is given below such that every process p executes it with a distinct proposed value v_p.

Algorithm 1 Process(v_p)

$V_p := \langle \perp, \perp, \ldots, \perp \rangle$ {p's *estimate of the proposed values*}
$V_p[p] := v_p$
$\Delta_p := V_p$ {*To send/receive proposed values*}

Phase 1: {*Asynchronous rounds* r_p, $1 \leq r_p \leq n-1$}
for $r_p = 1$ to $n-1$ **do**
  send (r_p, Δ_p, p) *to all*
  wait until [$\forall q$: *received* (r_p, Δ_q, q) **or** $q \in D_p$] {*Query the failure detector and get* D_p, *i.e., a set of suspected processes. If* $q \notin D_p$ *then receive message from* q *for round* r_p}
  $msgs_p[r_p] := \{(r_p, \Delta_q, q) \mid$ *received* $(r_p, \Delta_q, q)\}$
  $\Delta_p := \langle \perp, \perp, \ldots, \perp \rangle$
  for $k = 1$ to n **do**
    if $V_p[k] = \perp$ **and** $\exists (r_p, \Delta_q, q) \in msgs_p[r_p]$ *with* $\Delta_q[k] \neq \perp$ **then**
      $V_p[k] := \Delta_q[k]$
      $\Delta_p[k] := \Delta_q[k]$
    end if
  end for
end for

Phase 2: *send* V_p *to all*
wait until [$\forall q$: *received* V_q **or** $q \in D_p$]
$lastmsgs_p := \{V_q \mid$ *received* $V_q\}$
for $k = 1$ to n **do**
  if $\exists V_q \in lastmsgs_p$ *with* $V_q[k] = \perp$ **then**
    $V_p[k] := \perp$
  end if
end for

Phase 3:
decide (first non-$\perp$ element of V_p)

2.3 Solving consensus using strong completeness and eventual weak accuracy

In the previous section, we gave the algorithm to solve consensus using strong completeness and weak accuracy where at least one process was supposed to be correct. Now we introduce the algorithm, proposed in [1], to solve the same problem with strong completeness and eventual weak accuracy. This algorithm demands a majority of processes to be correct. The protocol is executed in rounds and in each round, there is a unique coordinator, namely, the one with identifier $c = (r \ mod \ n) + 1$. If a process is correct, which may or may not be suspected, it eventually decides some value with the consent of the coordinator.

In every round there are four phases. In the first phase each process sends its proposed value (estimate) to the coordinator (timestamped with the round number). In the second phase, the coordinator receives the estimates from non-suspected processes and then selects one of them as their new

estimate. The selected value is the estimate of a process that has the largest timestamp. In the same phase, the coordinator broadcasts its estimate. In the third phase, processes receive the value sent by the coordinator and send back either *ack* (acknowledgement message) if the coordinator is not suspected or otherwise *nack* (no acknowledgement). In the fourth phase, the coordinator waits for $\lceil \frac{(n+1)}{2} \rceil$ replies and if all of them are of type *ack* then $estimate_c$ is locked, or otherwise it starts a new round and consequently other processes waiting for a decision also start a new round. The only reason to send a *nack* message (in Phase 3) is having suspicion (due to failure detector) for the coordinator. However, if all of the $\lceil \frac{(n+1)}{2} \rceil$ acknowledgements (*ack* type messages) are received, then the coordinator decides the locked value and broadcasts it through a channel, called *R-broadcast*. Every process p in this protocol executes the following algorithm [1] where the parameter v_p denotes the proposed value.

Algorithm 2 Process(v_p)

$estimate_p := v_p$ {$estimate_p$ *is estimated decision value of* p}
$state_p := undecided$
$r_p := 0$ {r_p *is* p*'s current round number*}
$ts_p := 0$ {ts_p *is the last round in which* p *updated* $estimate_p$}

{*Rotate through coordinators until decision is reached*}

while $state_p = undecided$ **do**
  $r_p := r_p + 1$
  $c_p := (r_p \bmod n) + 1$ {c_p *is the current coordinator*}

  Phase 1: {*All processes* p *send* $estimate_p$ *to the current coordinator*}
  send $(p, r_p, estimate_p, ts_p)$ *to* c_p

  Phase 2: {*The current coordinator gathers* $\lceil \frac{(n+1)}{2} \rceil$ *estimates and proposes a new estimate*}
  if $p = c_p$ **then**
    wait until [*for* $\lceil \frac{(n+1)}{2} \rceil$ *processes* q : *received* $(q, r_p, estimate_q, ts_q)$ *from* q]
    $msgs_p[r_p] := \{(q, r_p, estimate_q, ts_q) \mid p$ *received* $(q, r_p, estimate_q, ts_q)$ *from* $q\}$
    $t :=$ *largest* ts_q *such that* $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
    $estimate_p :=$ *select one* $estimate_q$ *such that* $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
    send $(p, r_p, estimate_p)$ *to all*
  end if

  Phase 3: {*All processes wait for the new estimate proposed by the current coordinator*}
  wait until [*received* $(c_p, r_p, estimate_{c_p})$ *from* c_p **or** $c_p \in D_p$]
  if [*received* $(c_p, r_p, estimate_{c_p})$ *from* c_p] **then** {p *received* $estimate_{c_p}$ *from* c_p}
    $estimate_p := estimate_{c_p}$
    $ts_p := r_p$
    send (p, r_p, ack) *to* c_p
  else
    send $(p, r_p, nack)$ *to* c_p {p *suspects that* c_p *crashed*}
  end if

  Phase 4: {*The current coordinator waits for* $\lceil \frac{(n+1)}{2} \rceil$ *replies. If they indicate that* $\lceil \frac{(n+1)}{2} \rceil$ *processes adopted its estimate, the coordinator R-broadcasts a decide message*}
  if $p = c_p$ **then**
    wait until [*for* $\lceil \frac{(n+1)}{2} \rceil$*processes* q : *received* (q, r_p, ack) **or** $(q, r_p, nack)$
    if [for $\lceil \frac{(n+1)}{2} \rceil$ *processes* q : *received* (q, r_p, ack)] **then**
      R-broadcast $(p, r_p, estimate_p, decide)$ {*reliable broadcast*}
    end if
  end if
end while

{*if* p *R-delivers a decide message,* p *decides accordingly*}

when *R-deliver* $(q, r_q, estimate_q, decide)$
if $state_p = undecided$ **then**
  decide $(estimate_q)$
  $state_p := decided$
end if

3 Formal Specification

In this section, we discuss the formalization of the consensus algorithms, given in Sections 2.2 and 2.3, respectively. We use mCRL2 [6] as our formal specification language. We need some data types, functions and operators to specify the behaviour of the protocols in terms of communication channels, failure detectors and the different phases of the protocols. In the formal specification of both algorithms, we use a separate channel for every type of message in every round to entertain asynchrony with respect to communication channels. So there is no bound on message delays and a message sent in a previous round can reach its destination after a message of the current round.

3.1 Data types

We use the built-in support for data types in mCRL2 like; $\mathbb{B}$ (for Boolean, i.e., *true* or *false*), $\mathbb{Z}$ (for integers) and $\mathbb{N}$ (for natural numbers). The toolset defines both $\mathbb{Z}$ and $\mathbb{N}$ as unbounded, i.e., there is no largest number in these data types (and no smallest for $\mathbb{Z}$). The toolset also provides many data structures, we use one of them, called *List*, to handle homogeneous data, e.g., *estimates*, *msgs*, *lastMsgs* etc.

3.2 Consensus with strong completeness and weak accuracy

Before discussing the formalization details of the protocol, we present all auxiliary functions, which are defined in the form of rewrite rules. Function types are used to define customized transformations on (a combination of) abstract data types. We define the following customized functions where keywords **map**, **var** and **eqn** in mCRL2 are used for function signature, variable declaration and function definition (in terms of equations), respectively.

- *minus:* To subtract a list from another, e.g., if A and B are two lists of natural numbers then $minus(A, B)$ is also a list having all such elements of A which do not belong to B. This definition is formally specified as:

  map
  $minus : List(\mathbb{N}) \times List(\mathbb{N}) \rightarrow List(\mathbb{N})$;
  $eliminate : List(\mathbb{N}) \times \mathbb{N} \rightarrow List(\mathbb{N})$;
  {to eliminate the first occurrence of a value from the list}
  var
  $ln, lg : List(\mathbb{N})$;
  $m, n : \mathbb{N}$;

eqn
$minus([], lg) = []$; $\quad$ {[] is an empty list}
$minus(ln, []) = ln$;
$minus(n \rhd ln, m \rhd lg) =$
$if(m \in n \rhd ln, minus(eliminate(n \rhd ln, m), lg), minus(n \rhd ln, lg))$;
{$\rhd$ is the operator to insert an element at the head of a list}
$eliminate(n \rhd ln, m) = if(n \approx m, ln, n \rhd eliminate(ln, m))$;

- *makeIdentical:* This function makes two lists (of the same size) identical by replacing every element that appears in one but not in the other with $\perp$ (used for null value) at each location. In Phase 2, processes exchange their lists of values and using this function make them identical.

map
$makeIdentical : List(\mathbb{N}) \times List(\mathbb{N}) \rightarrow List(\mathbb{N})$;
var
$ln : List(\mathbb{N})$;
$x, n : \mathbb{N}$;
eqn
$makeIdentical([], ln) = ln$;
$makeIdentical(ln, []) = []$;
$makeIdentical(x \rhd lg, n \rhd ln) =$
$if(x \approx \perp, \perp \rhd makeIdentical(lg, ln), n \rhd makeIdentical(lg, ln))$;

- *findDecided:* This function finds the first available non-$\perp$ value from a list. Each process uses this function in Phase 3 to decide a value.

map
$findDecided : List(\mathbb{N}) \rightarrow \mathbb{N}$;
var
$ln : List(\mathbb{N})$;
$n : \mathbb{N}$;
eqn
$findDecided([]) = \perp$;
$findDecided(n \rhd ln) = if(n \not\approx \perp, n, findDecided(ln))$;

- *updateDelta*: Δ is the list used in every round of Phase 1 to send the proposed value to all other processes. After sending Δ, each process initializes it with $\perp$ and then updates it with the values received in the current round but not in the previous rounds. To update the data values in this list, the function *updateDelta* is used. This function is only defined when the three lists have the same size.

map
$updateDelta : List(\mathbb{N}) \times List(\mathbb{N}) \times List(\mathbb{N}) \rightarrow List(\mathbb{N})$;
var
$lg, ln, ld : List(\mathbb{N})$;
$x, n, m : \mathbb{N}$;
eqn
$updateDelta([], lg, ln) = []$;
$updateDelta(n \rhd lg, m \rhd ln, x \rhd ld) =$
$if(m \not\approx n, m \rhd updateDelta(lg, ln, ld), x \rhd updateDelta(lg, ln, ld))$;

- *updateMsgs*: In phases 1 and 2 processes use two lists *msgs* and *lastmsgs* respectively to store the lists of other processes. This function helps the processes to store a list at a particular location.
  map
  $updateMsgs : \mathbb{N} \times List(List(\mathbb{N})) \times List(\mathbb{N}) \rightarrow List(List(\mathbb{N}))$
  var
  $lg, ln : List(\mathbb{N})$;
  $n : \mathbb{N}$;
  $msgs : List(List(\mathbb{N}))$;
  eqn
  $updateMsgs(\perp, lg \rhd msgs, ln) = ln \rhd msgs$;
  $updateMsgs(\perp, [], ln) = [ln]$;
  $(n > 0) \rightarrow updateMsgs(n, lg \rhd msgs, ln) =$
  $lg \rhd updateMsgs(Int2Nat(n-1), msgs, ln)$;
  {Int2Nat function determines the natural number of
  an integer value}

- *updateCrashed*: Failure detectors use this function to add a crashed process in the list of suspects.
  map
  $updateCrashed : List(\mathbb{N}) \times \mathbb{N} \rightarrow List(\mathbb{N})$;
  var
  $ln : List(\mathbb{N})$;
  $n : \mathbb{N}$
  eqn
  $updateCrashed(ln, n) = if(n \in ln, ln, n \rhd ln)$;

Next we discuss the process definitions which specify the behaviour of every participant in the protocol.

3.2.1 The process for failure detectors:

A failure detector provides a list of suspected processes whenever a process requires it. In [1], the behaviour of a failure detector is defined in terms of abstract properties. In accordance to these properties, we devise one process to represent the failure detectors of all processes as shown in Figure 1,

where the processes query the failure detector and get the list of suspects. To get the reduced state space, we instantiated this process once and allowed its interaction with other processes in the network where the processes also communicate with each other in different phases and rounds. This process

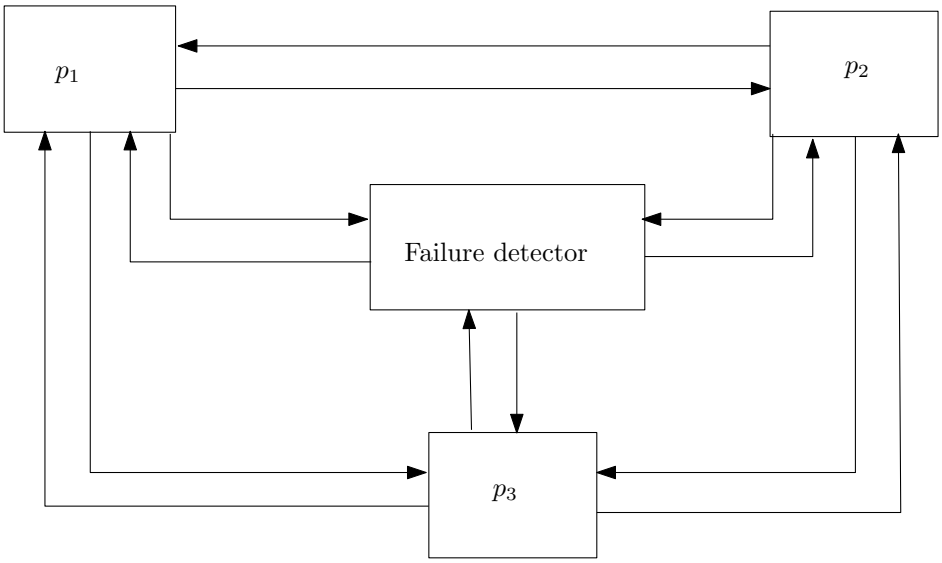

Figure 1: Failure detector used in the model for Algorithm 1, where $\pi = \{p_1, p_2, p_3\}$

eventually realizes the strong completeness property when a crashed process is permanently added in the list of suspects. Each process can query this process like communicating with the local failure detector. This failure detector is unreliable, so by mistake it can include correct processes (except one, when it satisfies weak accuracy) among the suspected processes. The property of weak accuracy is implemented in the process for Phase 1 (discussed in Section 3.2.2) to reduce the state space. Initially, it does not care about strong completeness but non-deterministically at any point (afterwards), it provides the complete list of crashed process. We define this process by means of a parameter, i.e., *crashed*:

- $crashed : List(\mathbb{N})$: The list of the crashed processes, i.e., sent as a reply to the querying process. In the start this list is empty but eventually it contains every crashed process.

1: $FD(crashed : List(\mathbb{N})) =$
2: $\sum_{id:\mathbb{N}} rcv_addRequest(id).FD(updateCrashed(crashed, id))$
3: $+$
4: $\sum_{p:\pi} send_list(crashed, p).FD(crashed)$

The name of the process for the failure detector is *FD* as shown in line 3.2.1 with one parameter. We implemented the eventuality with the help of a process, called *CrashedProc*. *CrashedProc* is a simple process (not defined here but given in appendices 1 and 2) where a participant can send a message to the failure detector to add its ID to the list of crashed failures. It notices the process crashing and then continuously pings the failure detector until the ID of the crashed failure is added in the list of suspects. Once the list with respect to a particular process is updated then afterwards the failure detector permanently declares this process as suspected but the time between the crash and the permanent suspicion is not fixed. *FD* has two non-deterministic choices; updating a list of crashed processes and replying the query of a process, which are shown in lines 3.2.1 and 3.2.1, respectively. So eventually each crashed process becomes part of the list called *crashed*, hence we can say that the given failure detector satisfies the property of strong completeness.

3.2.2 The process for Phase 1:

We define this process with the help of following six parameters:

- *myId*:$\mathbb{N}$: The ID-number of the process.
- *round*:$\mathbb{N}$: Every process executes $n-1$ asynchronous rounds and this parameter denotes the current round number. In every round, each process p waits for the message of each correct process q, if q is not suspected.
- $List(\mathbb{N})$: The list that contains the proposed values of all non-suspected processes.
- $\Delta : List(\mathbb{N})$: The list to exchange the proposed values, as discussed in Section 3.2.
- $msg : List(List(\mathbb{N}))$: A two-dimensional list to store the messages of every process in each round.
- $msg_sent : \mathbb{B}$: In every round a process sends its message and then waits without sending the next message. This parameter is used to keep this sequence.

In the following definition we assume the existence of a process *Correct* that remains operational and never gets suspected where $Correct \in \pi$.

1: $Phase1(myId, round : \mathbb{N}, V, \Delta : List(\mathbb{N}),$
$msgs : List(List(\mathbb{N})), msg_sent : \mathbb{B}) =$
2: $(myId \not\approx Correct) \rightarrow$
$crashed(myId) \cdot CrashedProc(myId, false, false, false, false)$
3: $+$
4: $(round \leq N-1) \rightarrow ((\neg msg_sent) \rightarrow send2all(round, \Delta, myId) \cdot$
$Phase1(myId, round, V, \Delta, msgs, true)$

5:
6: $\overset{\diamond}{\sum_{lst:List(\mathbb{N})}} queryFD(lst, myId) \cdot$
$WaitandReceive(myId, round, V, \Delta, msgs, minus(\pi, lst))$
7: $)\diamond$
8: $Phase2(myId, V, [], false);$

9: $WaitandReceive(myId, round : \mathbb{N}, V, \Delta : List(\mathbb{N}),$
$msgs : List(List(\mathbb{N})), from : List(\mathbb{N})) =$
10: $(\#from > 0) \rightarrow ($
11: $\sum_{p:\pi}(p \in from) \rightarrow \sum_{\Delta_q:List(\mathbb{N})} receive(round, \Delta_q, p, myId) \cdot$
12: $(suspected(myId, p, false) \cdot WaitandReceive(myId, round, V,$
$[\perp, \perp, \perp], updateMsgs(p, msgs, \Delta_q), minus(from, [p]))$
13: $+$
14: $(p \not\approx Correct) \rightarrow suspected(myId, p, true) \cdot$
$WaitandReceive(myId, round, V, [\perp, \perp, \perp], msgs, minus(from, [p]))$
15: $)$
16: $+$
17: $rcv_stopWaiting(p) \cdot WaitandReceive(myId, round, V,$
$[\perp, \perp, \perp], msgs, minus(from, [p]))$
18: $)$
19: $\diamond$
20: $Phase1(myId, round + 1, update_V(V, msgs),$
$updateDelta(V, update_V(V, msgs), [\perp, \perp, \perp]), msgs, false);$

The above definition shows that a process in Phase 1, can crash or can send a message to others as shown in lines 3.2.2 and 3.2.2, respectively. *WaitandReceive* is another process, defined in line 3.2.2, used to wait until a process receives all current round message from non-suspected processes. While waiting if it learns from the failure detector that some correct process q has crashed and $q \in D_p$, it stops waiting for the respective message as

shown in line 3.2.2. The process *WaitandReceive* has the same parameters like the process *Phase1*, except a list called *from*. Initially, this list is equal to the non-suspected processes, i.e., $\pi - suspects$ and upon receiving a message from an arbitrary process, say p, it is updated as $from := from - [p]$. It is clear from the informal specifications of Algorithm 1, that a process p is interested to get the list of suspects and to know whether some process q belongs to D_p or not whenever p receives a message from q. So a process in Phase 1 always has two non-deterministic choices (suspect or unsuspect) for a process that is sending messages. If the last argument in an action *suspected* (given in lines 3.2.2 and 3.2.2) is *true* then the sender of the message is suspected, so its sent message is discarded. Whereas the value *false* in the same action points to non-suspicion and thus the list Δ_q is added to $msgs$ using a function, called *updateMsgs*. The condition given in line 3.2.2 takes into account a correct process that is never suspected. The empty list (called *from*) in line 3.2.2 shows that there is no process to wait for, so every process moves to Phase 1.

3.2.3 The process for Phase 2

The process in Phase 2 uses three parameters of Phase 1 (myId, round and V) and a list, called *lastmsgs* to store the lists of other processes.

1: $Phase2(myId : \mathbb{N}, V : List(\mathbb{N}), lastmsgs : List(List(\mathbb{N})),$
$V_sent : \mathbb{B}) =$
2: $(myId \not\approx Correct) \rightarrow send_crashed(myId) \cdot CrashedProc(myId)$
3: $+$
4: $(\neg V_sent) \rightarrow send2all(0, V, myId) \cdot Phase2(myId, V, lastmsgs, true)$
5: $\diamond$
6: $\sum_{lst:List(\mathbb{N})} queryFD(lst, myId) \cdot$
$WaitandReceive2\ (myId, V, lastmsgs, minus(\pi, lst))$

7: $WaitandReceive2(myId : \mathbb{N}, V : List(\mathbb{N}), lastmsgs : List(List(\mathbb{N})),$
$from : List(\mathbb{N})) =$
8: $(\#from > 0) \rightarrow \sum_{q:\mathbb{N}} \sum_{V_q:List(\mathbb{N})} receive(V_q, q, myId) \cdot$
9: $WaitandReceive2(myId, V, updateMsgs(q, lastmsgs, V_q),$
$minus(from, [q]))$
10: $\diamond$
11: $Phase3(myId, updateLastmsgs(lastmsgs, V));$

In this phase, a process has a choice to crash if it is not the correct process (as it has a possibility of erroneous suspicion by the failure detector). The second choice, shown in line 3.2.3, is to first send the list of values and

then receive from all non-suspected correct processes. Line 3.2.3 shows that process queries the failure detector before waiting and then waits by initiating a process called *WaitandReceive2* defined in line 3.2.3. Every participant in this process receives the list of proposed values from other processes and then moves to Phase 3 after making its list similar to others.

3.2.4 The process for Phase 3:

The process for Phase 3 is very simple. Each participant decides the first non-$\perp$ value from its list of available proposed values. The process for *Phase3* takes two parameters, the process ID and the list of values which has been already updated in Phase 2. The definition of this process is:

1: $Phase3(myId : \mathbb{N}, V : List(\mathbb{N})) = decide(myId, findDecided(V))$

The above specification shows that each process in Phase 3, decides a value (non-$\perp$) from the proposed values and then stops.

3.3 Consensus with strong completeness and eventual weak accuracy

The specification settings for this protocol use the functions discussed in Section 3.2. In this protocol different message types are sent and received in different phases. For example, in Phase 1, processes send their estimates, in Phase 3 acknowledgement messages (*ack* or *nack*) are communicated and in Phase 4 either they receive the decided value or start the next round. So we define different channels according to their message types. In this protocol, at a time, only the coordinator is either a source or destination of every message, i.e, other processes send their messages to the coordinator and receive messages from the coordinator only. To realize eventual weak accuracy, we define the following processes with the assumption that $Correct \in \pi$ is one of the correct processes that is never suspected after a certain amount of time.

3.3.1 The process for failure detector

In this protocol the majority of the processes remains correct and we implement this property with the help of a failure detector. It keeps track of the number of crashes (f) and guarantees that $f < \lceil \frac{(n+1)}{2} \rceil$. There are three parameters used in the definition;

- *crashed*:$List(\mathbb{N})$: A list to store the ID-number of the crashed process.
- *totalCrashed*:$\mathbb{N}$: To keep track of the number of crashes.
- *weaklyAccurate*:$\mathbb{B}$: To determine whether the failure detector satisfies weak accuracy or not.

1: $FD(crashed : List(\mathbb{N}), totalCrashed : \mathbb{N}, weaklyAccurate : \mathbb{B}) =$
2: $(totalCrashed \approx 0) \rightarrow \sum_{id:\mathbb{N}} rcv_crashed(id) \cdot$
$FD(crashed, totalCrashed + 1, weaklyAccurate)$
3: $+$
4: $\sum_{id:\mathbb{N}} rcv_addRequest(id) \cdot$
$FD(updateCrashed(crashed, id), totalCrashed, weaklyAccurate)$
5: $+$
6: $(\neg weaklyAccurate) \rightarrow weakAccuracy.FD(crashed, totalCrashed, true)$
7: $+$
8: $(weaklyAccurate) \rightarrow \sum_{round:\mathbb{N}} \sum_{p:\pi} replyQuery(crashed, p, round)$
9:
10: $\overset{\diamond}{\sum_{round:\mathbb{N}}} \sum_{p:\pi} (replyQuery(crashed, p, round)$
11: $+$
12: $replyQuery(Addcrashed([Correct], crashed), p, round)$
$) \cdot FD(crashed, totalCrashed, weaklyAccurate);$

In line 3.3.1, the failure detector determines the number of already crashed processes. If they are less than $\frac{N}{2}$ (i.e., equal to 0, if N=3) and any other process crashes in the meanwhile then the counter for crash failures increases without immediately adding such process to the crashed processes. To meet the property of strong completeness, a crashed process is eventually added to the crashed processes as shown in line 3.3.1. In the same way, the weak accuracy is also eventual, so non-deterministically at some point the failure detector becomes weakly accurate (line 3.3.1), i.e., from on, it will not consider a particular correct process as crash failure (line 3.3.1). Otherwise, due to unreliability of the failure detector, it can send a list of crashed processes containing a correct process as shown in line 3.3.1.

3.3.2 The process for Phase 1

It is assumed that every sent message will be eventually delivered but the protocol specification gives us no information about a message that is sent from a process and the only recipient, i.e., the coordinator crashes before receiving it. Due to the asynchronous behaviour of the distributed system, the delays in channels are unbounded and there is no guarantee that messages will be delivered in the same order in which they are sent. To alleviate this problematic situation, we modeled the process for Phase 1 in a way that every process uses a separate channel for a message in each round. In this way the algorithm demonstrates the asynchronous behaviour. But to reach the terminated state, a process can go through several asynchronous rounds

[1], so we modeled the Phase 1 in a manner that if the algorithm does not terminate in N rounds (N is the number of processes) then the round number is reset to its initial value, shown in line 3.3.2. In every round, there is a new coordinator. So, the recipient varies with respect to round number. We define this process by means of four parameters, $myId, round, estimate$ and ts where ts is the last round number in which a process has updated its estimate (default is 0).

1: $Phase1(myId, round, estimate, ts : \mathbb{N}) =$
2: $(round \leq N) \rightarrow send(1, myId, round, estimate, ts) \cdot Phase2(myId, round, estimate, ts, \pi, 0)$
$\diamond$
$send(1, myId, 0, estimate, ts) \cdot Phase2(myId, 0, estimate, ts, \pi, 0)$
3: $+$
4: $(myId \not\approx Correct) \rightarrow send_crashed(myId) \cdot CrashedProc(myId, round, minus(\pi, [myId]), false)$

3.3.3 The process for Phase 2

Every process initiates this phase from Phase 1 but only the coordinator executes it and the rest of the processes jump to Phase 3. This phase is formally specified as:

1: $Phase2(myId, round, estimate, ts : \mathbb{N}, from : List(\mathbb{N}), i : \mathbb{N}) =$
2: $(myId \not\approx Correct) \rightarrow send_crashed(myId) \cdot$
$Crashed(myId, round, minus(\pi, [myId]), false)$
3: $+$
4: $((round \bmod N) + 1 \approx myId \;\&\&\; \#from > 0) \rightarrow$
5: $((i < (N+1)) \; div \; 2) \rightarrow$
6: $\sum_{q, estimate_q, ts_q : \mathbb{N}} rcvfrom(1, q, round, estimate_q, ts_q, myId) \cdot$
7: $Phase2(myId, round, updateEstimate(estimate, estimate_q, ts, ts_q), isGreater(ts, ts_q), minus(from, [q]), i+1)$
8: $\diamond$
9: $send(2, myId, round, estimate, ts) \cdot Phase3(myId, round, estimate, ts)$
10:)
11: $\diamond$
12: $Phase3(myId, round, estimate, ts);$

Line 3.3.3 shows that a process can crash if it is not a process due to which this protocol satisfies weak accuracy. In line 3.3.3, the coordinator waits for at least $\lceil \frac{(n+1)}{2} \rceil$ processes. If a process q sends its message such that $ts_q > ts_c$, then the coordinator adopts the q's estimate. For this purpose it uses a

specifically defined function, called *updateEstimate*, shown in line 3.3.3. After receiving the messages from the majority, the coordinator broadcasts its estimate and proceeds for Phase 3, as shown in line 3.3.3.

3.3.4 The process for Phase 3

We define the process for Phase 3 as:

1: $Phase3(myId, round, estimate, ts : \mathbb{N}) =$
2: $(myId \not\approx Correct) \rightarrow send_crashed(myId) \cdot$
$Crashed(myId, round, minus(\pi, [myId]))$
3: $+$
4: $rcv_CFailure(myId, round) \cdot Phase1(myId, round + 1, estimate, ts)$
5: $+$
6: $\sum_{est_q, ts_q : \mathbb{N}} rcvfrom(2, (round \bmod N) + 1, round, est_q, ts_q, myId) \cdot$
7: $\sum_{lst : List(\mathbb{N})} rcv_list(lst, myId, round) \cdot$
8: $((round \bmod N) + 1 \in lst) \rightarrow$
$send3(myId, round, nack, (round \bmod N) + 1) \cdot$
$Phase4(myId, round, estimate, ts, 0, \pi)$
9: $\diamond$
10: $send3(myId, round, ack, (round \bmod N) + 1)$
$\cdot Phase4(myId, round, est_q, ts_q, 0, \pi);$

Crashing of any process at this phase is shown in line 2, whereas line 4 shows the crashing of coordinator and if this happens then every process restarts Phase 1 with the next round number. According to round number, the new coordinator is designated and the other processes send their estimates to the current coordinator. If both the process and the coordinator are not crashed then the process receives the estimate of coordinator (line 3.3.4) and quires the failure detector (line 3.3.4) to send either *ack* or *nack*. The message *ack*, if coordinator is not in the list of suspects(line 3.3.4) otherwise the message *nack* is sent as a reply (line 3.3.4).

3.3.5 The process for Phase 4

In this phase either all of the processes including the coordinator agree upon a value or move to the next round. We define the process with two extra parameters from Phase 3; $i : \mathbb{N}$ and $from : List(\mathbb{N})$. The first one is used for counting the received messages and second one (initially π) is used to receive one message from each process.

1: $Phase4(myId, round, estimate, ts, i : \mathbb{N}, from : List(\mathbb{N})) =$
2: $(myId \not\approx Correct) \rightarrow send_crashed(myId)\cdot$
$Crashed(myId, round, minus(\pi, [myId]), false)$
3: $+$
4: $((round \bmod N) + 1 \approx myId) \rightarrow$
5: $($ $(i < (N+1)\ div\ 2) \rightarrow$
6: $(\sum_{q:\mathbb{N}} \sum_{msg_type:Ack_Type} rcvAckNack(q, round, msg_type, myId)\cdot$
7: $(msg_type \approx ack) \rightarrow$
8: $Phase4(myId, round, estimate, ts,$
$i+1, minus(from, [q]))$
9: $\diamond$
10: $StartNextRound(myId, round, estimate,$
$ts, minus(from, [q]))$
11: $)$ $\diamond$
12: $sendDecision(myId, estimate, true)\cdot$
$decide(myId, estimate)\cdot\delta$ {δ denotes the deadlock}
13: $)$
$\diamond$
14: $Wait4decision(myId, round, estimate, ts, false, false)$;
15:
16: $Wait4decision(myId, round, estimate, ts : \mathbb{N}, decided, finish : \mathbb{B}) =$
17: $waiting4decision(myId)\cdot$
18: $(rcv_CFailure(myId, round) \cdot Phase1(myId, round+1, estimate, ts)$
19: $+$
20: $\sum_{v:\mathbb{N}} \sum_{done:\mathbb{B}} rcvDecisioFrom(v, done, myId) \cdot (done) \rightarrow decide(myId, v).\delta$
21: $\diamond$
22: $Phase1(myId, round+1, estimate, ts)$
$)$;

The option for a process to crash is shown in line 3.3.5 and line 4 shows that it waits for $\lceil \frac{(n+1)}{2} \rceil$ messages if it is a coordinator. If a majority send *ack* messages, t'he coordinator decides and sends the decided value to all processes as shown in line 3.3.5 and respective channel ensures that this decided value is delivered.

4 General Requirements

The general requirements of a consensus problem given in [1] are:

R1. *Uniform Agreement:* "No two processes decide differently".

R2. *Uniform Integrity:* "Each process decides at most once".

R3. *Termination* "All correct processes eventually decide on some value".

R4. *Uniform Validity* "If a process decides on value v, then v has been proposed by some process".

4.1 Requirement specification in the μ-calculus

In order to verify the requirements with respect to the formalization, they are specified in the modal μ-calculus ([7], extended with data-dependent processes and regular formulae).

R1. According to "uniform agreement" in [9] any two processes always decide the same value, i.e., the decision of all processes is unanimous [1, 8]. We devise the following formula for any two processes $p, p' \in \pi$, to ensure that their decided values cannot be different. Assume that V is the set of all values.

$$\forall_{v,v' \in V} \forall_{p,p' \in \pi} [true^* \cdot decide(p,v) \cdot true^* \cdot decide(p',v')](v = v')$$

R2. The following formula specifies for each process p, the action $decide(p,v)$, for any arbitrary value v appears at most once in each trace. This in turn guarantees uniform integrity.

$$\forall_{p \in \pi}, \forall_{v,v' \in V} [true^* \cdot decide(p,v).true^* \cdot decide(p,v')]false$$

R3. Termination of a process can be viewed in two different scenarios; crashed and correct. If a process is crashed before reaching the last phase, according to both Algorithms 1 and 2, it cannot decide a value. On the other hand, if it remains correct throughout the execution, it eventually decides a value provided that the respective failure detector satisfies certain properties regarding accuracy and completeness. This requirement for Algorithm 1 is expressed in the μ-calculus as follows:

$$\forall_{p \in \pi}\ \mu X \cdot ([\overline{crashed(p)} \wedge (\forall_{v \in V} \overline{decide(p,v)}\ )]X \wedge < true > true)$$

Where $p \in \pi$ and V is the set of proposed values. This formula states that either the action *crash* or *decide* must unavoidably be taken. The formula does not speak about strong completeness because according to LEMMA 5 in [1] Algorithm 1 is blocked forever if a process p is waiting for a message from a crashed process q and $q \notin D_p$, i.e., no strong completeness. According to the specification in [1], there is a time after which D_p satisfies strong completeness, i.e., $q \in D_p$, hence waiting forever is ultimately avoided. The same holds for Algorithm 2

where the property of eventual weak accuracy is also mandatory but the time required for its adoption by the failure detector is not fixed. To handle this eventuality, we introduce an action for the failure detector, called *weakAccuracy* (discussed in Section 3.3.1) to determine whether the failure detector is weakly accurate or not. As soon as it satisfies this property, every non-crashed process is supposed to either reach to a decision or crash. So, for Algorithm 2, we express this requirement in μ-calculus as:

$$\forall_{p\in\pi}[(\overline{crashed(p)} \wedge (\forall_{v\in V}\overline{decide(p,v)}\ ))^*.weakAccuracy]$$

$$\mu X \cdot ([\overline{crashed(p)} \wedge (\forall_{v\in V}\overline{decide(p,v)}\ )]X \wedge < true > true)$$

R4. In Phase 1 of both Algorithms 1 and 2, every correct process proposes a value and in the last phase, it decides a value. According to this requirement, the decided value can only be a proposed value by some participant. The formalization of this requirement in the μ-calculus is:

$$\forall_{p\in\pi}, \forall_{v\in V}[(\forall_{p'\in\pi}\overline{send(p',v)})^* \cdot decide(p,v)]false$$

4.2 Verification results

To verify whether the above-mentioned requirements are satisfied or violated, we use the Evaluator model checker (version 1.5) of the CADP toolset [2, 5] and found that both protocols meet all of these requirements. Model checking was done for three number of processes and we use Pentium Dual Core (1.8 GHz) machine with 2 GB of RAM. The amount of time spent on the verification of each property is reported in Table 1. We use strong bisimulation reduction technique to reduce the size of the state space, hence the time mentioned in Table 1 also includes this reduction time. The following commands in given sequence make the results available where the INFILE contains formal specification and the FORMULA file contains a μ-calculus formula.

1. mcrl22lps -v -D INFILE.mcrl2 OUTFILE.lps
   To translate an mCRL2 process specification from INFILE.mcrl2 to a linear process specification (LPS), to be stored in the file named, OUTFILE.lps. The option *v* (verbose) displays the short intermediate messages while the option *D* (delta) is necessary to enforce the untimed semantics of mCRL2 (i.e., to allow for arbitrary time steps in all reachable states).

2. lpsconstelm -v OUTFILE.lps temp.lps
   To reduce the linear process specification by removing spurious constant process parameters from the OUTFILE.lps and write the result to temp.lps.

3. lpssumelm -v temp.lps OUTFILE.lps
   To remove superfluous summations from the temp.lps and write the result to OUTFILE.lps.

4. lpsparelm -v OUTFILE.lps temp.lps
   To remove unused parameters from the OUTFILE.lps and write the result to temp.lps.

5. lps2lts -v -ftree temp.lps OUTFILE.svc
   To generate a labelled transition system (LTS) from the temp.lps and write the result to OUTFILE.svc. The option *ftree* is used to store state internally in tree format for efficient usage of memory.

6. ltsconvert -ebisim -v OUTFILE.svc OUTFILE.aut
   To convert the labelled transition system (LTS) in OUTFILE.svc to OUTFILE.aut after applying the modulo strong bisimilarity as minimisation method.

7. bcg_io OUTFILE.aut OUTFILE.bcg
   To convert graphs from OUTFILE.aut into the Binary Coded Graphs (BCG) format, which is the input format for CADP toolset.

8. bcg_open OUTFILE.bcg evaluator -verbose -bfs -diag FORMULA.mcl
   To diagnose that whether the formula given in FORMULA.mcl satisfied or not. In case it is refuted then a trace showing the counter example is displayed due to the option *diag* where the option *bfs* is used for breadth first search.

	Algorithm 1	Algorithm 2
Time to generate state space	9h54m0s	1h37m0s
Number of states	1507990	45329
R1	12m13.470s	0m22.013s
R2	12m4.160s	0m22.135s
R3	7m17.847s	0m9.490s
R4	0m5.573s	0m0.315s

Table 1: Time required for the verification using the CADP toolset

We also apply another tool for model-checking, called PBES2Bool (version June 2009), which is part of the mCRL2 toolset and give the required amount of time for the verification in Table 2. The advantage of this tool, compared to the Evaluator tool, is that it does not require generation of state space and the time required for the verification of each individual requirement is less than the time needed to both generate the state space and verify the same requirement in CADP, shown in Table 2. However, the total

time for the verification of all the requirements is little bit longer: namely $1h38m24.304s$ for PBES2Bool vs $1h37m53.953s$ for generating state-space plus modelchecking in CADP. We could verify the requirements only for Algorithm 2 with $n = 3$ because of its smaller number of transitions. To get the results we use the following commands in the given order after generating linear process specification in temp.lps file (after step 4 given above) and specify μ-calculus formulae in FORMULA.mcf file.

1. lps2pbes -f FORMULA.mcf temp.lps OUTFILE.pbes
   To convert the state formula in FORMULA.mcf and the LPS in temp.lps to a parameterized boolean equation system (PBES) and save it to OUTFILE.pbes.

2. pbesparelm -v temp.pbes OUTFILE.pbes
   To apply parameter elimination on temp.pbes and write it to OUTFILE.pbes.

3. pbes2bool -vprjittyc OUTFILE.pbes -s1
   To solve the parameterized boolean equation system (PBES) in OUTFILE.pbes. The option *vprjittyc* is combination of multiple abbreviations; v to display short intermediate messages, p to precompile the pbes for faster rewriting and r to use the rewrite strategy, called jittyc [10].

	Algorithm 2
R1	40m58.730s
R2	42m15.587s
R3	14m59.494s
R4	0m10.493s

Table 2: Time required for verification using the mCRL2 toolset

5 Conclusions

In fault-tolerant distributed systems, the consensus problem plays a fundamental role [9]. In the consensus problem, every process proposes a value and if it remains non-crashed during execution then it eventually decides a value with the property that the decision is irrevocable and unanimous [8]. Consensus cannot be solved in asynchronous distributed systems with crash failures [4]. Hence to implement consensus, participating processes rely on a notion of the failure detector. A failure detector is called *perfect*, if it never suspects a correct process but eventually suspects every crashed process. In asynchronous systems, it is impossible to devise a perfect failure detector because it cannot differentiate between a crashed failure and a slow process. In

[1], unreliable failure detector are introduces to solve the consensus problem in an asynchronous system with crash failures provided that they satisfy the properties of completeness and accuracy.

In this paper, we formalized two distributed algorithms for the consensus problem with their requirements. Our verification shows that all of the requirements are satisfied by both algorithms. We presented our approach for specification of the protocols in the mCRL2 syntax and the requirements in the modal μ-calculus. We devised a common failure detector that satisfies weak accuracy and strong completeness (or eventual strong completeness). We model-checked the behaviour of the protocols with three participating process.

Acknowledgements

The author would like to thank MohammadReza Mousavi, Jan Friso Groote and Muhammad Rizwan Asghar for reviews and valuable comments.

References

[1] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[2] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp - a protocol validation and verification toolbox. In *CAV*, pages 437–440, 1996.

[3] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In Marek Karpinski, editor, *FCT*, volume 158 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 1983.

[4] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[5] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification (CAV'2007) Lecture Notes in Computer Science*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163, Berlin Germany, 2007.

[6] Jan Friso Groote, Aad Mathijssen, Muck van Weerdenburg, and Yaroslav S. Usenko. From μCRL to mCRL2: motivation and outline. *Electr. Notes Theor. Comput. Sci.*, 162:191–196, 2006.

[7] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[8] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing.* Cambridge University Press, The Edinburgh Building, Cambridge CB2 8RU, UK, 2008.

[9] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *J. Algorithms*, 11(3):374–419, 1990.

[10] Muck van Weerdenburg. An account of implementing applicative term rewriting. *Electron. Notes Theor. Comput. Sci.*, 174(10):139–155, 2007.

A mCRL2 specification for consensus problem with strong completeness and weak accuracy

This is the mCRL2 specifications of the consensus problem discussed in Section 2.2.

```
map

  N : ℕ;
  minus : List(ℕ) × List(ℕ) → List(ℕ);
  eliminate : List(ℕ) × ℕ → List(ℕ);
  update_V : List(ℕ) × List(List(ℕ)) → List(ℕ);
  removeBottom : List(ℕ) × List(ℕ) → List(ℕ);
  update_V2phase : List(List(ℕ)) × List(ℕ) × ℕ → List(ℕ);
  updateDelta : List(ℕ) × List(ℕ) × List(ℕ) → List(ℕ);
  findDecided : List(ℕ) → ℕ;
  π : List(ℕ);
  updateMsgs : ℕ × List(List(ℕ)) × List(ℕ) → List(List(ℕ));
  updateCrashed : List(ℕ) × ℕ → List(ℕ);
  addcrashed : List(ℕ) × List(ℕ) → List(ℕ);
  makeIdentical : List(ℕ) × List(ℕ) → List(ℕ);
  updateLastmsgs : List(List(ℕ)) × List(ℕ) → List(ℕ);
  Correct : ℕ;

var

  ln, lg, ld : List(ℕ);
  msgs : List(List(ℕ));
  lb : List(𝔹);
  x, m, n, k : ℕ;
  s, b, p : 𝔹;

eqn

  updateLastmsgs(lg ▷ msgs, ln) =
     if(#msgs > 0, updateLastmsgs(msgs, makeIdentical(lg, ln)), makeIdentical(lg, ln));
  updateLastmsgs([], ln) = ln;
  makeIdentical(ln, []) = [];
  makeIdentical(x ▷ lg, n ▷ ln) = %0 is used for ⊥
     if(x ≈ 0, 0 ▷ makeIdentical(lg, ln), n ▷ makeIdentical(lg, ln));
  N = 3; % Total Number of processes
  π = [0, 1, 2]; %IDs of the processes
  Correct = 2; %ID of the correct process
  minus([], lg) = [];
  minus(ln, []) = ln;
  minus(n ▷ ln, m ▷ lg) = if(m ∈ n ▷ ln, minus(eliminate(n ▷ ln, m), lg), minus(n ▷ ln, lg));
  eliminate(n ▷ ln, m) = if(n ≈ m, ln, n ▷ eliminate(ln, m));
  updateDelta([], lg, ln) = [];
  updateDelta(n ▷ lg, m ▷ ln, x ▷ ld) =
       if(m ≉ n, m ▷ updateDelta(lg, ln, ld), x ▷ updateDelta(lg, ln, ld));
  update_V(ln, lg ▷ msgs) =
  if(#msgs > 0, update_V(removeBottom(ln, lg), msgs), removeBottom(ln, lg));
  removeBottom(n ▷ ln, k ▷ lg) =
  if(n ≈ 0 ∧ k ≉ 0, k ▷ removeBottom(ln, lg), n ▷ removeBottom(ln, lg));
  removeBottom([], []) = [];
  removeBottom([], lg) = [];
  removeBottom(ln, []) = [];
  update_V2phase(msgs, [], k) = [];
  update_V2phase(ln ▷ msgs, n ▷ lg, k) =
     if(ln.k ≈ 0, 0 ▷ update_V2phase(msgs, lg, k + 1),
                  n ▷ update_V2phase(msgs, lg, k + 1));
```

```
56  findDecided([]) = 0;
57  findDecided(n ▷ ln) = if(n ≉ 0, n, findDecided(ln));
58  updateMsgs(0, lg ▷ msgs, ln) = ln ▷ msgs;
59  updateMsgs(0, [], ln) = [ln];
60  (n > 0) → updateMsgs(n, lg ▷ msgs, ln) = lg ▷ updateMsgs(Int2Nat(n − 1), msgs, ln);
61  updateCrashed([], n) = [];
62  updateCrashed(ln, n) = if(n ∈ ln, ln, n ▷ ln);
63  addcrashed([], []) = [];
64  addcrashed(ln, []) = ln;
65  addcrashed(ln, n ▷ lg) = if(n ∈ ln, addcrashed(ln, lg), n ▷ addcrashed(ln, lg));
66
67  act
68
69  send2all, rcv, broadcast : ℕ × List(ℕ) × ℕ;
70  sendTo, receive, received : ℕ × List(ℕ) × ℕ × ℕ;
71  decide : ℕ × ℕ;
72  rcv_crashing, rcv_query : ℕ;
73  send_list, queryFD, getCrashedList : List(ℕ) × ℕ;
74  suspected : ℕ × ℕ × 𝔹;
75  crashed,
76  send_stopWaiting, rcv_stopWaiting, stopWaiting, strongComplete : ℕ;
77  suspect : ℕ × ℕ;
78
79  proc
80
81      %%%%
```

```
118  +
119  (round ≤ N − 1) → ((¬msg_sent) → send2all(round, Δ, myId).
120                                                  Phase1(myId, round, V, Δ, msgs, true)
121                                          ◇
122                                          ∑_{lst:List(ℕ)} .queryFD(lst, myId).
123                                  WaitandReceive(myId, round, V, Δ, msgs, minus(π, lst))
124                          )
125                  ◇
126   Phase2(myId, V, [minus([0], [0]), minus([0], [0]), minus([0], [0])], false);
127
128     %%%%
```

```
(¬mt2 ∧ stronglyComplete) → send_stopWaiting(myId).
                        CrashedProc(myId, true, mt3, stronglyComplete);
+
(¬mt3 ∧ stronglyComplete) → send_stopWaiting(myId).
                        CrashedProc(myId, mt2, true, stronglyComplete)

    %%%%
```

B mCRL2 specification for consensus problem with strong completeness and eventual weak accuracy

This is the mCRL2 specifications of the consensus problem discussed in Section 2.3.

```
sort

  Ack_Type = struct ack | nack;

map

  N : Pos;
  Correct : ℕ;
  π : List(ℕ);
  minus : List(ℕ) × List(ℕ) → List(ℕ);
  eliminate : List(ℕ) × ℕ → List(ℕ);
  isGreater : ℕ × ℕ → ℕ;
  updateEstimate : ℕ × ℕ × ℕ × ℕ → ℕ;
  addcrashed : List(ℕ) × List(ℕ) → List(ℕ);
  Addcrashed : List(ℕ) × List(ℕ) → List(ℕ);
  updateCrashed : List(ℕ) × ℕ → List(ℕ);

var

  ln, lg, ld : List(ℕ);
  msgs : List(List(ℕ));
  lb : List(𝔹);
  x, m, n, k : ℕ;
  s, b : 𝔹;

eqn

  N = 3;
  Correct = 2;
  π = [0, 1, 2];
  minus([], lg) = [];
  minus(ln, []) = ln;
  minus(n ▷ ln, m ▷ lg) = if(m ∈ n ▷ ln, minus(eliminate(n ▷ ln, m), lg), minus(n ▷ ln, lg));
  eliminate(n ▷ ln, m) = if(n ≈ m, ln, n ▷ eliminate(ln, m));
  isGreater(n, m) = if(m > n, m, n);
  updateEstimate(x, k, n, m) = if(m > n, k, x);
  Addcrashed(n ▷ ln, lg) =
      if(n ≈ Correct, addcrashed(ln, lg), addcrashed(n ▷ ln, lg));
  Addcrashed([], lg) = lg;
  addcrashed([], []) = [];
  addcrashed(ln, []) = ln;
  addcrashed(ln, n ▷ lg) = if(n ∈ ln, addcrashed(ln, lg), n ▷ addcrashed(ln, lg));
  updateCrashed([], n) = [];
  updateCrashed(ln, n) = if(n ∈ ln, ln, n ▷ ln);

act

  send, rcv, broadcast : ℕ × ℕ × ℕ × ℕ × ℕ;
  sendTo, rcvfrom, received : ℕ × ℕ × ℕ × ℕ × ℕ × ℕ;
  weakAccuracy, replyQuery, rcv_list, queryFD : List(ℕ) × ℕ × ℕ;
  sendDecision, rcvDecision, DecisionBC : ℕ × ℕ × 𝔹 × List(ℕ);
  rcvDecisioFrom, sendDecisionTo, DecisionRcvd : ℕ × 𝔹 × ℕ;
  decide : ℕ × ℕ;
  send3, rcv3, SendAckNack : ℕ × ℕ × Ack_Type × ℕ;
  sendAckNack, rcvAckNack, AckNack_rcvd : ℕ × ℕ × Ack_Type × ℕ;
```

```
56  rcv_crashed, send_crashed, crashed, waiting4decision : ℕ;
57  send_CFailure, rcv_CFailure, CFailure : ℕ × ℕ;
58  send_addRequest, rcv_addRequest, strongComplte : ℕ;
59
60 proc
61
62  FD(crashed : List(ℕ), totalCrashed : ℕ, weaklyAccurate : 𝔹) =
63  % only one process out of three is allowed to crash
64  (totalCrashed ≈ 0) → ∑_{id:ℕ} .rcv_crashed(id).
65                                  FD(crashed, totalCrashed + 1, weaklyAccurate)
66  +
67  ∑_{id:𝔹} .rcv_addRequest(id).FD(updateCrashed(crashed, id), totalCrashed, weaklyAccurate)
68  +
69  (¬weaklyAccurate) → weakAccuracy.FD(crashed, totalCrashed, true)
70  +
71  ((weaklyAccurate) → (∑_{round:ℕ} .replyQuery(Addcrashed(((round mod N) + 1) ▷ [], crashed), 0, round)
72     +
73     ∑_{round:ℕ} .replyQuery(crashed, 0, round)
74     +
75     ∑_{round:ℕ} .replyQuery(Addcrashed(((round mod N) + 1) ▷ [], crashed), 1, round)
76     +
77     ∑_{round:ℕ} .replyQuery(crashed, 1, round)
78     +
79     ∑_{round:ℕ} .replyQuery(Addcrashed(((round mod N) + 1) ▷ [], crashed), 2, round)
80     +
81     ∑_{round:ℕ} .replyQuery(crashed, 2, round)
82  )
83  ◇
84  (
85   ∑_{round:ℕ} .replyQuery(Addcrashed([Correct], crashed), 0, round)
86     +
87     ∑_{round:ℕ} .replyQuery(crashed, 0, round)
88     +
89     ∑_{round:ℕ} .replyQuery(Addcrashed([Correct], crashed), 1, round)
90     +
91     ∑_{round:ℕ} .replyQuery(crashed, 1, round)
92     +
93     ∑_{round:ℕ} .replyQuery(Addcrashed([Correct], crashed), 2, round)
94     +
95     ∑_{round:ℕ} .replyQuery(crashed, 2, round)
96  )
97  ).FD(crashed, totalCrashed, weaklyAccurate);
98
99      %%%%
```

```
118 )◇
119           ( sendTo(phase, myId, round, estimate, ts, (round mod N) + 1)
120           ).Channel(myId, round);
121
122 Channel4AckNack(myId, round : ℕ) =
123   ∑_{to:ℕ, msg_type:Ack_Type} .rcv3(myId, round, msg_type, to).
124   (sendAckNack(myId, round, msg_type, to).Channel4AckNack(myId, round);
125
126 Channel4Decision(myId : ℕ) =
127 ∑_{estimate:ℕ} · ∑_{flag:𝔹} · ∑_{To:List(ℕ)} .rcvDecision(myId, estimate, flag, To).
128                                   randomBroadcastDecision(myId, estimate, flag, To);
129
130 randomBroadcastDecision(myId, estimate : ℕ, flag : 𝔹, To : List(ℕ)) =
131   (#To > 0) →
132     ((0 ∈ To) → sendDecisionTo(estimate, flag, 0).
133                   randomBroadcastDecision(myId, estimate, flag, minus(To, [0]))
134     +
135     (1 ∈ To) → sendDecisionTo(estimate, flag, 1).
136                   randomBroadcastDecision(myId, estimate, flag, minus(To, [1]))
137     +
138     (2 ∈ To) → sendDecisionTo(estimate, flag, 2).
139                   randomBroadcastDecision(myId, estimate, flag, minus(To, [2]))
140     )
141     ◇
142       Channel4Decision(myId);
143
144       %%%%
```

```
179     +
180     ∑_{est_q,ts_q:ℕ} .rcvfrom(2, (round mod N) + 1, round, est_q, ts_q, myId).
181                                 ∑_{lst:List(ℕ)} .rcv_list(lst, myId, round).
182     ((round mod N) + 1 ∈ lst) → send3(myId, round, nack, (round mod N) + 1).
183                                     Phase4(myId, round, estimate, ts, 0, π)
184                                 ◇
185                                     send3(myId, round, ack, (round mod N) + 1).
186                                     Phase4(myId, round, est_q, ts_q, 0, π);
187
188 Phase4(myId, round, estimate, ts, i : ℕ, from : List(ℕ)) =
189     (myId ≉ Correct) →
190                 send_crashed(myId).Crashed(myId, round, minus(π, [myId]), false)
191                 +
192                 ((round mod N) + 1 ≈ myId) →
193                     ((i < (N + 1) div 2) →
194                         (∑_{q:ℕ} · ∑_{msg_type:Ack_Type} .rcvAckNack(q, round, msg_type, myId).
195                             (msg_type ≈ ack) →
196                                                 Phase4(myId, round, estimate,
197                                                         ts, i + 1, minus(from, [q]))
198                                             ◇
199                                               StartNextRound(myId, round, estimate,
200                                               ts, minus(from, [q]))
201                             )
202                                         ◇
203                         sendDecision(myId, estimate, true, minus(π, [myId])).
204                         decide(myId, estimate).δ
205                 )
206                     ◇
207                   Wait4decision(myId, round, estimate, ts, false, false);
208
209 StartNextRound(myId, round, estimate, ts : ℕ, from : List(ℕ)) =
210   (#from > 0) → ∑_{msg_type:Ack_Type} ·
211         ((0 ∈ from) → (rcvAckNack(0, round, msg_type, myId)
212                         +
213                         rcv_discardWaiting(0, myId)
214                       ).
215                   StartNextRound(myId, round, estimate, ts, minus(from, [0]))
216                   +
217                   (1 ∈ from) → (rcvAckNack(1, round, msg_type, myId)
218                                   +rcv_discardWaiting(1, myId)
219                                 ).StartNextRound(myId, round, estimate
220                                                   , ts, minus(from, [1]))
221                   +(2 ∈ from) → (rcvAckNack(2, round, msg_type, myId)
222                                   +rcv_discardWaiting(2, myId)
223                                 ).StartNextRound(myId, round, estimate
224                                                   , ts, minus(from, [2]))
225         )
226   ◇
227 sendDecision(myId, estimate, false, minus(π, [myId])).
228     Phase1(myId, round + 1, estimate, ts);
229
230 Wait4decision(myId, round, estimate, ts : ℕ, decided, finish : 𝔹) =
231 waiting4decision(myId).(
232     rcv_CFailure(myId, round).Phase1(myId, round + 1, estimate, ts)
233     +
234     ∑_{v:ℕ} · ∑_{done:𝔹} .rcvDecisionFrom(v, done, myId)
235     .(done) → decide(myId, v).δ
236                 ◇
237                   Phase1(myId, round + 1, estimate, ts));
238
239
```

```
%%%
```