Developing a Consensus Algorithm using Stepwise Refinement

Jeremy Bryans

School of Computing Science, Newcastle University, United Kingdom
Jeremy.Bryans@ncl.ac.uk

Abstract. Consensus problems arise in any area of computing where distributed processes must come to a joint decision. Although solutions to consensus problems have similar aims, they vary according to the processor faults and network properties that must be taken into account, and modifying these assumptions will lead to different algorithms. Reasoning about consensus protocols is subtle, and correctness proofs are often informal. This paper gives a fully formal development and proof of a known consensus algorithm using the stepwise refinement method Event-B. This allows us to manage the complexity of the proof process by factoring the proof of correctness into a number of refinement steps, and to carry out the proof task concurrently with the development. During the development the processor faults and network properties on which the development steps rely are identified. The research outlined here is motivated by the observation that making different choices at these points may lead to alternative algorithms and proofs, leading to a refinement tree of algorithms with partially shared proofs.

Keywords: Consensus Algorithms, Stepwise Refinement, Verification, Event-B

1 Introduction

A consensus problem is one in which a number of distributed processes must come to a common decision despite different initial proposals from the processors. They arise in many areas of computing, such as the decision to commit to a transaction on a distributed database or agreeing a common value from a number of independent sensors. A consensus algorithm is an algorithm which solves the consensus problem for particular processor and network fault assumptions, timing models and reliability/performance trade-offs. The wide variety of these assumptions has led to the design of a wide variety of bespoke consensus algorithms.

Developing consensus algorithms and proving them to be correct is a challenging task and in many cases informal proofs of correctness are provided. The research in this paper is motivated by the eventual goal of defining a taxonomy of consensus algorithms, in which algorithms are more or less closely related according to the similarity or disparity of their underlying assumptions. Such a taxonomy could then form a basis for a set of stepwise-refined formal developments of consensus algorithms which would share more or less steps according to the similarity of their fault assumptions.

The purpose of this work is to give a refinement-based approach to the formal development and proof of a well-known consensus algorithm as a means of evaluating the

plausibility of a formal taxonomy of consensus protocols. During the development the processor faults and network properties on which the development steps rely are identified. Development using stepwise refinement has a number of benefits. Proof complexity is managed by splitting the proof over a number of refinement steps, so proof invariants may be given and proved at the earliest possible stage, before the introduction of distracting detail. A stepwise development naturally postpones some decisions (such as particular fault and network models) and related algorithms may therefore be developed by making different choices at these points, therefore reusing early parts of a development.

In this paper a formal development of the Floodset consensus algorithm [13] is given, using the modelling language Event-B [1]. Floodset is chosen because it is a relatively straightforward consensus algorithm with strong fault assumptions, and the Event-B formalism is chosen because it supports stepwise refinement by structuring developments into a chain of machines linked by refinement relations, thereby managing the complexity of proof. It is also well supported by proof tools.

Sect. 2 gives the consensus correctness criteria, as well as a description of the Floodset algorithm and assumptions. The modelling technology used is outlined in Sect. 3. The body of the work is in Sect. 4, which describes the development of the Floodset algorithm by stepwise refinement. Sect. 5 draws some conclusions and considers the plausibility of this work as a basis for a taxonomy of consensus algorithms.

Related Work Event-B is used in [5] to model the distributed reference counting algorithm, which shares and removes resources in a distributed way while ensuring that shared resources currently being used elsewhere are not removed. The given algorithm does not allow for potential faults. In [3] the authors use Event-B to give a stepwise refinement model of the IEEE 1394 Tree Identify Protocol. This is a specialised consensus problem in which participants must elect a leader. A single abstract event is refined into an existing protocol and message-passing between participants is introduced in a later refinement. Potential faults within the system are not considered. In [8] an algorithm for topology discovery is presented in which individual nodes in a network must remain up-to-date about the changing topology of the network.

In [14] security protocols are developed using Isabelle/HOL. Stepwise refinement is exploited to break the development into logical stages, and to allow the possibility of making different choices at various stages in a development. In [9] Event-B is used to model a consensus protocol under similar assumptions to those made here – messages may be dropped but not forged. The initial machine is roughly equivalent to our machine $X4$. The authors do not address the algorithmic description of protocols. Event-B is used to consider consensus analysis in [15]. The focus there is on multi-agent systems and the specification of separate machines which are later composed.

The Heard-Of model [7] is a common representation of a number of standard systems and failure assumptions, and has been used to verify complex protocols [6].

2 The Floodset Algorithm

A consensus algorithm is one which meets a number of correctness properties and there are a number of ways in the literature of formulating these. In this work, the following definitions, taken from [13], are chosen.

Agreement: No two correct processes decide on different values,
Validity: Any decision value for a process is an initial value for some process, and
Termination: All correct processes eventually reach a decision.

The *Floodset* algorithm [13] is a solution to the consensus problem. It assumes a synchronous network model (processor computation takes place in synchronous rounds) and *failstop* processors (processors may only fail by stopping, and once stopped cannot restart during that execution of the algorithm.) Processors may not behave maliciously. Floodset also assumes a reliable network, although messages may not be received if the receiver has failed. The number of rounds executed is a parameter of Floodset, and up to t processor failures may be tolerated, provided $t+1$ rounds are executed, and the original number of processors is greater than t.

The Floodset algorithm proceeds as follows. Each process[1] begins with an initial value. In the first round, every process sends its identity and value to all other processes. Processes retain all received *(*process, value) pairs. In each subsequent round, all processes send all the pairs they currently know[2]. Faulty processors may fail at any time.

Fig. 1 depicts the first two rounds of an example execution of Floodset on processors p_1, p_2 and p_3. Fig. 1(a) gives the initial state of the three processes. During the first round, processor p_2 fails after process p_2 has sent its name and value to process p_3, but before sending them to process p_1. It receives nothing from either of the other processes. Processes p_1 and p_3 communicate fully with each other. The state after the first round is given in Fig. 1(b). During the second round, processes p_1 and p_3 again communicate fully, leading to the state shown in Fig. 1(c).

After $t+1$ rounds have been carried out, each process arrives at its final value by running a deterministic decision function on its final (local) state, which selects one of the values known to that process. To demonstrate that each correct process arrives at the same value, it is sufficient to ensure that the initial inputs each correct process provides to the deterministic decision function are identical.

To see that Floodset is correct, recall that $t+1$ rounds are executed, up to t failures are tolerated, and that failures are failstop (failed processes do not resume execution.) There must therefore be a round in which no failures occur. After this round (which we refer to later as the *saturation* round) all working processes (a superset of correct processes) must have the information, and this information cannot be added to at later rounds in the protocol. Each correct process therefore has the same input at decision

[1] A process is assumed to run on a single processor, and we therefore conflate process and processor, referring to both in the subsequent text as p_i.

[2] A version of Floodset can be implemented which sends only values, and omits process names. Process names are included to make this model more reusable in the future development of more complex consensus algorithms.

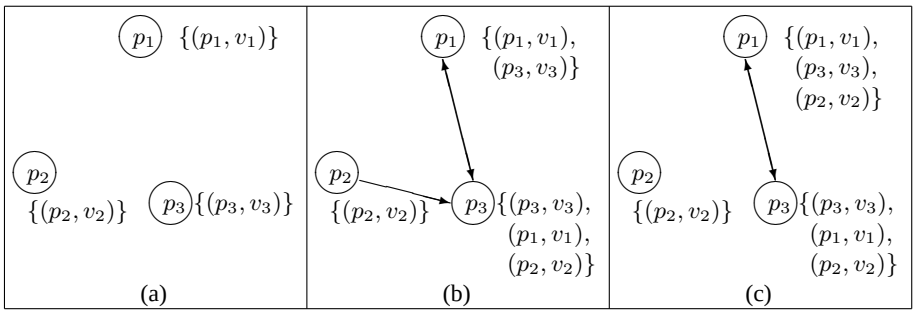

Fig. 1. Example initial rounds of Floodset with three processors.

time, and the same value will be reached. Formalising this argument to derive a precise specification of the state information after each round forms the second part of the formal development in Sect. 4.2 – 4.5.

3 Event-B

A Event-B [1] model is composed of a sequence of *machines*, each of which (apart from the first) is linked to its predecessor by a *refinement* relation. A machine contains *variables* modelling state data, *invariants* which restrict the possible values of variables, and *events* which change the values of variables. An event consists of *guards*, which must be true in order for the event to occur, and *actions*, in which the values of variables are changed. Events may be parameterised, and in general an event takes the form

eventname
any p **where**
$G(p, v)$
then
$S(p, v)$
end

where p are the event parameters, v are the state variables of the machine, G is a list of guards and S is the list of actions, made up of one or more assignments to variables. Each machine may have associated *carrier sets* and *constants*, which are held in a *context* visible to the machine. A context may be *extended* by another context, visible to subsequent machines in the sequence of refinements.

Proof obligations allow us to establish the internal consistency of individual machines, and the validity of the refinement relation between machines. *Invariant preservation* is the proof obligation that requires each invariant to continue to hold whenever any event occurs.

For any step in the refinement chain, the relationship between the variables in the abstract model and the variables in the concrete model is given by a *gluing invariant*.

To show that an event in the concrete model refines an event in the abstract model, it must be shown that the guards of the concrete event imply the guards of the abstract event, and that the variable states reached after the occurrence of the concrete and the abstract event are linked by the gluing invariant.

Proof obligations are generated and in some cases proved automatically by the Rodin Tools [2]. Those that are not proved automatically may be discharged with the help of the interactive theorem prover.

4 Development

The approach taken to the development has three stages[3]. The first stage is the specification of the result of a successful run of any consensus algorithm by giving an abstract description of the chosen consensus properties above. This stage is independent of the algorithm chosen and corresponds to the initial machine in the development ($X0$).

The proof of the agreement property relies on the fact that the local views of correct processes are identical at the end of any execution. To show this, we show that the views of all working processes become equal before the end of an execution, and do not change for the remainder of that execution.

The second stage derives a precise specification of the behaviour of each round of Floodset by formalising the informal proof of correctness given in Sect. 2. The first machine in this stage ($X1$) introduces the round structure of the algorithm, and identifies three *phases* in the execution. The *saturation* round is a separate phase, and is the first round in which no processes fail. All preceding rounds are part of the *pre-saturation* phase, and all subsequent rounds are part of the *post-saturation* phase. The specification of the three phases therefore varies according to phase.

Within an execution, a process cannot know which phase it is in, as phase is a global notion and not a local one. The final specification of the round behaviour must not therefore vary according to phase. However, identifying the phase facilitates our development and proof, so phase distinctions are introduced in $X1$ and used in $X2$ and $X3$. The stronger guards in the saturation round specification in $X1$ play an important role in proving the key invariant at the end of Sect. 4.2. In refinement $X2$ the set $live$ is identified, which is the faulty processes still working during the saturation round. Refinement $X3$ makes further use of the fault assumptions to define a function between round numbers and the processes which fail in that round. This brings the phase descriptions to the point where they are equal but for the phase information. Refinement $X4$ then merges the three events together, producing a common specification for the behaviour of each round of Floodset. The final stage is the final refinement ($X5$) in which the sending and receiving behaviour of individual processes is introduced and an abstract network description is given. It is shown by refinement that this description meets the specification deduced in $X4$.

The first two consensus properties (agreement and validity) are established in $X0$ and demonstrated to hold throughout the development using refinement. The third prop-

[3] The model is available at http://deploy-eprints.ecs.soton.ac.uk/. A Technical Report version of this paper is available [4].

erty (termination) is shown by model-checking the completed development. Termination was therefore shown using ProB, a model-checker for Event-B [12].

4.1 The initial machine

The purpose of the initial machine is to define the success conditions for Floodset. We begin with some terminology. The distributed system considered contains a finite set of processes, P. Each process p in P has an initial value given by $INIT(p)$ and drawn from a set V, which is proposed to its peers as a possible final value. The set $CORR \subseteq P$ is the set of processes which behave correctly throughout the execution of the algorithm.

After the execution of Floodset each correct process p_c has a *v*iew – a set containing all the learned *(*process, value) pairs. The function M gives the final view of each correct process. At termination, each correct process p_c runs the decision function on $M(p_c)$.

The initial machine contains a single success event floodset (see Fig. 2) which will fire when the correctness properties hold. On firing, floodset assigns a value to M which is a correct final outcome of the Floodset algorithm – the properties defining consensus hold over M.

The guards on the floodset event define the correctness conditions by imposing restrictions on the event parameter m, which is then assigned to the final views M. The first guard gives the type of m, which is the same as the variable M: it assigns views to correct processes. The second guard establishes the first two consensus properties. f and g are two arbitrary views from m. The first conjunct of the consequent of guard 2 ensures that these are equal, which is a sufficient condition for the consensus property of agreement. To ensure validity, the second conjunct ($CORR \lhd INIT \subseteq f$) requires that a process is aware of the initial values of all correct processes and the third ($f \subseteq INIT$) conjunct requires that no incorrect values (i.e. ones not in $INIT$) are present in any final view. We assume that the decision function picks one of values given in the final view.

floodset
any m **where**
(1) $m \in CORR \rightarrow (P \nrightarrow V)$
(2) $\forall f, g \cdot (g \in ran(m) \wedge f \in ran(m)) \Rightarrow$
$f = g \wedge CORR \lhd INIT \subseteq f \wedge f \subseteq INIT$
then
$M := m$
end

Fig. 2. The floodset event in the initial machine.

The third consensus property, that of termination, is a consequence of the firing of floodset, rather than being a precondition to its firing. We establish this for the final development using the model checker ProB [12].

4.2 The first refinement: introducing phase specifications

The first refinement begins the second development stage, in which a specification for the behaviour of a round of Floodset is derived. A different specification is introduced for each of the three phases. Recall that in an execution of Floodset, the *saturation* round is the first round in which no failures occur. In it all currently working processes will learn all known information. Any rounds before the saturation round are modelled by the event presat. The saturation round is modelled by the event saturation, and rounds after the saturation round are modelled by the event postsat. The saturation round may be any round in an execution. We cannot tell in advance which round will be the saturation round, only that there will be one.

In this refinement a progress counter r is introduced. When $r \in 1..t+1$ it records the current round number. $r = t+2$ when the final round is completed, and $r = t+3$ when the floodset event has taken place.

The saturation round is labelled j, where j, where $j \in 1..t+1$. In the presat rounds $r < j$ and in the postsat rounds $r > j$. Since presat, postsat and saturation are new events they are considered to refine the skip event. The final event is floodset, a refinement of floodset in the previous machine. Machines $X0$ and $X1$ and the refinement relationship between them are summarised in Fig. 3.

X0	skip	skip	skip	floodset
	⇑	⇑	⇑	⇑
X1	presat	saturation	postsat	floodset
	$r < j$	$r = j$	$r > j$ ∧ $r < t+2$	$r = t+2$

Fig. 3. The refinement relationship between the first two machines.

During execution, each process maintains a working view of the information it has received. These working views are given by the global variable $W \in P \rightarrow (P \nrightarrow V)$. Initially $W(p) = \{(p, INIT(p))\}$, since each process begins knowing its own value.

In each round, each process p sends $W(p)$ to all other processes, and at the end of each round W is updated.

The presat event (Fig. 4) defines the intermediate view W for pre-saturation rounds (guard 1). The parameter new gives all the information received by each process during the round. This could include information already known to the process. The only restriction on new is given in guard 4 – no process is sent false information. The parameter w is the updated state of the views of each process when this new information is received (guard 5). It is assigned to the working view W.

presat
any new, w **where**
(1) $r < j$
(2) $new \in P \rightarrow (P \nrightarrow V)$
(3) $w \in P \rightarrow (P \nrightarrow V)$
(4) $\forall p \cdot p \in P \Rightarrow new(p) \subseteq INIT$
(5) $\forall p \cdot w(p) = W(p) \cup new(p)$
then
$W := w$
$r := r + 1$
end

saturation
any f, w **where**
(1) $r = j$
(2) $w \in P \rightarrow (P \nrightarrow V)$
(3) $f \in (P \nrightarrow V)$
(4) $f \subseteq INIT$
(5) $CORR \lhd INIT \subseteq f$
(6) $\forall g \cdot g \in ran(CORR \lhd w) \Rightarrow f = g$
then
$W := w$
$r := r + 1$
end

Fig. 4. The presat and saturation events in the first refinement.

After the saturation event (Fig. 4), every correct process will have the same view. At this level of abstraction, it is not possible to give a precise specification of this view, but some restrictions may be identified. The parameter f is a view of an arbitrary process. Guard 4 requires that it may include only correct information (information from the initial state) and guard 5 requires that it must include the proposed values of all correct processes. The parameter w has the same purpose as in the presat event – to identify the updated value of W – but in the saturation round more precise restrictions can be placed on w. Since no processor fails in this round, all currently working processes send and receive all their information successfully. After this round all currently working processors will therefore have the same view. It is not possible at this level of abstraction to identify the set of currently working processes precisely, but it must contain the set of correct processes. Thus the only values of W allowed after saturation are those in which all correct processes share the same view (given by the parameter f). This view must be shared by at least all the correct processes (guard 6.)

Since no process can now learn new information, (and therefore W cannot change further) the postsat event simply increments the round counter until the remaining rounds have been completed.

The refined floodset event (not given) simply increments the round counter after the final round.

The invariant below moves the correctness criteria from the floodset event in the previous machine and shows that the first two consensus properties hold for all rounds following the saturation round (rounds in which $r > j$).

$$
\begin{aligned}
r > j \Rightarrow (\exists f \cdot (& f \in (P \nrightarrow V) \wedge \\
& CORR \lhd INIT \subseteq f \wedge \\
& f \subseteq INIT \wedge \\
& (\forall g \cdot g \in ran(CORR \lhd W) \Rightarrow f = g)))
\end{aligned}
$$

4.3 Identifying live processes: $X2$

This refinement introduces no new state, but looks more closely at the existing state variable W and strengthens the set of invariants relating to it (Fig. 5). The first invariant gives an upper bound on the information known by a process. It states that all (process,value) pairs known by any process must be valid, in the sense that they are given by the original function $INIT$. This excludes the possibility of a process learning false information at any stage. The second invariant states that every process is aware of its own initial value.

$$
\begin{array}{ll}
(1) & \forall p \cdot p \in dom(W) \Rightarrow W(p) \subseteq INIT \\
(2) & \forall p \cdot p \in dom(W) \Rightarrow p \mapsto INIT(p) \in W(p)
\end{array}
$$

Fig. 5. Invariants in X2.

The set of processes which fail in an execution is defined as the $FLT = P \backslash CORR$. The number of failing processes must not be more than the number of faults (failing processes) that can be tolerated: $t \geq card(FLT)$.

presat
refines $presat$
any new, w **where**
(1) $r < j$
(2) $new \in P \rightarrow (P \nrightarrow V)$
(3) $w \in P \rightarrow (P \nrightarrow V)$
(4) $\forall p \cdot p \in CORR \Rightarrow$
$CORR \lhd INIT \subseteq new(p)$
(5) $\forall p \cdot p \in CORR \Rightarrow new(p) \subseteq INIT$
(6) $\forall p \cdot p \in FLT \Rightarrow new(p) \subseteq INIT$
(7) $\forall p \cdot w(p) = W(p) \cup new(p)$
then
$W := w$
$r := r + 1$
end

postsat
refines $postsat$
any new, w **where**
(1) $r > j$
(2) $r \leq t+1$
(3) $w \in P \rightarrow (P \nrightarrow V)$
(4) $new \in P \rightarrow (P \nrightarrow V)$
(5) $\forall p \cdot p \in P \Rightarrow new(p) \subseteq W(p)$
(6) $\forall p \cdot p \in P \Rightarrow$
$w(p) = W(p) \cup new(p)$
then
$W := w$
$r := r + 1$
end

Fig. 6. The presat and postsat events in $X2$.

The event presat is now refined to the description given in Fig. 6. Since processes may fail during these rounds and therefore fail to send or receive information the value of new is non-deterministic. Guard 4 gives a lower bound: each correct process receives

saturation
refines $saturation$
any $w, live$ **where**
(1) $r = j$
(2) $w \in P \rightarrow (P \nrightarrow V)$
(3) $live \subseteq FLT$
(4) $\forall p \cdot p \in CORR \cup live \Rightarrow w(p) = union(W[(CORR \cup live)])$
(5) $\forall p \cdot p \in FLT \setminus live \Rightarrow w(p) = W(p)$
(6) $\forall p, q \cdot \{p, q\} \subseteq CORR \Rightarrow w(p) = w(q)$
then
$W := w$
$r := r + 1$
end

Fig. 7. The saturation event in $X2$.

information from all correct processes. Guards 5 and 6 give an upper bound for new for correct and faulty processes respectively – in each case the process receives only valid information. The final guard creates the new value for W using the parameter w.

The refined postsat event (Fig. 6) adds the restriction that no process learns anything new after saturation (guard 6).

The saturation event (Fig. 7) gives the value of W after this round more precisely. The faulty processes which are currently working at the time of the saturation round are identified using the parameter $live \subseteq FLT$ (guard 3). After this round, all currently working processes ($CORR \cup live$) know all that each currently working process knows (guard 4.) Processes that have already failed (those in $FLT \setminus live$) learn nothing new (guard 5). We remove the parameter f from saturation to make it more consistent with the definitions of presat and postsat. The refinement is performed using the witness $f = union(W[CORR \cup live])$. The property that all correct processes have the same view after saturation is therefore recorded in a different way in guard 6.

4.4 Homogenising the events: $X3$

The events presat, saturation and postsat are still unimplementable, as they rely on processes knowing in advance which round will be the saturation round. Over this refinement ($X3$) and the next ($X4$) this reliance on the global saturation variable is removed by merging these three events into a single event which does not depend on j. The purpose of this refinement is to finally "set up" this merging by providing versions of the three events in which each event has the same guards and actions (excluding those guards which refer to j). The subsequent refinement then merges these three events into a single event which does not rely on j.

To do this, the set of processes are considered more carefully and those which will fail in each round are identified. The function d (in context $X3\_ctx$) maps each round to the set of processes which fail in that round, and is defined by axioms 1–4 in Fig. 8.

Axiom 1 gives the type of d, and axiom 2 ensures that no process can fail in two separate rounds. All processes in FLT will fail (axiom 3), and no process fails in the saturation round (axiom 4). The function d may be any function that satisfies these axioms.

(1) $d : 1..t+1 \rightarrow \mathbb{P}(FLT)$
(2) $\forall i, k \cdot i \in dom(d) \wedge k \in dom(d) \wedge i \neq k \Rightarrow d(i) \cap d(k) = \varnothing$
(3) $\forall p \cdot p \in FLT \Rightarrow (\exists i \cdot i \in dom(d) \wedge p \in d(i))$
(4) $d(j) = \varnothing$
(5) $dead : 1..t+2 \rightarrow \mathbb{P}(FLT)$
(6) $dead(1) = \varnothing$
(7) $\forall i \cdot i \in dom(dead) \wedge i \geq 2 \Rightarrow dead(i) = union(d[(1 \,..\, (i-1))])$

Fig. 8. The axiomatic definition of d and $dead$ in context $X3\_ctx$.

The helper function $dead$ is defined using d in axioms 5–7 in Fig. 8. For each round $dead$ returns all the processes that have failed prior to the start of that round (axiom 7). We assume that no processes fail before the start of the execution (axiom 6).

The descriptions of the events presat, saturation, postsat differ only by their second guard ($r < j, r = j, r > j$ respectively). The definition of presat is given in Fig. 9. The guards distinguish three disjoint sets of processes, depending on whether they will work correctly throughout that round, will fail at some point during the round, or have failed already.

For the first two sets the guards give an upper and lower bound on the new information that can be received in a round. All working processes send and receive to themselves. Since the processes which fail before round r are given by $dead(r)$, processes working at the start of a round r are given by $CORR \cup (FLT \setminus dead(r))$. The processes working correctly at the end of round r are given by $CORR \cup (FLT \setminus dead(r+1))$.

Guard 5 of presat states that the most information a process which works for the whole round may learn is $union(W[CORR \cup (FLT \setminus dead(r))])$. In this case all processes in $d(r)$ transmit all messages before failing. Guard 6 states that the least information a process working for the whole round will receive is everything known by any process which survives the round. In this case all processes in $d(r)$ fail before sending any messages. Processes which have failed before this round and are no longer communicating will learn nothing in this round (guard 7). Guard 8 states that processes in $d(r)$ may learn as much as the processes which continue to function correctly for the whole round. In the worst case, they will fail before receiving any information (guard 9). As previously, the new value for W is identified as w (guard 10). Apart from guard 1, these guards are now identical for each of the three round events.

The invariant on W can now be strengthened, and is given below. It states that every process still operating after the saturation round learns nothing new after the saturation round. The common information known at the saturation round is given by $union(W[CORR \cup (FLT \setminus dead(j))])$.

presat
refines $presat$
any w, new **where**
(1) $r < j$
(2) $r < t+2$
(3) $w \in P \rightarrow (P \nrightarrow V)$
(4) $new \in P \rightarrow (P \nrightarrow V)$
(5) $\forall p \cdot p \in CORR \cup (FLT \setminus dead(r+1)) \Rightarrow$
$new(p) \subseteq union(W[CORR \cup (FLT \setminus dead(r))])$
(6) $\forall p \cdot p \in CORR \cup (FLT \setminus dead(r+1)) \Rightarrow$
$union(W[(CORR \cup (FLT \setminus dead(r+1)))]) \subseteq new(p)$
(7) $\forall p \cdot p \in dead(r) \Rightarrow new(p) = \varnothing$
(8) $\forall p \cdot p \in d(r) \Rightarrow new(p) \subseteq union(W[CORR \cup (FLT \setminus dead(r))])$
(9) $\forall p \cdot w(p) = W(p) \cup new(p)$
then
$W := w$
$r := r + 1$
end

Fig. 9. The event presat in $X3$.

$$\forall p \cdot r \in dom(dead) \wedge p \in CORR \cup (FLT \setminus dead(r)) \wedge r > j \Rightarrow W(p) = union(W[CORR \cup (FLT \setminus dead(j))])$$

4.5 Refining out the saturation assumption: $X4$

In this refinement, the floodset event remains unchanged and the three events presat, saturation and postsat are *merged* into the single event round (Fig. 10). To perform the merging, it must be shown that the concrete guards of round imply the disjunction of the guards of the merged events. The guards of round are identical to the guards of the three events in the previous refinement, except that the second guard has been removed, so the proof reduces to proving the trivial theorem $r < j \vee r = j \vee r > j$.

This round event is now a sufficiently detailed description of a single round of the algorithm to allow an implementation to be developed and a possible implementation is shown in the next section.

4.6 Implementing the round event: $X5$

The round event provides a global specification of the desired behaviour of Floodset at each round. The purpose of this refinement is to define the local behaviour of individual processes. A message passing network model is also introduced.

round
refines $presat, saturation, postsat$
any new, w **where**
(1) $r < t + 2$
(2) $w \in P \rightarrow (P \nrightarrow V)$
(3) $new \in P \rightarrow (P \nrightarrow V)$
(4) $\forall p \cdot p \in CORR \cup (FLT \setminus dead(r+1)) \Rightarrow$
$new(p) \subseteq union(W[CORR \cup (FLT \setminus dead(r))])$
(5) $\forall p \cdot p \in CORR \cup (FLT \setminus dead(r+1)) \Rightarrow$
$union(W[(CORR \cup (FLT \setminus dead(r+1)))]) \subseteq new(p)$
(6) $\forall p \cdot p \in dead(r) \Rightarrow new(p) = \varnothing$
(7) $\forall p \cdot p \in d(r) \Rightarrow new(p) \subseteq union(W[CORR \cup (FLT \setminus dead(r))])$
(8) $\forall p \cdot w(p) = W(p) \cup new(p)$
then
$W := w$
$r := r + 1$
end

Fig. 10. The event round in $X4$.

A round is now split into three phases: $sending$, $receiving$, and $restarting$. In the $sending$ phase messages are sent to the network middleware. In the $receiving$ phase all the messages for each process are sent to that process. The $restarting$ phase is used to reset the state of processes after a round. The variable $phase$ records the phase of the round.

The point at which a process fails is now identified more accurately using the variables die_in_send and die_in_rec. No process sends or receives messages in the $restarting$ phase, so a process which fails during $restarting$ may be considered to have failed during $receiving$, after all messages have been sent. The important axioms are

$$\forall i \cdot i \in 1 \mathinner{..} t+1 \Rightarrow die\_in\_send(i) \cap die\_in\_rec(i) = \varnothing$$
$$\forall i \cdot i \in 1 \mathinner{..} t+1 \Rightarrow die\_in\_send(i) \cup die\_in\_rec(i) = d(i)$$

The network middleware is given the variable mw, where $mw(p)$ is the set of all $(process, value)$ pairs that have been sent to process p. A process p records the processes to which it has sent messages as $sent(p)$.

The $sending$ phase consists of multiple occurrences of the send event (Fig. 11), each parameterised by the sender (fr) and receiver (to). The only processes unable to send information in round r are the ones which have already failed (given by $dead(r)$), so fr may be drawn from any other process (guard 3). Processes do not maintain a record of their failed peers, so each process sends to all other processes. It would also be possible to design a "failure aware" algorithm in which a process learns about and records failed peers, and does not send to processes it knows have failed. The end of the $sending$ phase is marked by a phase transition event (not given.)

send
any fr, to **where**
(1) $r < t + 2$
(2) $phase = sending$
(3) $fr \in CORR \cup (FLT \setminus dead(r))$
(4) $to \in P$
(5) $to \notin sent(fr)$
then
$mw(to) := mw(to) \cup W(fr)$
$sent(fr) := sent(fr) \cup \{to\}$
end

rec
any p **where**
(1) $r < t+2$
(2) $p \in CORR \cup (FLT \setminus dead(r+1)) \cup die\_in\_rec(r)$
(3) $p \notin received$
(4) $phase = receiving$
then
$received := received \cup \{p\}$
$W\_part(p) := W\_part(p) \cup mw(p)$
end

Fig. 11. The events send and rec in $X5$.

In the rec event (Fig. 11) one of the working processes receives all its amalgamated information from the middleware in a single message. The processes which are working at the start of the $receiving$ phase are given by the invariant

$$receiving \subseteq CORR \cup (FLT \setminus dead(r+1)) \cup \ die\_in\_rec(r)$$

and any of these processes may receive from the middleware (guard 2). Those that will fail during this round ($die\_in\_rec(r)$) may or may not receive from the middleware before they fail. Receiving processes are added to the set $received$, which is local to the middleware. W_part is a temporary variable, which contains the partially updated view of W during the $receiving$ phase.

The end of the $receiving$ phase is marked by the event end_rec_phase in Fig. 12 which assigns the partial view W_part to W, and refines the event round from the previous refinement. It may fire once all working processes have received messages from the middleware (guard 3).

end_rec_phase
refines $round$
when
(1) $r < t+2$
(2) $phase = receiving$
(3) $(CORR \cup (FLT \setminus dead(r+1))) \subseteq received$
then
$W := W\_part$
$r := r + 1$
$phase := restarting$
end

Fig. 12. The end_rec_phase event in $X5$.

A number of implementation issues remain open. In particular, events which mark the end of the sending or receiving phase are global specifications, using global variables in the guards. The event end_rec_phase refers to $received$, which suggests that processes have knowledge of the internal state of the middleware. In reality this reliance would be removed by implementing these events locally as time-triggered events on each processor. The functions d and $dead$ could be removed from the specification using a description of an explicit fault injector in the network model.

5 Discussion and Conclusions

We have demonstrated the stepwise refinement in the development of a well-known consensus algorithm, Floodset. The initial, most abstract model captured the three generic consensus properties in Sect. 2. The first two (agreement and validity) are demonstrated by construction. They are captured in the initial abstract model, and shown by refinement to continue to hold at each step.

The third property, that all correct processes eventually reach a decision, may be shown by demonstrating deadlock freeness — that each model in a development (apart form the first) does not deadlock more often than its predecessor. That is, the only execution paths permitted are those which eventually satisfy the most abstract specification in the refinement chain. In this development, the description of rounds in $X1$ is deliberately more non-deterministic than necessary. The second refinement introduces no new state, so all properties introduced in $X2$ could have been introduced in $X1$. The more restrictive invariants in $X2$ mean fewer execution paths, and therefore deadlock freedom cannot be proved at this step. However, this refinement is carried out over two steps to simplify the proofs involved at each stage. Termination was therefore shown using the ProB [12] model-checker. It was shown that the development has not introduced a possible deadlock where the floodset event cannot eventually occur. This was shown for three processes with arbitrary initial state, by checking the truth of the temporal logic proposition $\mathbf{F}[\mathsf{floodset}]$ (eventually the floodset event occurs).

A good level of automatic proof ($> 75\%$) is achieved, given the complexity of the development. However the manual proof overhead is still relatively high, and this may lead away from the goal of reusable models and proofs.

A number of decision points were identified during the development. Each of these is a potential point of branching, and so using this development as a basis for a branching taxonomy seems to be a promising approach. On the other hand, the manual proof effort required by this work may be too high to be reused in more complex developments. This work sought to provide a reuseable platform for the development of consensus algorithms with weaker failure and network models and so the algorithm transmits sets of $(process, value)$ pairs, rather than just values. Refactoring the development to use more simple datatypes may lead to improved levels of automatic proof, and therefore improve the potential for reuse. A further possibility is to split the final refinement step to introduce the network model and the individual processes separately.

Floodset relies on the assumption that processes can only fail by stopping entirely. Allowing Byzantine failures naturally leads to more complex algorithms. An interesting intermediate case is to allow only authenticated messages between processes. Further-

more, Floodset relies on a synchronous timing model and is a round-based algorithm, and the development here makes use of that structure. Algorithms developed for asynshronous timing models are less structured, and developing such models using stepwise refinement is a more challenging task. We will investigate these alternative network and timing models using the Byzantine Generals algorithm [11] and the Paxos algorithm [10].

Acknowledgements: This work was supported by the EU Integrated Project DEPLOY (www.deploy-project.eu/) and by the EPSRC Platform Grant TrAmS. John Fitzgerald suggested this line of research. Thanks also to Sascha Romanovsky and Alexei Iliasov, and to the anonymous reviewers who made a number of suggestions which led to improvements in the work.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 2010.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An Open Extensible Tool Environment for Event-B. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer Berlin / Heidelberg, 2006. 10.1007/11901433_32.
3. Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.
4. Jeremy Bryans. Developing a consensus algorithm using stepwise refinement. Technical Report CS-TR-1228, Newcastle University, Dec. 2010.
5. Dominique Cansell and Dominique Méry. Formal and incremental construction of distributed algorithms: On the distributed reference counting algorithm. *Theoretical Computer Science*, 364(3):318 – 337, 2006. Applied Semantics.
6. Bernadette Charron-Bost and Stephan Merz. Formal Verification of a Consensus Algorithm in the Heard-Of Model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.
7. Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22:49–71, 2009.
8. Thai Son Hoang, Hironobu Kuruma, David A. Basin, and Jean-Raymond Abrial. Developing Topology Discovery in Event-B. In Michael Leuschel and Heike Wehrheim, editors, *IFM*, volume 5423 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2009.
9. Roman Krenický and Mattias Ulbrich. Deductive verification of a byzantine agreement protocol. Technical report, Karlsruhe Institute of Technology, April 2010.
10. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
11. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
12. Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer Berlin / Heidelberg, 2003.
13. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1st edition, March 1997.
14. Christoph Sprenger and David Basin. Developing security protocols by refinement. In *17th ACM Conference on Computer and Communications Security (CCS 2010)*, 2010.
15. Ninh-Thuan Truong, Thanh-Binh Trinh, and Viet-Ha Nguyen. Coordinated consensus analysis of multi-agent systems using Event-B. In *Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 201–209, 2009.