Proving Distributed Algorithm Correctness using Fault Tolerance Bisimulations

Adrian Francalanza[1] and Matthew Hennessy[2]

[1] Imperial College, London SW7 2BZ, England, adrianf@doc.ic.ac.uk
[2] University of Sussex, Brighton BN1 9RH, England, matthewh@sussex.ac.uk

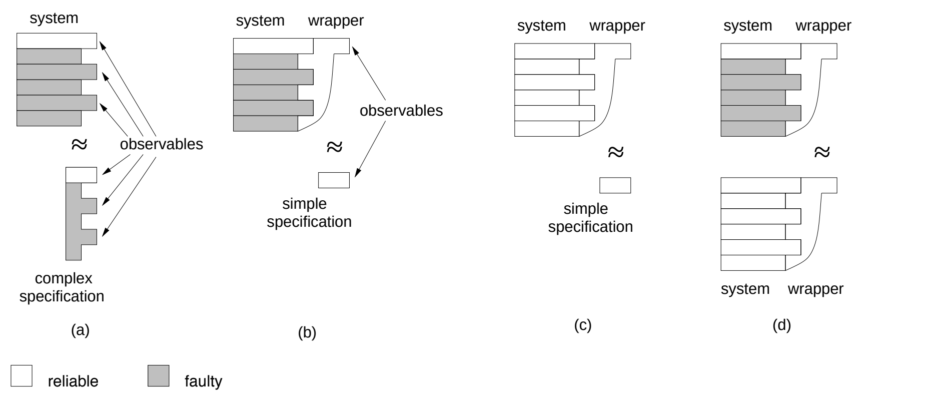
Abstract. The possibility of partial failure occuring at any stage of computation complicates rigorous formal treatment of distributed algorithms. We propose a methodology for formalising and proving the correctness of distributed algorithms which alleviates this complexity. The methodology uses fault-tolerance bisimulation proof techniques to split the analysis into two phases, that is a failure-free phase and a failure phase, permitting separation of concerns. We design a minimal partial-failure calculus, develop a corresponding bisimulation theory for it and express commit and consensus algorithms in the calculus. We then use the consensus example and the calculus theory as the framework in which to demonstrate the benefits of our methodology.

1 Introduction

The areas of Distributed Systems and Process Calculi are two (major) areas in Computer Science addressing the same problems but ”speak(ing) different languages” [14]. In particular, seminal work in Distributed Systems, such as [2, 11] present algorithms in semi-formal pseudo-code and correctness proofs of an informal algorithmic nature. Attempts at applying the rigorous theory of process calculi to formal proofs for standard distributed algorithms are not popular because of the sheer size of the resulting formal descriptions, which leads to an explosion in the complexity of related proofs. This problem is accentuated when failures are considered: these typically occur at any point during the execution of the algorithm and can potentially affect the progress of execution. To tame such complexity, attempts at formalising distributed algorithm proofs often make use of mechanised theorem provers [8] or translations into tailor-made abstract interpretations [14]. In spite of their effectiveness, such tools and techniques tend to obscure the natural structure of the proofs of correctness, because they either still produce large one-chunk proofs that are hard to digest with the naked eye, or else depart from the source formal language in which the algorithm is expressed.

We propose an alternative methodology to formally prove correctness of distributed algorithms in the presence of failure, based on well-studied bisimulation techniques. In a process calculus with a labelled transition system (lts) formal semantics and corresponding bisimulation equivalence $\approx$, bisimulation proofs generally consist in comparing the distributed algorithm, described in the base calculus, to a concise correctness specification, also defined in the base calculus, using $\approx$ (Table 1(a)). The required witness relations satisfying this direct global approach turn out to be substantial, even for

Table 1. *Correctness proofs using fault-tolerant bisimulation techniques*



the simplest of algorithms and specifications. Even worse, in a setting with partial failure, the simplicity of the correctness specification is often muddled by the different observable behaviour the algorithm exhibits when failure occurs.

We propose a methodology to solve the problem of complex specifications, based on a common assumption that some processes are assumed to be reliable, thus immortal. More specifically, failure can affect behaviour either *directly*, when the process that produces the observable effect itself fails, or *indirectly*, when a process produces an observable behaviour which depends on an internal interaction with a secondary process which in turn fails. By using wrapper code around the algorithm being analysed and intentionally limiting observations to reliable processes *only*, we reformulate the equivalence described earlier into a comparison between the re-packaged algorithm and a *simpler* specification, only specifying behaviour from reliable processes (Table 1(b)). Global specifications can thus be decomposed into simpler specification, encoded as dedicated wrappers each testing for separate aspects of the system, which are easier to formulate and verify against the expected behaviour.

This reformulation carries more advantages than merely decomposing the specification and shifting some of the complexity of the equivalence from the specification side to the algorithm side in the form of wrappers. A specification which exclusively deals with behaviour that is only *indirectly* affected by failure permits *separation of concerns* by tackling the comparison as a *fault-tolerance* problem. By this, we mean that we can decompose our reformulated equivalence into two sub-equivalences. In the first sub-equivalence, we compare the specification with the behaviour of the repackaged algorithm in a *failure-free* setting (Table 1(c)); this allows us to temporarily ignore failures and use ”standard” bisimulations. In the second sub-equivalence we compare the behaviour of the repackaged algorithm in the failure-free setting with the repackaged algorithm itself in the failure setting (Table 1(d)), to ensure that the expected behaviour, already tested for in the first sub-equivalence, is *preserved* when failure occurs. Apart from decomposing the proof into two sub-proofs, which can be tested independently and which, we argue, is a natural way how to tackle such proof, the fault-tolerance reformulation carries further advantages. For a start, the first equivalence is considerably easier to prove, and can be treated as a vetting test before attempting the more involving second proof. Moreover, when proving the second equivalence, which compares the algorithm with itself but under different conditions, we can exploit the common structure on both sides of the equivalence to construct the required witness bisimulation.

Our proposed methodology goes one step further and uses (permanent) failure to reduce the size of witness bisimulations in two ways. First, we note that while permanent failure may affect the behaviour of the remaining live code, it also *eliminates* the transitions from dead code. Thus, by developing appropriate abstractions to represent dead code, we can greatly reduce the presentation size of bisimulations. Second, we note that distributed algorithms tolerate failure (and preserve the expected behaviour) through the use of *redundancy* which is usually introduced in the form of *symmetrical replicated code*. As a result, such algorithms are often characterised by a considerable number of transitions that are similar in structure in our witness bisimulation. This, in turn, gives us scope for identifying a subset of such similar transitions which are *confluent* and develop up-to techniques that abstract over these confluent moves. The range of replication patterns are arguably bounded and are reused throughout a substantial number of fault-tolerant algorithms, which means that we expect these up-to techniques to be applicable, at least in part, to a range of fault-tolerant distributed algorithm. But even when this is not the case, and some of these confluent moves appear to be specific to the algorithm in question, we still argue that the technique of identifying confluent moves and developing related up-to techniques is a worthwhile endeavour towards our end. The frequent occurrence of these confluent transitions in the algorithm means that the development of such up-to techniques greatly alleviates the burden of exhibiting witness bisimulations in our proofs. More importantly however, they promote the (non-confluent) transitions that really matter, making the bisimulation proofs easier to understand.

The remaining text is structured as follows. In Section 2 we introduce our language. In Section 3 we express atomic commit and consensus algorithms in our calculus and show how to express the correctness of the latter algorithm as a fault-tolerance problem (consensus has long been considered as such [4]). In Section 4 we develop up-to techniques for our algorithm and in Section 5 we give its proof of correctness.

2 Language

Our *partial-failure* calculus is inspired by [15] and consists of processes from a subset of CCS[12], distributed across a number of failing locations. We assume a set of $\textsc{Act}$ of communicating actions a, b constructed from a set of names $\textsc{Names}$, such that for every name $a \in \textsc{Names}$ we have a complement $\overline{a}$ and both $a, \overline{a} \in \textsc{Act}$. ($\overline{\cdot}$ is a bijection on $\textsc{Act}$); α ranges over strong actions, defined as $\textsc{Act} \cup \{\tau\}$, including the distinguished silent action τ. We also assume a set $\textsc{Locs}$ of locations l, k which also includes the immortal location $\star$.

Processes, defined in Table 2, can be guarded by an action, composed using choice, composed in parallel or scoped. As in [15], only actions can be scoped (not locations). By contrast to [15], we here simplify the calculus and disallow process constants and replication (thus no recursion and infinite computation) and migration of processes (thus

Table 2. *Syntax*

Processes

$$P,\ Q ::= \alpha.P \quad \textit{(guard)} \quad \mid \ P+Q \quad \textit{(choice)} \quad \mid \ (\nu a)P \quad \textit{(scoping)} \ \mid \ P|Q \quad \textit{(fork)}$$
$$\mid \ \mathsf{fail}\ k.P \quad \textit{(failure detector)}$$

Systems

$$M,\ N ::= l[\![P]\!] \quad \textit{(located)} \ \mid \ N|M \quad \textit{(parallel)} \ \mid \ (\nu\, a)N \quad \textit{(scoping)}$$

no change in failure dependencies). Another important departure from [15] is that instead of *ping* we use a guarding construct fail $l.P$, already introduced in [5], which *tests* for the status of l and releases P once l dies. Prior programming experience [6, 7] has shown that the latter is more useful in a setting with dynamic fail-stop failures since ping only yields *snapshot* liveness information that may be immediately outdated by a subsequent dynamic fail. Systems, also defined in Table 2, are *located* processes composed in parallel with channel scoping. Our calculus is called *partial-failure* and not *distributed* because distributed action synchronisations are permitted. This translates to a tighter synchronisation assumption across locations, which merely embody *units of failure*. Nevertheless, choices across locations are disallowed because their implementation would still be problematic in a dynamic partial-failure setting.

Notation: We denote a series of parallel processes $P_1|\ldots|P_n$ as $\prod_{i\in I} P_i$ and a series of choices $P_1+\ldots+P_n$ as $\sum_{i\in I} P_i$ for $I = \{1,\ldots,n\}$. The inactive process $\sum_{\emptyset} P_i$ is written as $\mathbf{0}$ and we omit the final $\mathbf{0}$ term in processes, writing $a.\mathbf{0}$ as a. We also denote the located inactive process $l[\![\mathbf{0}]\!]$ as simply $\mathbf{0}$ and omit location information for processes located at the immortal location. Thus, at system level, we write $M\,|\,P$ to denote $M\,|\ \star\,[\![P]\!]$.

Operational Semantics: We define a *liveset*, $\mathcal{L}$, a set of locations, $\{l_1,\ldots,l_n\}$ denoting the locations that are alive - we omit the special location $\star$ from $\mathcal{L}$. A system M subject to a liveset, $\mathcal{L}$, and a bounded number of dynamic fails, n, is called a configuration, and is denoted as $\langle\mathcal{L},n\rangle \triangleright M$. Intuitively it denotes a system M that is running on the network (state) $\mathcal{L}$ where at most n locations from $\mathcal{L}$ may fail. Transitions are defined between tuples of configurations as

$$\langle\mathcal{L},n\rangle \triangleright M \quad \xrightarrow{\alpha} \quad \langle\mathcal{L}',n'\rangle \triangleright M' \tag{1}$$

by the rules in Table 3. To improve readability, we abbreviate (1) to $\langle\mathcal{L},n\rangle\triangleright M \ \xrightarrow{\alpha}\ M'$ whenever the state of the network $\langle\mathcal{L},n\rangle$ does not change in the residual configuration. The rules in Table 3 are standard located CCS rules, with the exception of (Fail) describing the reduction of the new fail $l.P$ construct, and (Halt) describing dynamic failure.

Example 1. In (2) below, the system $\star[\![a.P + \mathsf{fail}\ l.P]\!]$ is in some sense *fault tolerant* up to 1 failure occuring in $\mathcal{L}$. Even though $a.P$ depends on l to proceed as P, fail $l.P$ produces the same continuation P when the former is blocked (because the co-action $l[\![\overline{a}]\!]$ is dead). We have three cases to consider to verify this: (a) if $l \notin \mathcal{L}$ then fail $l.P$ will trigger and produce $\star[\![P]\!]$; (b) if $l \in \mathcal{L}$ and $n = 0$, then l can never die and $a.P$ will

Table 3. *Reduction Rules*

Assuming $l \in \mathcal{L}$, $n \geq 0$

(Act)
$$\frac{}{\langle\mathcal{L},n\rangle \triangleright l[\![\alpha.P]\!] \xrightarrow{\alpha} l[\![P]\!]}$$

(Fail)
$$\frac{}{\langle\mathcal{L},n\rangle \triangleright l[\![\mathsf{fail}\;k.P]\!] \xrightarrow{\tau} l[\![P]\!]}\; k \notin \mathcal{L}$$

(Halt)
$$\frac{}{\langle\mathcal{L},n{+}1\rangle \triangleright M \xrightarrow{\tau} \langle\mathcal{L}{-}l,n\rangle \triangleright M}$$

(Fork)
$$\frac{}{\langle\mathcal{L},n\rangle \triangleright l[\![P|Q]\!] \xrightarrow{\tau} l[\![P]\!]|l[\![Q]\!]}$$

(New)
$$\frac{}{\langle\mathcal{L},n\rangle \triangleright l[\![(\nu a)P]\!] \xrightarrow{\tau} (\nu a)l[\![P]\!]}$$

(Sum)
$$\frac{\langle\mathcal{L},n\rangle \triangleright l[\![P_i]\!] \xrightarrow{\alpha} l[\![P]\!]}{\langle\mathcal{L},n\rangle \triangleright l[\![\sum_{i\in I} P_i]\!] \xrightarrow{\alpha} l[\![P]\!]}$$

(Rest)
$$\frac{\langle\mathcal{L},n\rangle \triangleright M \xrightarrow{\alpha} \langle\mathcal{L}',n'\rangle \triangleright M'}{\langle\mathcal{L},n\rangle \triangleright (\nu a)M \xrightarrow{\alpha} \langle\mathcal{L}',n'\rangle \triangleright (\nu a)M'}\; \alpha \notin \{a,\bar{a}\}$$

(Par)
$$\frac{\langle\mathcal{L},n\rangle \triangleright M \xrightarrow{\alpha} \langle\mathcal{L}',n'\rangle \triangleright M'}{\begin{array}{l}\langle\mathcal{L},n\rangle \triangleright M|N \xrightarrow{\alpha} \langle\mathcal{L}',n'\rangle \triangleright M'|N \\ \langle\mathcal{L},n\rangle \triangleright N|M \xrightarrow{\alpha} \langle\mathcal{L}',n'\rangle \triangleright N|M'\end{array}}$$

(Com)
$$\frac{\langle\mathcal{L},n\rangle \triangleright M \xrightarrow{\alpha} M' \qquad \langle\mathcal{L},n\rangle \triangleright N \xrightarrow{\bar{\alpha}} N'}{\begin{array}{l}\langle\mathcal{L},n\rangle \triangleright M|N \xrightarrow{\tau} M'|N' \\ \langle\mathcal{L},n\rangle \triangleright N|M \xrightarrow{\tau} N'|M'\end{array}}$$

always synchronise with $l[\![\bar{a}]\!]$ and continue as $\star[\![P]\!]$; (c) if $l \in \mathcal{L}$ and $n \neq 0$ then if l dies before the synchronisation on a occurs, we have case (a), otherwise we have case (b).

$$\langle\mathcal{L},n\rangle \triangleright (\nu a)\;\; l[\![\bar{a}]\!] \mid \star\, [\![a.P + \mathsf{fail}\; l.P]\!] \tag{2}$$

The equivalence relation chosen for our partial-failure calculus is *(weak) bisimulation equivalence*, based on weak matching moves $\overset{\hat{\alpha}}{\Longrightarrow}$ denoting $\xrightarrow{\tau}^{*}\xrightarrow{\alpha}\xrightarrow{\tau}^{*}$ if $\alpha \in \{a,\bar{a}\}$ and $\xrightarrow{\tau}^{*}$ if $\alpha = \tau$.

Definition 1 (Weak bisimulation equivalence). *Denoted as $\approx$, is the largest relation over configurations such that if $\langle\mathcal{L}_1,n_1\rangle \triangleright M_1 \approx \langle\mathcal{L}_2,n_2\rangle \triangleright M_2$ then*

– $\langle\mathcal{L}_1,n_1\rangle \triangleright M_1 \xrightarrow{\alpha} \langle\mathcal{L}_1',n_1'\rangle \triangleright M_1'$ *implies* $\langle\mathcal{L}_2,n_2\rangle \triangleright M_2 \overset{\hat{\alpha}}{\Longrightarrow} \langle\mathcal{L}_2',n_2'\rangle \triangleright M_2'$ *such that* $\langle\mathcal{L}_1',n_1'\rangle \triangleright M_1' \approx \langle\mathcal{L}_2',n_2'\rangle \triangleright M_2'$

– $\langle\mathcal{L}_2,n_2\rangle \triangleright M_2 \xrightarrow{\alpha} \langle\mathcal{L}_2',n_2'\rangle \triangleright M_2'$ *implies* $\langle\mathcal{L}_1,n_1\rangle \triangleright M_1 \overset{\hat{\alpha}}{\Longrightarrow} \langle\mathcal{L}_1',n_1'\rangle \triangleright M_1'$ *such that* $\langle\mathcal{L}_1',n_1'\rangle \triangleright M_1' \approx \langle\mathcal{L}_2',n_2'\rangle \triangleright N_2'$

Assuming that $\mathbf{loc}(M)$ is a function returning the set of all location names used in M, then system M is said to be executing in a *failure-free* setting if it is subject to the network $\langle\mathbf{loc}(M),0\rangle$. Based on this intuition and our notion of equivalence, we can give a formal definition for fault-tolerant systems.

Definition 2 (Fault Tolerance). *A system M is fault tolerant up to n faults whenever*

$$\langle\mathbf{loc}(M),0\rangle \triangleright M \;\;\approx\;\; \langle\mathbf{loc}(M),n\rangle \triangleright M$$

Table 4. *Two-Phase Commit Algorithm in our Partial-Failure Calculus*

$$P \overset{\text{def}}{=} \mathit{prop}^{\mathit{true}}.(\overline{\mathit{vote}^{\mathit{true}}} \mid \mathit{dec}^{\mathit{true}}.\overline{\mathit{commit}^{\mathit{true}}} + \mathit{dec}^{\mathit{false}}.\overline{\mathit{commit}^{\mathit{false}}}) \;+\; \mathit{prop}^{\mathit{false}}.\overline{\mathit{vote}^{\mathit{false}}}.\overline{\mathit{commit}^{\mathit{false}}}$$

$$C \overset{\text{def}}{=} (\underbrace{\mathit{vote}^{\mathit{true}} \ldots \mathit{vote}^{\mathit{true}}}_{n}.\prod_{i=1}^{n} \overline{\mathit{dec}^{\mathit{true}}} \;\mid\; \mathit{vote}^{\mathit{false}}.\prod_{i=1}^{n} \overline{\mathit{dec}^{\mathit{false}}}) \;\mid\; \prod_{i=1}^{n} \mathsf{fail}\ l_i.\overline{\mathit{vote}^{\mathit{false}}}$$

$$\mathit{2PC} \overset{\text{def}}{=} (\nu \mathit{vote}^{\mathit{true}}, \mathit{vote}^{\mathit{false}}, \mathit{dec}^{\mathit{true}}, \mathit{vote}^{\mathit{false}})\;\; l_0[\![C]\!] \mid \prod_{i=1}^{n} l_i[\![P]\!]$$

Our chosen definitions are not arbitrary. Definition 1 is sound with respect to a standard notion of contextual equivalence called reduction barbed congruence [10]. Definition 2 is sound with respect to a notion of dynamic fault-tolerance up-to n faults defined in [7], using fault inducing contexts. The adaptation of these concepts to our calculus and the proof of the corresponding soundness statements will appear in the full version of the paper.

Example 2. Using Definitions 1 and 2, we can now show that (2) is fault tolerant up to 1 fault by giving a witness bisimulation relation satisfying

$$\langle\{l\}, 0\rangle \triangleright (\nu a)\; l[\![\overline{a}]\!] \mid \star\; [\![a.P + \mathsf{fail}\ l.P]\!] \quad\approx\quad \langle\{l\}, 1\rangle \triangleright (\nu a)\; l[\![\overline{a}]\!] \mid \star\; [\![a.P + \mathsf{fail}\ l.P]\!]$$

3 Fault-Tolerant Distributed Algorithms

Despite its limitations (no infinite computation), our calculus is expressive enough to describe a number of (non-recursive) standard distributed algorithms in the presence of dynamic failure. The system *2PC*, defined in Table 4, describes the two-phase commit algorithm solving atomic commit with weak termination [11]. It consists of n participants executing P, located at independently failing locations l_i, and a single coordinator C, located at another failing location l_0. Participants are initialised to either *true* or *false* and then vote this value to the coordinator; if they have a *false* they immediately commit on *false*; otherwise they await for the decided value from the coordinator before commiting. The coordinator collects the votes: if it has n *true* votes it broadcasts $\overline{\mathit{dec}^{\mathit{true}}}$; if it has a single *false* vote or a missing vote because the participant died, it decides *false*.

The correctness condition for the two phase commit states that every participant that commits must commit on the same value. Moreover, if there is a single *false* value proposed then *false* is the only value that can be commited. The weak termination condition states that if there is failure (to a participant or the coordinator) then some participants may never commit. The specification in Table 5 attempts to describe this behaviour directly, using the approach of (Table 1(a)). It consists of two phases, the voting phase $\mathit{Spec}(i, s)$ and the decision phase, $\mathit{DecT}(j, s)$ and $\mathit{DecF}(i, j, s)$; i denotes the number of participants that still need to be proposed, j denotes the number of participants that can still commit and s is a set of numbers denoting the participants (l_i) that are still alive. The specification makes sure that participants cannot commit before being proposed and that we immediately switch to *DecF* as soon as one participant is initialise to *false*. We thus expect the following to hold for $\mathcal{L}_n^+ = \{l_0, l_1, \ldots, l_n\}$:

$$\langle \mathcal{L}_n^+, n+1\rangle \triangleright \mathit{2PC} \;\approx\; \langle \mathcal{L}_n^+, n+1\rangle \triangleright \mathit{Spc}(n, \{1, \ldots, n\})$$

Table 5. *Correctness Specification for the Two-Phase Commit Algorithm of Table 4*

$$
\begin{array}{rcll}
Spc(i,s) & \overset{\text{def}}{=} & \left(\begin{array}{l} prop^{true}.Spc(i-1,s) \;+\; prop^{false}.\,DecF(i-1,\,|s|-(i-1),\,s) \\ +\sum_{k\in s}\mathsf{fail}\ l_k.\left(\begin{array}{l}\tau.DecF(i-1,\,|s/\{k\}|-i,\,s/\{k\}) \\ +\,\tau.DecF(i,\,|s/\{k\}|-i,\,s/\{k\})\end{array}\right) \;+\; \mathsf{fail}\ l_0.\mathbf{0}\end{array}\right) & i>0 \\
Spc(0,s) & \overset{\text{def}}{=} & DecT(n-|s|,s) \qquad\qquad DecT(0,s) \overset{\text{def}}{=} \mathbf{0} & \\
DecT(j,s) & \overset{\text{def}}{=} & \left(\begin{array}{l} commit^{true}.DecT(j-1,s) \\ +\sum_{k\in s}\mathsf{fail}\ l_k.DecT(j-1,s/\{k\}) \;+\; \mathsf{fail}\ l_0.\mathbf{0}\end{array}\right) & j>0 \\
DecF(i,j,s) & \overset{\text{def}}{=} & \left(\begin{array}{l} Prop(i,j,s) \;+\; Comm(i,j,s) \;+\; \mathsf{fail}\ l_0.\mathbf{0} \\ +\;\sum_{k\in s}\mathsf{fail}\ l_k.\left(\begin{array}{l}\tau.DecF(i-1,\,j-1,\,s/\{k\}) \\ +\,\tau.DecF(i,\,j-1,\,s/\{k\})\end{array}\right)\end{array}\right) & \\
Prop(i,j,s) & \overset{\text{def}}{=} & \begin{cases} prop^{true}.DecF(i+1,j+1,s) \;+\; prop^{false}.DecF(i+1,j+1,s) & i<n \\ \mathbf{0} & i\geq n\end{cases} & \\
Comm(i,j,s) & \overset{\text{def}}{=} & \begin{cases} commit^{false}.DecF(i,j-1,s) & j>0 \\ \mathbf{0} & j\leq 0\end{cases} &
\end{array}
$$

Table 6. *The Rotating Coordinator Algorithm for Participant i*

```
1        $x_i$ := input;
2        for $r$ := 1 to $n$ do {  if $r = i$ then broadcast $x_i$;
3                                  if alive($p_r$) then $x_i$ := input_from_broadcast };
4        output $x_i$;
```

As stated in the Introduction, apart from complexity arising from globally testing for all correctness conditions at one go, the atomic commit correctness specification of Table 5 is further complicated by the failure conditions that need to be catered for. At each stage, if the coordinator fails (l_0), then there is the possibility that participants stop commiting. Also, when a participant fails, it either means that we have one less commit or one less propose and commit. These complications lessen our confidence in the correctness of the specification and complicate subsequent proofs.

To illustrate how our methodology works, we use the calculus to describe another distributed algorithm, the rotating co-ordinator algorithm [16] of Table 6, solving a specific instance of consensus using *strong* failure detectors ($\mathcal{S}$). The algorithm consists of n parallel, independently failing processes, ordered and named 1 to n, each *inputting* a value v from a set of values V and then *deciding* by outputting a value $v' \in V$. Each process executes the code in Table 6: It performs n rounds (the loop on lines 2 and 3), changing the broadcasting coordinator to process i for round $r = i$. The correctness criteria for *consensus* is often defined by the following three conditions:

Termination: All non-failing processes must eventually decide
Agreement: No two processes decide on different values
Validity: If all processes are given the same value $v \in V$ as input, then v is the only possible decision value.

To attain consensus with $n-1$ dynamic failures, the algorithm needs to be fault-tolerant with respect to two error conditions, namely *Decision Blocking* (when a participant may be waiting forever for a value to be broadcast by a dead coordinator) and *Corrupted Broadcast* (when coordinator may broadcast its values to a *subset* of the participants before failing). The code in Table 6 overcomes decision blocking by using a failure

Table 7. *Rotating Co-ordinator Algorithm in our Partial-Failure Calculus*

(Consensus)

$$\mathsf{C} \overset{\text{def}}{=} \left(\nu^{n}_{i,r=1} \mathit{true}_{i,r}, \mathit{false}_{i,r}\right) \prod_{i=1}^{n} l_i[\![\mathit{prop}_i^{\mathit{true}}.\mathsf{P}^{\mathit{true}}_{i,1} + \mathit{prop}_i^{\mathit{false}}.\mathsf{P}^{\mathit{false}}_{i,1}]\!]$$

(Participant)

$$\mathsf{P}^{x}_{i,r} \overset{\text{def}}{=} \mathsf{R}^{x}_{i,r} \mid \mathsf{B}^{x}_{i,r} \quad x \in \{\mathit{true}, \mathit{false}\},\ r < n$$

$$\mathsf{P}^{x}_{i,n} \overset{\text{def}}{=} \overline{\mathit{dec}^{x}_{i}} \quad x \in \{\mathit{true}, \mathit{false}\}$$

(Broadcast)

$$\mathsf{B}^{x}_{i,r} \overset{\text{def}}{=} \prod_{j=1}^{n} \overline{x_{j,r}} \quad x \in \{\mathit{true}, \mathit{false}\},\ r = i$$

$$\mathsf{B}^{x}_{i,r} \overset{\text{def}}{=} \mathbf{0} \quad x \in \{\mathit{true}, \mathit{false}\},\ r \neq i$$

(Recieve)

$$\mathsf{R}^{x}_{i,r} \overset{\text{def}}{=} \mathit{true}_{i,r}.\mathsf{P}^{\mathit{true}}_{i,r+1} + \mathit{false}_{i,r}.\mathsf{P}^{\mathit{false}}_{i,r+1} + \mathsf{fail}\ l_r.\mathsf{P}^{x}_{i,r+1}$$

detector to determine the state of the coordinator (`alive(`p_r`)`) and overcomes the possibility of $(n-1)$ corrupted broadcasts by repeating the broadcast for n rounds.

We give a precise description of the rotating co-ordinator algorithm as the system C, given in Table 7. Without loss of generality, we assume that the decision set is simply $V = \{true, false\}$ and have n processes located at *independently failing* locations $l_1 \ldots l_n$. The process $\mathsf{P}^{x}_{i,r}$, for $x \in \{true, false\}$, denotes the i^{th} participant, at round r, with current estimate x. It is defined in terms of two parallel processes, $\mathsf{B}^{x}_{i,r}$ for *broadcasting* the current value at round r, and $\mathsf{R}^{x}_{i,r}$ for *receiving* the new value at round r. As in Table 6, broadcast is only allowed if $i = r$ and otherwise it acts as the inert process. On the other hand, the receiver at round r awaits synchronisation on $\mathit{true}_{i,r}$ or $\mathit{false}_{i,r}$ and updates the estimate for round $(r+1)$ accordingly. At the same time, the receiver guards this distributed synchronisation with $\mathsf{fail}\ l_r.\mathsf{P}^{x}_{i,r+1}$ to prevent decision blocking in case l_r, the location of the participant currently in charge of the broadcast, fails. Estimates for round r can only come from the participant at l_r and thus all actions $\mathit{true}_{i,r}$ and $\mathit{false}_{i,r}$ are scoped in C. Every participant can be arbitrarily initialised as $\mathsf{P}^{\mathit{true}}_{i,1}$ or $\mathsf{P}^{\mathit{false}}_{i,1}$ through the free actions $\mathit{prop}^{\mathit{true}}_{i}$ and $\mathit{prop}^{\mathit{false}}_{i}$ respectively. Finally every participant decides at round $(n+1)$ to either report true, executing $\mathit{dec}^{\mathit{true}}_{i}$, or report false, executing $\mathit{dec}^{\mathit{false}}_{i}$.

We can also give a precise description of the consensus correctness requirements in our calculus. As stated in the Introduction, instead of expressing these requirements in terms of a specification to be compared to, we repackage our algorithm as a fault-tolerant system where any interactions with observers occur with code residing at the immortal location $\star$. This allows us to decompose our proof into the failure-free phase and the $n-1$ failure phase.

Table 8 defines the *wrapper code* which, when put in parallel with C of Table 7, provides separate *testing scenarios* for the algorithm. We have two forms of initialization code: I^{gen} initialises every participant to either *true* or *false* arbitrarily after the action *start* whereas $\mathsf{I}^{\mathit{true}}$ and $\mathsf{I}^{\mathit{false}}$ initialise *all* participants to just *true*, or just *false* respectively, after *start*. Similarly, we have two forms of code that evaluates the values decided upon: A^{gen}_{1} checks that all the participants $1, \ldots, n$ agreed upon a value (either *true* or *false*) or else died, performing the action $\overline{ok}$ if the test is successful; $\mathsf{A}^{\mathit{true}}_{1}$ and $\mathsf{A}^{\mathit{false}}_{1}$ check whether all participants have agreed upon the particular value *true*, and *false* respectively, or died, outputting $\overline{ok}$ if the test is successful.

Table 8. *Consensus Wrappers*

(Initialisation)		
$\mathsf{I}^{x} \overset{\text{def}}{=} \mathit{start}.\prod_{i=1}^{n} \overline{\mathit{prop}_i^{x}}$	$\mathsf{I}^{gen} \overset{\text{def}}{=} \mathit{start}.\prod_{i=1}^{n} \overline{\mathit{prop}_i^{true}} + \overline{\mathit{prop}_i^{false}}$	$x \in \{\mathit{true},\ \mathit{false}\}$
(Agreement)		
$\mathsf{A}_i^{x} \overset{\text{def}}{=} \mathit{dec}_i^{x}.\mathsf{A}_{i+1}^{x} + \mathsf{fail}\ l_i.\mathsf{A}_{i+1}^{x}$	$\mathsf{A}_{n+1}^{x} \overset{\text{def}}{=} \overline{\mathit{ok}}$	$x \in \{\mathit{true},\ \mathit{false}\},\ i \le n$
$\mathsf{A}_i^{gen} \overset{\text{def}}{=} \mathit{dec}_i^{true}.\mathsf{A}_{i+1}^{true} + \mathit{dec}_i^{false}.\mathsf{A}_{i+1}^{false} + \mathsf{fail}\ l_i.\mathsf{A}_{i+1}$		$i \le n$

Table 9. *Atomic Commit Agreement Wrappers*

$\mathsf{I}^{gen} \overset{\text{def}}{=} \mathit{start}.\prod_{i=1}^{n} \overline{\mathit{prop}_i^{true}} + \overline{\mathit{prop}_i^{false}}$	$\mathsf{I}^{1false} \overset{\text{def}}{=} \mathit{start}.(\overline{\mathit{prop}_i^{false}} + \prod_{i=1}^{n-1} \overline{\mathit{prop}_i^{true}} + \overline{\mathit{prop}_i^{false}})$	
$\mathsf{T}_i^{S,x} \overset{\text{def}}{=} \mathit{commit}_i^{x}.\mathsf{A}_{i-1}^{S,x} + \mathsf{fail}\ l_0.\overline{\mathit{ok}} + \prod_{k\in S} \mathsf{fail}\ k.\mathsf{A}_{i-1}^{S/\{k\},x}$		$x \in \{\mathit{true},\ \mathit{false}\},\ i \le n$
$\mathsf{T}_i^{S,gen} \overset{\text{def}}{=} \mathit{commit}_i^{true}.\mathsf{A}_{i-1}^{S,true} + \mathit{commit}_i^{false}.\mathsf{A}_{i-1}^{S,false} + \mathsf{fail}\ l_0.\overline{\mathit{ok}} + \prod_{k\in S} \mathsf{fail}\ k.\mathsf{A}_{i-1}^{S/\{k\},gen}$		$i \le n$
$\mathsf{T}_0^{S,x} \overset{\text{def}}{=} \overline{\mathit{ok}}$		$x \in \{\mathit{true},\ \mathit{false},\ \mathit{gen}\},\ i \le n$

Definition 3 (Consensus). *Let $\mathcal{L}_n$ denote the liveset $\{l_1 \dots l_n\}$, and $(\tilde{n})$ stand for the sequence of actions prop_i^{true}, prop_i^{false}, dec_i^{true}, dec_i^{false}, $1 \le i \le n$. Then system C satisfies consensus whenever*

Strong ff-Agreement: $\langle \mathcal{L}_n, 0\rangle \triangleright (\nu\tilde{n})C \,|\, I^{gen} \,|\, A_1^{gen} \;\approx\; \langle \emptyset, 0\rangle \triangleright \mathit{start}.\overline{\mathit{ok}}$

ff-Validity:
$$\langle \mathcal{L}_n, 0\rangle \triangleright (\nu\tilde{n})C \,|\, I^{true} \,|\, A_1^{true} \;\approx\; \langle \emptyset, 0\rangle \triangleright \mathit{start}.\overline{\mathit{ok}}$$
$$\langle \mathcal{L}_n, 0\rangle \triangleright (\nu\tilde{n})C \,|\, I^{false} \,|\, A_1^{false} \;\approx\; \langle \emptyset, 0\rangle \triangleright \mathit{start}.\overline{\mathit{ok}}$$

and moreover

Strong ft-Agreement: $\langle \mathcal{L}_n, 0\rangle \triangleright (\nu\tilde{n})C \,|\, I^{gen} \,|\, A_1^{gen} \;\approx\; \langle \mathcal{L}_n, (n-1)\rangle \triangleright (\nu\tilde{n})C \,|\, I^{gen} \,|\, A_1^{gen}$

ft-Validity:
$$\langle \mathcal{L}_n, 0\rangle \triangleright (\nu\tilde{n})C \,|\, I^{true} \,|\, A_1^{true} \;\approx\; \langle \mathcal{L}_n, (n-1)\rangle \triangleright (\nu\tilde{n})C \,|\, I^{true} \,|\, A_1^{true}$$
$$\langle \mathcal{L}_n, 0\rangle \triangleright (\nu\tilde{n})C \,|\, I^{false} \,|\, A_1^{false} \;\approx\; \langle \mathcal{L}_n, (n-1)\rangle \triangleright (\nu\tilde{n})C \,|\, I^{false} \,|\, A_1^{false}$$

In the above definition, *strong agreement* subsumes the agreement and termination conditions; it composes C with I^{gen} and A_1^{gen}. *Validity* uses more specific wrappers, and composes C first with $\mathsf{I}^{true} \,|\, \mathsf{A}_1^{true}$ and then with $\mathsf{I}^{false} \,|\, \mathsf{A}_1^{false}$. Scoping the actions prop_i^{true}, prop_i^{false}, dec_i^{true}, dec_i^{false} in each of these three cases limits external interaction to the non-failing actions *start* and $\overline{\mathit{ok}}$ at the immortal location, in the style of Table 1(c) and (d). Stated otherwise, Definition 3 reduces more complex formulations of *consensus*, as in Table 1(b), to a combination of a failure-free statement and a fault-tolerance statement. For example

Strong Agreement: $\langle \mathcal{L}_n, (n-1)\rangle \triangleright (\nu\tilde{n})C \,|\, \mathsf{I}^{gen} \,|\, \mathsf{A}_1^{gen} \;\approx\; \langle \emptyset, 0\rangle \triangleright \mathit{start}.\overline{\mathit{ok}}$

follows from **Strong ff-Agreement**, **Strong ft-Agreement** and transitivity of $\approx$.

The remainder of the paper is dedicated to the proofs of these correctness criteria for consensus. We however note that our methodology should also be applicable to prove the correctness of *2PC* of Table 4. More concretely, this should involve proving the following two equivalences,

$$\langle \mathcal{L}_n^{+}, n+1\rangle \triangleright (\nu\tilde{n})2PC \,|\, \mathsf{I}^{gen} \,|\, \mathsf{T}_n^{S^n,gen} \;\approx\; \langle \emptyset, 0\rangle \triangleright \mathit{start}.\overline{\mathit{ok}}$$

$$\langle \mathcal{L}_n^{+}, n+1\rangle \triangleright (\nu\tilde{n})2PC \,|\, \mathsf{I}^{1false} \,|\, \mathsf{T}_n^{S^n,false} \;\approx\; \langle \emptyset, 0\rangle \triangleright \mathit{start}.\overline{\mathit{ok}}$$

Table 10. *Structural Equivalence Rules*

(s-comm)	$\langle \mathcal{L}, n\rangle \triangleright N \mid M \equiv \langle \mathcal{L}, n\rangle \triangleright M \mid N$	
(s-assoc)	$\langle \mathcal{L}, n\rangle \triangleright (N \mid M) \mid M' \equiv \langle \mathcal{L}, n\rangle \triangleright N \mid (M \mid M')$	
(gc-Inert)	$\langle \mathcal{L}, n\rangle \triangleright M \mid \mathbf{0} \equiv \langle \mathcal{L}, n\rangle \triangleright M$	
(s-Extr)	$\langle \mathcal{L}, n\rangle \triangleright (\nu a)(M \mid N) \equiv \langle \mathcal{L}, n\rangle \triangleright M \mid (\nu a)N$	$a \notin \mathbf{fn}(M)$
(gc-Scope)	$\langle \mathcal{L}, n\rangle \triangleright (\nu a)M \equiv \langle \mathcal{L}, n\rangle \triangleright M$	$a \notin \mathbf{fn}(M)$
(gc-Act)	$\langle \mathcal{L}, n\rangle \triangleright (\nu a)l[\![\alpha.P + \sum_i P_i]\!] \equiv \langle \mathcal{L}, n\rangle \triangleright (\nu a)l[\![\sum_i P_i]\!]$	$\alpha \in \{a, \overline{a}\}$
(gc-Fail)	$\langle \mathcal{L}, 0\rangle \triangleright l[\![\text{fail } k.P + \sum_i P_i]\!] \equiv \langle \mathcal{L}, 0\rangle \triangleright l[\![\sum_i P_i]\!]$	$k \in \mathcal{L}$
(s-Dead)	$\langle \mathcal{L}, n\rangle \triangleright l[\![P]\!] \equiv \langle \mathcal{L}, n\rangle \triangleright l[\![Q]\!]$	$l \notin \mathcal{L}$

(s-Rest)
$$\frac{\langle \mathcal{L}, n\rangle \triangleright M \equiv \langle \mathcal{L}, n\rangle \triangleright N}{\langle \mathcal{L}, n\rangle \triangleright (\nu a)M \equiv \langle \mathcal{L}, n\rangle \triangleright (\nu a)N}$$

(Par)
$$\frac{\langle \mathcal{L}, n\rangle \triangleright M \equiv \langle \mathcal{L}, n\rangle \triangleright M'}{\begin{array}{c}\langle \mathcal{L}, n\rangle \triangleright M|N \equiv \langle \mathcal{L}, n\rangle \triangleright M'|N \\ \langle \mathcal{L}, n\rangle \triangleright N|M \equiv \langle \mathcal{L}, n\rangle \triangleright N|M'\end{array}}$$

where $S^n = \{1..n\}$ and $\tilde{n} = prop_i^{true}, prop_i^{false}, commit_i^{true}, commit_i^{false}, \ 1 \leq i \leq n$ using the tests defined in Table 9. As above, these equivalences can be further split into the failure-free and fault-tolerant phases.

4 Up-to Techniques in the Presence of Failure

The 6 bisimulations proving the correctness for the rotating co-ordinator algorithm have limited external interaction; rather, the complication of proving these bisimulations lies in the large amount of internal actions that we need to consider. As we discussed in the Introduction, a large number of these internal actions are regular in structure (processes executing symmetric transitions at different locations and at different rounds). It turns out that a large number of these transitions are *confluent* transitions, meaning that they do not affect the set of transitions we can undertake in our bisimulations, now or in the future. Moreover, in the fault-tolerance bisimulations, we end up with an extensive number of *dead code*, that is code residing at dead locations or code that is forever blocked because it can only be released by actions residing at dead locations. We here develop up-to bisimulation techniques that abstract over confluent moves and dead code. This alleviates the burden of exhibiting our witness bisimulations and allows us to focus on the transitions that really matter.

We start by defining a structural equivalence relation over configurations as the least relation satisfying the rules in Table 10. Even though this equivalence is usually defined over systems, we here use the state of the network $\langle \mathcal{L}, n\rangle$ to define a stronger relation. Apart from the first five rules and the last two (contextual) rules, all of which are fairly standard, we also have new rules such as (s-Dead), adopted from [7], equating any code at dead locations, irrespective of its form. The network information is also used to define the new structural rule (gc-Fail), identifying fail branches that can never trigger because the location tested for can never fail (it is alive and no more failures can be induced). Also new is (gc-Act) which identifies action branches that can never trigger because they

Table 11. *Transition Rules for β-moves*

(BLin)
$$\frac{}{\langle \mathcal{L}, 0\rangle \triangleright (\nu a)(l[\![\bar{a}.P]\!] \mid k[\![a.Q]\!]) \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, 0\rangle \triangleright (\nu a)(l[\![P]\!] \mid k[\![Q]\!])}\; l,k \in \mathcal{L}$$

(BLoc)
$$\frac{}{\langle \mathcal{L}, n\rangle \triangleright (\nu a)(l[\![\bar{a}.P]\!] \mid l[\![a.Q]\!]) \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright (\nu a)(l[\![P]\!] \mid l[\![Q]\!])}$$

(BFTol)
$$\frac{}{\langle \mathcal{L}, n\rangle \triangleright (\nu a)(l[\![\mathsf{fail}\ k.P + a.P]\!] \mid k[\![\bar{a}]\!]) \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright (\nu a)l[\![P]\!]}\; l,k \in \mathcal{L}$$

(BNew)
$$\frac{}{\langle \mathcal{L}, n\rangle \triangleright l[\![(\nu a)P]\!] \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright (\nu a)l[\![P]\!]}\; l \in \mathcal{L}$$

(BRest)
$$\frac{\langle \mathcal{L}, n\rangle \triangleright M \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright M'}{\langle \mathcal{L}, n\rangle \triangleright (\nu a)M \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright (\nu a)M'}$$

(BFork)
$$\frac{}{\langle \mathcal{L}, n\rangle \triangleright l[\![P|Q]\!] \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright l[\![P]\!]|l[\![Q]\!]}\; l \in \mathcal{L}$$

(BPar)
$$\frac{\langle \mathcal{L}, n\rangle \triangleright M \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright M'}{\begin{array}{c}\langle \mathcal{L}, n\rangle \triangleright M|N \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright M'|N \\ \langle \mathcal{L}, n\rangle \triangleright N|M \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}, n\rangle \triangleright N|M'\end{array}}$$

are scoped and there is no corresponding co-action within that scope.[3] We next state a common sanity check ensuring that our structural equivalence is a strong bisimulation.

Lemma 1 (≡ is a strong bisimulation).

$$\begin{tikzcd} \langle \mathcal{L}, n\rangle \triangleright N \arrow[d, "\alpha"'] \arrow[r, phantom, "\equiv"] & \langle \mathcal{L}, n\rangle \triangleright M \arrow[r, phantom, "\textit{implies}"] & \langle \mathcal{L}, n\rangle \triangleright N \arrow[d, "\alpha"'] \arrow[r, phantom, "\equiv"] & \langle \mathcal{L}, n\rangle \triangleright M \arrow[d, "\alpha"'] \\ \langle \mathcal{L}', m'\rangle \triangleright N' & & \langle \mathcal{L}', m'\rangle \triangleright N' \arrow[r, phantom, "\equiv"] & \langle \mathcal{L}', m'\rangle \triangleright M' \end{tikzcd}$$

We identify a number of τ-actions, referred to as β-actions or β-moves, and show that they are confluent. These silent β-actions are denoted as

$$\langle \mathcal{L}, n\rangle \triangleright N \overset{\tau}{\longmapsto}_\beta \langle \mathcal{L}', m\rangle \triangleright M \qquad (3)$$

and defined in Table 11. We then develop up-to bisimulation techniques that abstract from matching configurations that denote β-moves. The details differ considerably from [7] because we use different constructs like choice and *fail*, and allow distributed synchronisation across locations. Thus, apart from the local rules ((BNew) and (BFork)) and the context rules ((BRest) and (BPar)), Table 11 includes three new rules dealing with synchronisations. (BLin) states that distribution does not interfere with a scoped linear synchronisation, as long as *we cannot induce more dynamic failures*, that is $n = 0$. (BLoc) states that a *local* scoped linear synchronisation is always a β-move. Finally,

[3] We purposefully use the naming convention (gc-) for certain structural rules that are generally applied in one direction rather than the other to "garbage collect" redundant dead-code.

(BFTol) states that a *distributed* scoped linear synchronisation is a β-move if it is *asynchronous from one end* and the co-synchronisation at the other end is *guarded by a* fail *with the same continuation*. In other words, these conditions make (BFTol), in a sense, *fault-tolerant* as we have already seen in (2). We prove a special form of confluence for our β-moves. The non-standard use of $\mathcal{R}$ to close the diamond instead of $\xmapsto{\tau}_{\beta}$ allows for the special case when the code causing the β-move becomes dead. In this case we only require that resulting pair are structurally equivalent, exploiting (s-Dead).

Lemma 2 (Confluence of β-moves). $\xmapsto{\tau}_{\beta}$ *observes the diamond property:*

$$\begin{tikzcd}
\langle\mathcal{L},n\rangle \triangleright N \arrow[r, mapsto, "\tau", "\beta"'] \arrow[d, "\alpha"'] & \langle\mathcal{L},n\rangle \triangleright M & \textit{implies} & \langle\mathcal{L},n\rangle \triangleright N \arrow[r, mapsto, "\tau", "\beta"'] \arrow[d, "\alpha"'] & \langle\mathcal{L},n\rangle \triangleright M \arrow[d, "\alpha"'] \\
\langle\mathcal{L}',n'\rangle \triangleright N' & & & \langle\mathcal{L}',n'\rangle \triangleright N' \arrow[r, phantom, "\mathcal{R}"] & \langle\mathcal{L}',n'\rangle \triangleright M'
\end{tikzcd}$$

where $\mathcal{R}$ *is* $\xmapsto{\tau}_{\beta}$ *or* $\equiv$, *or else* $\alpha=\tau$ *and* $\langle\mathcal{L},n\rangle\triangleright M = \langle\mathcal{L}',n'\rangle\triangleright N'$

We defined a modified bisimulation relation from Definition 1 where the conditions for the matching residuals are relaxed; instead of demanding that they are again related in $\approx$ we allow approximate matching through $\equiv$ and $\xmapsto{\tau}^{*}_{\beta}$.

Definition 4 (β-transfer property). *A relation $\mathcal{R}$ over configurations satisfies the β-transfer property if*

$$\begin{tikzcd}
\langle\mathcal{L},n\rangle \triangleright N \arrow[r, phantom, "\mathcal{R}"] \arrow[d, "\alpha"'] & \langle\mathcal{L},n\rangle \triangleright M & \textit{implies} & \langle\mathcal{L},n\rangle \triangleright N \arrow[r, phantom, "\mathcal{R}"] \arrow[d, "\alpha"'] & \langle\mathcal{L},n\rangle \triangleright M \arrow[d, "\alpha"'] \\
\langle\mathcal{L}',n'\rangle \triangleright N' & & & \langle\mathcal{L}',n'\rangle \triangleright N' \arrow[r, phantom, "\mathcal{A}_l\circ\mathcal{R}\circ\mathcal{A}_r"] & \langle\mathcal{L}',n'\rangle \triangleright M'
\end{tikzcd}$$

where $\mathcal{A}_l$ *is* $\equiv\circ\xmapsto{\tau}^{*}_{\beta}$ *and* $\mathcal{A}_r$ *is* $\approx$

Definition 5 (Bisimulation up-to-β). *A relation $\mathcal{R}$ over configurations is a bisimulation up-to-β if it and its inverse $\mathcal{R}^{-1}$ satisfy the β-transfer property.*

Before we can use bisimulations up-to-β, we need to show they are sound with respect to Definition 1. This soundness proof uses the results of Lemma 3.

Lemma 3 ($\xmapsto{\tau}^{*}_{\beta}$ implies $\approx$). *If $\langle\mathcal{L},n\rangle\triangleright N \xmapsto{\tau}^{*}_{\beta} \langle\mathcal{L},n\rangle\triangleright M$. then $\langle\mathcal{L},n\rangle\triangleright N \approx \langle\mathcal{L},n\rangle\triangleright M$.*

Theorem 1 (Soundness of bisimulations up-to-β). *If $\langle\mathcal{L},n\rangle\triangleright N\,\mathcal{R}\,\langle\mathcal{L}',m\rangle\triangleright M$ where $\mathcal{R}$ is a bisimulation up-to-β then $\langle\mathcal{L},n\rangle\triangleright N \approx \langle\mathcal{L}',m\rangle\triangleright M$*

Example 3. Consider the case where $l,k\in\mathcal{L}$ and we have to show that

$$\langle\mathcal{L},n\rangle\triangleright(\nu a,b)\ l[\![\overline{a}]\!]\,|\,k[\![a.P+b.Q+\mathsf{fail}\ l.P]\!] \quad\approx\quad \langle\mathcal{L},n\rangle\triangleright(\nu a,b)k[\![P]\!]$$

Using (s-Extr), (gc-Act) and (s-Extr) again we can tighten the scope of νb, garbage collect the branch guarded by b in the left hand configuration and then scope extrude νb again to obtain

$$\langle\mathcal{L},n\rangle\triangleright(\nu a,b)\ l[\![\overline{a}]\!]\,|\,k[\![a.P+b.Q+\mathsf{fail}\,l.P]\!] \quad\equiv\quad \langle\mathcal{L},n\rangle\triangleright(\nu a,b)\ l[\![\overline{a}]\!]\,|\,k[\![a.P+\mathsf{fail}\,l.P]\!] \quad (4)$$

We now have two cases to consider. If $n = 0$ we apply (gc-Fail) to (4) to garbage collect the fail branch and then apply (BLin) to the resultant configuration to obtain

$$\langle \mathcal{L}, n\rangle \triangleright (\nu a, b)\ l[\![\overline{a}]\!] \mid k[\![a.P]\!] \quad \stackrel{\tau}{\longmapsto}_{\beta} \quad \langle \mathcal{L}, n\rangle \triangleright (\nu a, b)\ k[\![P]\!] \tag{5}$$

If $n \neq 0$ we then apply (BFTol) to (4) to get

$$\langle \mathcal{L}, n\rangle \triangleright (\nu a, b)\ l[\![\overline{a}]\!] \mid k[\![a.P + \mathsf{fail}\ l.P]\!] \quad \stackrel{\tau}{\longmapsto}_{\beta} \quad \langle \mathcal{L}, n\rangle \triangleright (\nu a, b)\ k[\![P]\!] \tag{6}$$

For both cases, (5) and (6), we can then go on to show bisimilarity simply using the identity relation.

5 Consensus Satisfaction Proof

Using the soundness results of Section 4, we just need to give witness bisimulations up-to β-moves satisfying the bisimulations set out in Definition 3. In the following witness bisimulations, we use the letters t, f, p and d for the action names *true*, *false*, *prop* and *dec*. Our bisimulation presentation will make use of sets of integers I_i partitioning the set of integers $\{1 \ldots n\}$; the partition predicate is:

$$\mathsf{part}_1^n(I_1, \ldots, I_k) \quad \stackrel{\text{def}}{=} \quad I_1 \cup \ldots \cup I_k = \{1 \ldots n\} \ \text{ and } \ \forall i, j \in \{1..k\}\ I_i \cap I_j = \emptyset$$

We also denote the smallest number in such partition I as I_{min} and the largest number in a partition I_i that is smaller that any element in any other partition I_j as $I^{+}_{i_{min}}$.

We start by proving the failure-free equivalences. We here only give the witness bisimulation for *strong ff-agreement*; the two witness bisimulations required for ff-validity are similar but simpler. We assume $\tilde{n} = \prod_{i,r=1}^{n} t_{i,r}, f_{i,r}, p_i^t, p_i^f, d_i^t, d_i^f$ and use A, I, $\mathcal{L}_n$ and $\emptyset$ as shorthand for A_1^{gen}, I^{gen}, $\langle\{1 \ldots n\}, 0\rangle$ and $\langle \emptyset, 0\rangle$ respectively. We also partition $\{1 \ldots n\}$ into three sets: I denotes the set of participants that are not *initialised*, while J and H denote initialised participants that *agree* on t and on f respectively. We also use the process definition $N_i \stackrel{\text{def}}{=} l_i[\![p_1^t.\mathsf{P}_{i,1}^t + p_i^f.\mathsf{P}_{i,1}^f]\!] \mid \overline{p_t} + \overline{p_f}$ for *non-initialised* participant i.

$$\left\{\begin{array}{lll}
1) & \Big\langle \mathcal{L}_n \triangleright (\nu\tilde{n})\Big(C \mid \mathsf{I}^{gen} \mid \mathsf{A}_1^{gen}\Big),\ \emptyset \triangleright start.\overline{ok}\Big\rangle & \\
2) & \left\langle \mathcal{L}_n \triangleright (\nu\tilde{n})\left(\mathsf{A} \mid \prod_{i\in I} N_i \mid \prod_{j\in J} l_j[\![\mathsf{R}_{j,1}^t]\!] \mid \prod_{h\in H} l_h[\![\mathsf{R}_{h,1}^f]\!]\right),\ \emptyset \triangleright \overline{ok}\right\rangle & \left|\ \begin{array}{l}\mathsf{part}_1^n(I, J, H)\\ \text{and } I_{min} = 1\end{array}\right. \\
3) & \left\langle \mathcal{L}_n \triangleright (\nu\tilde{n})\left(\mathsf{A} \mid \prod_{i\in I}\left(N_i \mid \prod_{r=1}^{I_{min}-1} l_r[\![\overline{x_{i,r}}]\!]\right) \mid \prod_{j\in J} l_j[\![\mathsf{R}_{j,I_{min}}^x]\!]\right),\ \emptyset \triangleright \overline{ok}\right\rangle & \left|\ \begin{array}{l}\mathsf{part}_1^n(I, J)\\ \text{and } I_{min} \neq 1\\ \text{and } x \in \{t, f\}\end{array}\right. \\
4) & \Big\langle \mathcal{L}_n \triangleright \overline{ok},\ \emptyset \triangleright \overline{ok}\Big\rangle &
\end{array}\right\}$$

In the above witness bisimulation up-to β-moves case (2) represents the states where participants do not agree at round $r = 1$ because the broadcaster at round 1 has not been initialised yet. Case (3) represents participants in agreement for rounds $r \geq 2$ but blocked because the co-ordinator participant for round r has not been initialised. The main transitions for this bisimulation relation are overviewed in Appendix A.

$$
\left\{
\begin{array}{lll}
1) & \langle \mathcal{L}_n \triangleright (\nu\tilde{n})\mathsf{A} \mid \mathsf{I} \mid \mathsf{C},\ \mathcal{L}_n^K \triangleright (\nu\tilde{n})\mathsf{A} \mid \mathsf{I} \mid \mathsf{C}\rangle & | K \subseteq \{1\ldots n\} \\[2ex]
2) & \begin{array}{l}\Big\langle \mathcal{L}_n \triangleright (\nu\tilde{n})\Big(\mathsf{A} \mid \prod_{i\in I} N_i \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^t_{j,1}\rrbracket \mid \prod_{h\in H} l_h\llbracket \mathsf{R}^f_{h,1}\rrbracket \mid \prod_{k\in K} l_k\llbracket P_k\rrbracket\Big) \\ \quad ,\ \mathcal{L}_n^K \triangleright (\nu\tilde{n})\Big(\mathsf{A} \mid \prod_{i\in I} N_i \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^t_{j,1}\rrbracket \mid \prod_{h\in H} l_h\llbracket \mathsf{R}^f_{h,1}\rrbracket\Big)\Big\rangle\end{array} & \left|\begin{array}{l}\mathsf{part}^n_1(I,J,H,K) \\ \text{and } I_{min} = 1\end{array}\right. \\[4ex]
3) & \begin{array}{l}\Big\langle \mathcal{L}_n \triangleright (\nu\tilde{n})\left(\begin{array}{l}\mathsf{A} \mid \prod_{i\in I}\Big(N_i \mid B(i,x)_1^{I_{min}-1}\Big) \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^x_{j,I_{min}}\rrbracket \\ \mid \prod_{h\in H} l_h\llbracket \mathsf{R}^x_{h,I_{min}}\rrbracket \mid \prod_{k\in K} l_k\llbracket P_k\rrbracket\end{array}\right) \\ \quad ,\ \mathcal{L}_n^K \triangleright (\nu\tilde{n})\Big(\mathsf{A} \mid \prod_{i\in I} N_i \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^y_{j,I_{min}}\rrbracket \mid \prod_{h\in H} l_h\llbracket \mathsf{R}^{\ddot{y}}_{h,I_{min}}\rrbracket\Big)\Big\rangle\end{array} & \left|\begin{array}{l}\mathsf{part}^n_1(I,J,H,K) \text{ and} \\ 1 \neq I_{min} < J_{min}, H_{min} \\ \text{and } x,y \in \{t,f\} \\ \text{and } \{1..I_{min}\} \subseteq K\end{array}\right. \\[4ex]
4) & \begin{array}{l}\Big\langle \mathcal{L}_n \triangleright (\nu\tilde{n})\left(\begin{array}{l}\mathsf{A} \mid \prod_{i\in I}\Big(N_i \mid B(i,x)_1^{I_{min}-1}\Big) \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^x_{j,I_{min}}\rrbracket \\ \mid \prod_{h\in H} l_h\llbracket \mathsf{R}^x_{h,I_{min}}\rrbracket \mid \prod_{k\in K} l_k\llbracket P_k\rrbracket\end{array}\right) \\ \quad ,\ \mathcal{L}_n^K \triangleright (\nu\tilde{n})\left(\begin{array}{l}\mathsf{A} \mid \prod_{i\in I}\Big(N_i \mid B(i,y)_{J_{min}}^{J^+_{min}}\Big) \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^y_{j,I_{min}}\rrbracket \\ \mid \prod_{h\in H}\Big(l_h\llbracket \mathsf{R}^{\ddot{y}}_{h,I_{min}}\rrbracket \mid B(i,y)_{J_{min}}^{J^+_{min}}\Big)\end{array}\right)\Big\rangle\end{array} & \left|\begin{array}{l}\mathsf{part}^n_1(I,J,H,K) \\ \text{and } J_{min} < I_{min} < H_{min} \\ \text{and } x,y \in \{t,f\} \\ \text{and } |J|,|I| \geq 1\end{array}\right. \\[4ex]
5) & \begin{array}{l}\Big\langle \mathcal{L}_n \triangleright (\nu\tilde{n})\left(\begin{array}{l}\mathsf{A} \mid \prod_{i\in I}\Big(N_i \mid B(i,x)_1^{I_{min}-1}\Big) \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^x_{j,I_{min}}\rrbracket \\ \mid \prod_{h\in H} l_h\llbracket \mathsf{R}^x_{h,I_{min}}\rrbracket \mid \prod_{k\in K} l_k\llbracket P_k\rrbracket\end{array}\right) \\ \quad ,\ \mathcal{L}_n^K \triangleright (\nu\tilde{n})\left(\begin{array}{l}\mathsf{A} \mid \prod_{i\in I}\Big(N_i \mid B(i,y)_{J_{min}}^{J^+_{min}}\Big) \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^y_{j,H_{min}}\rrbracket \\ \mid \prod_{h\in H}\Big(l_h\llbracket \mathsf{R}^{\ddot{y}}_{h,J_{min}}\rrbracket \mid B(i,y)_{J_{min}}^{J^+_{min}}\Big)\end{array}\right)\Big\rangle\end{array} & \left|\begin{array}{l}\mathsf{part}^n_1(I,J,H,K) \\ \text{and } J_{min} < H_{min} < I_{min} \\ \text{and } x,y \in \{t,f\} \\ \text{and } |J|,|H|,|I| \geq 1\end{array}\right. \\[4ex]
6) & \begin{array}{l}\Big\langle \mathcal{L}_n \triangleright (\nu\tilde{n})\Big(\mathsf{A} \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^x_{j,I_{min}}\rrbracket \mid \prod_{h\in H} l_h\llbracket \mathsf{R}^x_{h,I_{min}}\rrbracket \mid \prod_{k\in K} l_k\llbracket P_k\rrbracket\Big) \\ \quad ,\ \mathcal{L}_n^K \triangleright (\nu\tilde{n})\Big(\mathsf{A} \mid \prod_{j\in J} l_j\llbracket \mathsf{R}^y_{j,H_{min}}\rrbracket \mid \prod_{h\in H}\Big(l_h\llbracket \mathsf{R}^{\ddot{y}}_{h,J_{min}}\rrbracket \mid B(i,y)_{J_{min}}^{J^+_{min}}\Big)\Big)\Big\rangle\end{array} & \left|\begin{array}{l}\mathsf{part}^n_1(J,H,K) \\ \text{and } J_{min} < H_{min} \\ \text{and } x,y \in \{t,f\} \\ \text{and } |J|,|H| \geq 1\end{array}\right. \\[4ex]
7) & \mathcal{L}_n \triangleright \overline{ok},\quad \mathcal{L}_n^K \triangleright \overline{ok} &
\end{array}
\right\}
$$

The witness bisimulation for *strong ft-agreement* up to $n-1$ faults is given above; we leave similar but simpler witness bisimulations for *ft-validity* to the interested reader. We carry over all the shorthand notation used for the failure-free witness bisimulation together with some more: the operation $\ddot{x}$ denotes value inverse for $x \in \{t, f\}$, and is defined as $\ddot{t} = f$ and $\ddot{f} = t$; for $K \subset \{1\ldots n\}$, $\mathcal{L}_n^K$ denotes the network state $\langle \mathcal{L}_n/\{l_k \mid k \in$

$K\}, n - |K|\rangle$; $B(x,i)_j^{j+n}$ denotes the sequence of broadcasts of x for participant i from rounds j up to rounds $j+n$, that is $\prod_{r=j}^{j+n} l_r[\![\overline{x}_{i,r}]\!]$.

In essence, our witness bisimulation highlights the fact that, whereas in the left configuration (failure-free setting), participants agree on some $x \in \{t, f\}$ for $r \geq 2$, in the right configuration (dynamic failure setting), participants may take longer to agree on a value due to corrupted broadcasts. Through the use of the β-move (BFTol), our witness bisimulation abstracts over broadcast communications where a participant receives the same estimate it currently holds. This way we can focus only on cases where participants change their value as a result of a broadcast, thus converging towards agreement. Moreover, through the structural rule (s-Dead), we can abstract over dead code and map the corresponding live participant in a failure free setting, irrespective of its state, to the inert process $\mathbf{0}$. The witness bisimulation partitions the n participants into 4 mutually exclusive sets, based on the higher combination of participant states possible in a dynamic failure setting:

- I denotes the participants that are yet un*initialised*.
- K denotes the participants that are *killed*.
- J denotes the participants that *agree* on x, the value being broadcasted.
- H denote the participants agreeing on $\ddot{x}$ which still need to accept (and converge to) the broadcasted value x.

The value being broadcasted in a dynamic failure setting depends on the live participant with the lowest index. In the witness bisimulation above, case (3) describes the case when the live participant with the lowest index i is uninitialised (thus no broadcasts); case (4) describes the case when the live participant with the lowest index j is initialised with x, and all initialised participants agreeing on x are blocked because I_{min} is yet to be initialised; case (5) is similar to case (4), only that participants converged on x are blocked on an initialised participant with estimate $\ddot{y}$ that still needs to consume a broadcast and converge; case (6) is a special case of (5) where there are no uninitialised participants. Thus, in this last case, (6), we map live blocked participants in a dynamic failure setting to unblocked participants in a failure free setting at the final round n. The main transitions in this bisimulation relation are overviewed in Appendix A.

6 Conclusion

We have designed a *partial-failure* process calculus in which distributed consensus algorithms can be formally described and analysed. We have also developed up-to techniques in this calculus by identifying novel confluent moves involving the choice operator and the *fail* operator, together with a stronger structural equivalence abstracting over dead code. Most importantly however, we have proposed a methodology for formally proving the correctness of distributed algorithms in the presence of failure using fault-tolerance bisimulation techniques. We have shown how this methodology can alleviate the burned of exhibiting such formal proofs by giving, what to our knowledge is, the first bisimulation-based proof of Consensus with perfect failure detectors.

Future Work: There are various possible extension to our calculus. We can weaken our failure detectors to $\diamond\mathcal{S}$, [2], by enhancing our network representation with two livesets, suspectable and non-suspectable, similar to the techniques used in [14, 13]. We can also introduce recursive computation, which would allow us to study consensus solving algorithms with no static bounds on the number of rounds. Such a study would require more sophisticated reasoning about termination; work such as [3, 17] should shed more light on this complication. Independent of the calculus, we plan to validate our proposed methodology by applying it to a range of fault-tolerant distributed algorithms expressed in various calculi; examples of such algorithms include those in [11, 16].

Related Work: The confluence of certain τ-steps has long been known as a useful technique in the management of bisimulations, [9]. See [8] for particularly good examples of where they have significantly decreased the size of witness bisimulations. We have extended the concept, by considering confluence up to a particularly strong form of structural equivalence which enables useful garbage collections to be carried out in fault-tolerance proofs, by virtue of the presence of dead locations.

The closest to our work is [14], where the correctness of a consensus solving algorithm for a more complex setting which uses $\diamond\mathcal{S}$ failure detectors is formalised using a process calculus. Their proof methods however, differ from ours: they give a translation from the calculus encoding of the algorithm into an abstract interpretation and then perform correctness analysis on the abstract interpretation. Similar to our work is also [1], where the atomicity of the 2-phase commit protocol is encoded and proved correct using a process calculus with persistence and transient failure: bisimulations are used to obtain algebraic laws which are then used to prove atomicity.

References

1. Martin Berger and Kohei Honda. The two-phase commitment protocol in an extended pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 39(1), 2000.
2. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
3. Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. In *IFIP TCS*, pages 619–632, 2004.
4. Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 127–140. Springer-Verlag, 1983.
5. Cedric Fournet, Georges Gonthier, Jean Jaques Levy, and Remy Didier. A calculus of mobile agents. *CONCUR 96*, LNCS 1119:406–421, August 1996.
6. Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failures. In *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2005.
7. Adrian Francalanza and Matthew Hennessy. A theory of system fault tolerance. In L. Aceto and A. Ingolfsdottir, editors, *Proc. of 9th Intern. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'06)*, volume 3921 of *LNCS*. Springer, 2006.
8. J. F. Groote and M. P. A. Sellink. Confluence for process verification. *Theor. Comput. Sci.*, 170(1-2):47–81, 1996.

9. Jan Friso Groote and Jaco van de Pol. State space reduction using partial tau-confluence. In *Mathematical Foundations of Computer Science*, pages 383–393, 2000.
10. K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.
11. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
12. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
13. Uwe Nestmann and Rachele Fuzzati. Unreliable failure detectors via operational semantics. In *ASIAN*, pages 54–71, 2003.
14. Uwe Nestmann, Rachele Fuzzati, and Massimo Merro. Modeling consensus in a process calculus. In *CONCUR: 14th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2003.
15. James Riely and Matthew Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 226:693–735, 2001.
16. Gerard Tel. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY, USA, 1994.
17. Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the pi-calculus. *Inf. Comput.*, 191(2):145–202, 2004.

A Main Transitions for Consensus Bisimulation Proofs

We here overview the main transitions of the bisimulation proofs given in Section 5. The first witness bisimulation presented was that for ff-Agreement, which had two main groups of states, enumerated as (2) and (3).

- If we are in (2) and the j^{th} participant in the left configuration is initialised (through a τ action) with $x \in \{t, f\}$ then
  - if $j \neq 1$ the participant proceeds to round 1 with estimate x and joins set J or H accordingly. We match this action by the empty move and remain in case (2).
  - if $j = 1$ the participant proceeds to round 1 and acts as the coordinator, broadcasting x. For all participants $j \in J$ or $h \in H$, broadcast synchronisation turns out to be a β-move using (BLin), and (gc-Act) and (gc-Fail) to garbage collect inactive branches as in Example 3. At this point all initialised participants agree on the broadcasted value x at round 2, and proceed through the next rounds using β-moves, still agreeing on x, until they block again on the next I_{min}. We match this action with the empty action and progress to case (3). We note that in case (3), uninitialised participants $i \in I$ will include the broadcasted values from previous rounds that are yet to be consumed by them once they are initialised.
- If we are in (3) and the i^{th} participant is initialised then
  - if $i \neq I_{min}$ then the right configuration performs an empty move and we remain in case (3), abstracting away from the β-moves of participant i synchronising with all the broadcasts to reach round I_{min} with estimate x.
  - if $i = I_{min}$ then the matching move is similar. However we have two further sub-cases
    - If $I = \{i\}$ then all participant would have agreed on x, the first broadcasted value and we progress to case (4) through a series of β-moves.
    - If $(I/\{i\}) \neq \emptyset$ then all initialised participants $j \in J$ progress to $(I/\{i\})_{min}$ and we remain in case (3)

The second relation presented in Section 5 is the witness bisimulation up-to-β for ft-Agreement. We overview the main transitions of the important (enumerated) stages in this relation, that is for stages (3), (4), (5) and (6):

Stage (3)**:** If participant $i \in I$ is initialised, then we go to stage (4) or (5), depending on the value y it is initialised to and whether $(I/\{i\})^{+}_{min} < J_{min}, H_{min}$. If participant $i \in I$ dies, then if $(I/\{i\})^{+}_{min} < J_{min}, H_{min}$ we remain in stage (3) else go to stage (4) or (5).

Stage (4)**:** If participant $j \in J$ is killed, then if $J_{min} = J^{+}_{min}$ we transition to stage (3), otherwise we remain in (4). If participant $i \in I$ is initialised, we remain in (4), unless the I_{min} participant is initialised to $\ddot{y}$, in which case we transition to either stage (5) if $|I| \neq 1$, stage (6) if $|I| = 1$ or stage (7) if $|H| = 0$ and $|I| = 1$. Similarly, if participant I_{min} dies, then depending on the next smallest participant every $j \in J$ blocks on, we can either remain in (4) or transition to stage (5) if $|I| \neq 1$, stage (6) if $|I| = 1$ or stage (7) if $|H| = 0$ and $|I| = 1$. Finally, if participant $h \in H$ consumes the broadcasts or dies, we still remain in stage (4).

Stage (5)**:** If participant $j \in J$ is killed, then if $J_{min} \neq J^{+}_{min}$ we remain in (5), otherwise we transition to stage (4) where participants $h \in H$ take the place of participants $j \in J$. If participant $h \in H$ accepts the broadcast or dies, we remain in (5) or transition back to (4), depending on whether $H_{min} = H^{+}_{min}$. If participant $i \in I$ is initialised, we still remain in (5) whereas if $i \in I$ dies, we remain in (5) or transition to (6) if $|I| = 1$.

Stage (6)**:** If participant $j \in J$ dies, then if $|J| = 1$ we reach agreement and go to stage (7), otherwise we remain in (6), possibly swapping participants $h \in H$ for participants $j \in J$. If participant $h \in H$ accepts the broadcast or dies, we transition to stage (7) if $|H| = 1$ or remain in (6).

All the above transitions are matched by the empty transition on the failure-free side, except those transitions that involve initialising participants: In this case we match the transition by initialising the corresponding participant in the failure-free setting.