Xjenza Online - Journal of The Malta Chamber of Scientists
www.xjenza.org
DOI: 10.7423/XJENZA.2014.2.04

Research paper

Formal Proofs for Broadcast Algorithms

Mandy Zammit[1] **and Adrian Francalanza**[1]
[1]University of Malta, Faculty of ICT, Department of Computer Science

Abstract. Standard distributed algorithmic solutions to recurring distributed problems are commonly specified and described informally. A proper understanding of these distributed algorithms that clarifies ambiguities requires formal descriptions. However, formalisation tends to yield complex descriptions. We formally study two broadcast algorithms and present an encoding framework using a process descriptive language and formalise these algorithms and their specifications using this framework. Following these new formal encodings we discuss *correctness* proofs for the same algorithms.

Keywords Distributed Algorithms – Broadcast Algorithms – Correctness

1 Introduction

Distributed Systems are decentralised systems made up of *autonomous*, *concurrent*, *interconnected* processes working towards a *common goal* (Tel, 1994; Lynch, 1996). Unlike standalone systems, where if something fails then all else fails, processes in a *distributed system* act as units of failures. This means that distributed systems may undergo *partial-failure* (as opposed to *total system failure*) where the failure of a process in such a system does not imply failure of the whole system.

Distributed problems are known recurring situations which arise when programming distributed systems. Two well known distributed problems are *consensus* (agreement between all processes) and *reliable broadcast* (reliable interprocess communication). *Standard Distributed Algorithms* are algorithmic solutions to these known recurring distributed problems. These distributed algorithms are specified through a number of correctness criteria. Correctness criteria specify the expected (*correct*) behaviour of the distributed algorithm that is being specified. For every distributed problem specification there may be various algorithmic solutions, each aimed to observe the behaviour specified by the correctness criteria.

The current modus operandi of describing distributed algorithms and their specifications is using informal or semi-formal descriptions. Specifications are normally expressed in natural language, whilst distributed algorithms are commonly expressed using some semi-formal pseudo-code, lacking any type of formal semantics (Tel, 1994; Lynch, 1996; Guerraoui and Rodrigues, 2006).

There are a number of reasons as to why the status quo is unsatisfactory. For one, there is a discrepancy between such potentially ambiguous semi-formal descriptions and the implementation of such algorithms (in a machine computable language), which is a large source of erroneous behaviour in the deployment of the system. Another reason why informal descriptions and specifications are deficient is that no formal verification of the algorithms may be done. Formal analysis is essential because such algorithms are very easy to get wrong. Given the concurrent nature of distributed systems, each execution results in a different execution path: if we want to verify that our algorithm observes the behaviour specified by the correctness criteria, then we need to reason on every possible path of execution of the algorithm, something that can only be done through proper formal analysis.

As in Francalanza and Hennessy (2007), Kühnrich and Nestmann (2009), Nestmann et al. (2003) we postulate that it would be desirable to encode both the algorithm and its specification in some formal process description language and then equate their behaviour using some notion of equivalence (as in Equation 1).

$$System = Specification \tag{1}$$

In this paper we focus on Broadcast Algorithms; a small subset of Distributed Algorithms. The rest of the paper is structured as follows; Section 2 describes the Distributed Broadcast problem, and presents two specifications together with an algorithmic solution for each specification. Section 3 presents a *partial-failure* calculus that is able to encode both specification and implementation of these algorithms. Section 4 expresses an encoding framework for broadcast algorithms and their specifications. Section 5 presents an encoded broadcast algorithm using the calculus and encoding framework. Finally section 6 discusses the respective proofs for these broadcast algorithms, and sections 7 and 8 conclude.

Correspondence to: Mandy Zammit (mandy.zammit.08@um.edu.mt)

2 Distributed Broadcast Problem

Broadcast is used by an entity to disseminate messages to a number of other entities including itself. Broadcast algorithms are a fundamental building block in the programming of distributed systems. There are a number of different broadcast specifications, each differing in the reliability requirements that it guarantees. We explore two different broadcast specifications and two algorithmic solutions; each one aimed at satisfying one of the broadcast specification as expressed in (Guerraoui and Rodrigues, 2006); the algorithms assume perfect failure detectors and are also taken from (Guerraoui and Rodrigues, 2006).

Terminology: ***Broadcast*** is the term used to describe the transition of a message by an entity (broadcaster) to all other entities (including itself) in a given network. This transmitted message is then said to be ***received*** by the other individual entities in the network. However its receipt is only permanently confirmed once it is ***delivered*** to some higher level of abstraction. A ***correct*** process is one which does not to fail during execution, whilst a ***faulty*** process may fail at any point in time during the execution of an algorithm.

2.1 Best-Effort Broadcast

The *Best-Effort* broadcast is a simple broadcast specification with a weak form of reliability, whereby if the broadcaster of a message fails at any point during execution, then no entities are expected to deliver the message. The set of correctness criteria that embody the *best-effort* broadcast specification are:

- ***Validity:*** *(liveness)* For any two processes p_i and p_j. If p_i and p_j are *correct*, then every message broadcast by p_i is eventually delivered by p_j.
- ***No Duplication:*** *(safety)* No message is delivered more than once.
- ***No Creation:*** *(safety)* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

A *liveness* property asserts that some "good" behaviour eventually happens, whereas a *safety* property asserts that nothing wrong should happen during execution.

Algorithm 1 Best Effort Broadcast (adapted from Guerraoui and Rodrigues (2006)

```
1: while ⟨ bebBroadcast — m⟩ do
2:     for p_i ∈ ∏ do trigger ⟨ pp2pSend — p_i, m⟩;
3:     end for
4: end while
5:
6: while ⟨ pp2pDeliver — p_i, m⟩ do
7:     trigger ⟨ bebDeliver — p_i, m⟩;
8: end while
```

Algorithm 1 depicts an algorithmic solution to the *Best-Effort* Specification presented by Guerraoui and Rodrigues (2006). Lines 1 to 4 express the broadcast of a message, whilst lines 6 to 8 describe delivery of a message.

Algorithm 2 (Lazy) Regular Reliable Broadcast (adapted from Guerraoui and Rodrigues (2006))

```
1: while ⟨ Init ⟩ do
2:     delivered := ∅; correct := ∏;
3:     for p_i ∈ ∏ do from[p_i] := ∅;
4:     end for
5: end while
6:
7: while ⟨ rbBroadcast — m⟩ do
8:     trigger ⟨ bebBroadcast — [DATA, self, m] ⟩;
9: end while
10:
11: while ⟨ bebDeliver — p_i, [DATA, s_m, m] ⟩ do
12:     if m ∉ delivered then
13:         delivered := delivered ∪{m};
14:         trigger ⟨ rbDeliver — s_m, m⟩;
15:         from[p_i] := from[p_i] ∪{(s_m, m)};
16:         if p_i ∉ correct then
17:             trigger ⟨ bebBroadcast — [DATA, s_m, m] ⟩;
18:         end if
19:     end if
20: end while
21:
22: while ⟨ crash — p_i ⟩ do
23:     correct := correct \{p_i};
24:     for (s_m, m) ∈ from[p_i] do
25:         trigger ⟨ bebBroadcast — [DATA, s_m, m] ⟩;
26:     end for
27: end while
28:
```

2.2 Regular Reliable Broadcast

The *Regular Reliable* broadcast specification has a stronger form of reliability than the *Best-Effort* broadcast, whereby it requires that, if at least one *correct* process delivers a broadcast message then all other *correct* entities in the network must deliver the same message. This specification shares the same *three* correctness criteria of the *best-effort* broadcast specification, with the addition of a new *liveness* property:

- ***Agreement:*** *(liveness)* If a message m is delivered by some correct process p_i, then m is eventually delivered by every correct process p_j.

Algorithm 2 is an algorithmic solution presented in Guerraoui and Rodrigues (2006) satisfying this specification. Lines 1 to 5 initialise the setting for this algorithm. Broadcast of a message is defined from line 7 to 9, while lines 11 to 15 express delivery of a message, introducing checks for no duplication. Lines 16 to 18 and 22 to 27 re-instantiate a broadcast if the sender of a message is detected to be faulty.

The problem with these algorithmic representations is that they are expressed using some pseudo-code which has no formal semantics. Secondly the main source of complication is the analysis of the interleaving of all the processes, which is not portrayed in these representations. It is also difficult to relate formally the specifications to the actual

code in a formal language. What's more, there exists an implicit problem: we expect that these specifications hold over all possible participants in a broadcast network.

3 A *Partial-Failure* Calculus

The *partial-failure* calculus presented in Francalanza and Hennessy (2007) formalises the characteristics of distributed systems in terms of concurrent process, independently failing locations, inter-process communication and failure detection.

Figure 1 defines the syntax of an extended version of the *partial-failure* calculus. It has two syntactic categories; namely *Processes* and *Systems*. *Processes* may contain nil which is the simplest construct of the language and represents termination. Input $(a?\vec{x})$ and output $(a!\vec{v})$ terms allow vector values $\vec{v}$ to be inputted (when matching the pattern of $\vec{x}$) or outputted on channel a. Processes may be composed of other processes running concurrently $(P \mid Q)$. Visibility of channels may be localised to a subset of processes through channel scoping $(\nu\, a)P$. Such scoping disallows any extenral communication interference on channel a. Two processes may preempt each other if they are composed by *choice* $(P + Q)$. That is, if a part of P executes first, then any execution from Q is preempted, and vice-versa. Processes may contain a *matching* construct which test for the identity of value vectors $\vec{v_1}$ and $\vec{v_2}$ and proceeds to P or Q accordingly. *Action renaming* $P[^a/_b]$ allows the renaming of (input and output) actions. *Failure-detection* is carried out using the *failure detector* construct (susp $l.P$). This is a guarding construct which tests for the liveness of location l and releases process P once it *correctly* suspects that l has failed. We extend these constructs with processes which may contain the zero construct, which guards process P, releasing it once it detects that no more failures may be induced in the network. This construct is novel to our setting and is only used as a specification construct, i.e., to describe correctness criteria. *Systems* are made up of processes residing at some arbitrary location l $(l[|P|])$. They may also be composed of other systems executing in parallel $(S \mid T)$, may contain scoped channels $((\nu\, a)S)$, or renamed channel names $(S[^a/_b])$.

Notations: A series of parallel processes $P_1 \mid \ldots \mid P_n$ is denoted by $\prod_{i=1}^{n} P_i$, whilst a series of choices $P_1 + \cdots + P_n$ is denoted by $\sum_{i=1}^{n} P_i$. We denote the scoping of more than one channel in P by $(\nu\, \tilde{n})P$, where $\tilde{n}$ is the set of channel names that are to be scoped. On the other hand when more than one action renaming is needed on a process, we denote it as $[^{\alpha_1}/_{\beta_1}, \ldots, ^{\alpha_n}/_{\beta_n}]$.

The operational semantics of the language is defined in terms of *configurations* of the form

$$\langle \mathcal{L}, n \rangle \triangleright S$$

where $\langle \mathcal{L}, n \rangle$ represents the distributed network state and S is the distributed system. $\mathcal{L}$ is a *liveset*, which is defined as a set of locations $\{l_1, \ldots, l_n\} \in$ Loc which are alive (Francalanza and Hennessy, 2007). A special type of location, the *immortal location* denoted by $\star$ is assumed to be included in every *liveset* $\mathcal{L}$. On the other hand n represents a bounded number of possible dynamic *location failures* in the network. Intuitively $\langle \mathcal{L}, n \rangle \triangleright\ S$ denotes the system S with its components (or processes) running on locations $\mathcal{L}$, subject to n possible location failures from $\mathcal{L}$. Computation is define by transitions between tuples of configurations as follows

$$\langle \mathcal{L}, n \rangle \triangleright\ S \xrightarrow{\alpha} \langle \mathcal{L}', n' \rangle \triangleright\ S'$$

where α is an action defined as

$$\begin{array}{rcll} \alpha \in \textsc{Act} & ::= & a?\vec{v} & \textit{Input} \\ & \mid & a!\vec{v} & \textit{Output} \\ & \mid & \tau & \textit{Internal} \end{array}$$

Weak actions $\stackrel{\hat{\alpha}}{\Longrightarrow}$ denote either

$$\xrightarrow{\tau}{}^{*} \xrightarrow{\alpha}{}^{*} \xrightarrow{\tau}{}^{*} \quad if\ \alpha \in \{a?\vec{v},\ a!\vec{v}\}$$

or

$$\xrightarrow{\tau}{}^{*} \quad if\ \alpha = \tau$$

and may also be used to transition between tuples.

The semantic rules of this calculus are given in Figure 1: transitions resulting in an unchanged network have this network omitted from the left tuple. Rule In, Out, and Com describe inter-process communication across a network. We note that we require l, the location where the communication is occurring to be alive. Rule Halt describes the failure of a location whereas, rule Susp describes the detection of a failed location. Rule Zero (a novel addition to the semantics of the calculus) triggers only when no more failures may be induced in the network. The remaining rules are fairly standard (see Francalanza and Hennessy (2007) for further details). The transition semantics of the *partial-failure* calculus induces the usual definition of weak bisimulation equivalence defined as follows

Definition 1 (Weak bisimulation equivalence)**.** Denoted as $\approx$, is the largest relation over configurations such that if $\langle \mathcal{L}_1, n_1 \rangle \triangleright S_1 \approx \langle \mathcal{L}_2, n_2 \rangle \triangleright S_2$ then

- $\langle \mathcal{L}_1, n_1 \rangle \triangleright S_1 \xrightarrow{\alpha} \langle \mathcal{L}'_1, n'_1 \rangle \triangleright S'_1$ implies $\langle \mathcal{L}_2, n_2 \rangle \triangleright S_2 \stackrel{\hat{\alpha}}{\Longrightarrow} \langle \mathcal{L}'_2, n'_2 \rangle \triangleright S'_2$ such that $\langle \mathcal{L}'_1, n'_1 \rangle \triangleright S'_1 \approx \langle \mathcal{L}'_2, n'_2 \rangle \triangleright S'_2$
- $\langle \mathcal{L}_2, n_2 \rangle \triangleright S_2 \xrightarrow{\alpha} \langle \mathcal{L}'_2, n'_2 \rangle \triangleright S'_2$ implies $\langle \mathcal{L}_1, n_1 \rangle \triangleright S_1 \stackrel{\hat{\alpha}}{\Longrightarrow} \langle \mathcal{L}'_1, n'_1 \rangle \triangleright S'_1$ such that $\langle \mathcal{L}'_1, n'_1 \rangle \triangleright S'_1 \approx \langle \mathcal{L}'_2, n'_2 \rangle \triangleright S'_2$

♦

4 Encoding Framework using the *Partial-Failure* Calculus

The *Partial-failure* calculus is expressive enough to encode both the broadcast specifications and algorithms in Section 2. We adhere to the following conventions.

4.1 Encoding Algorithms

- ***Independently failing participants*** Participants in broadcast algorithms are autonomous and can fail independently at any given point in time during execution. We encode this characteristic as code running on dedicated locations $l_n|[P_n|]$, where P_n denotes the code of participant n. Location l_n is a unit of failure,

Processes

$$
\begin{array}{lllllll}
P,Q \in \textsc{Proc} & ::= & \mathsf{nil} & \textit{(Inert)} & \mid & a?\vec{x}.P & \textit{(Input)} \\
& \mid & a!\vec{v} & \textit{(Output)} & \mid & (\nu\, a)P & \textit{(Channel Scoping)} \\
& \mid & P \mid Q & \textit{(Parallel Composition)} & \mid & P + Q & \textit{(Choice)} \\
& \mid & \mathsf{if}\ \vec{v_1} = \vec{v_2}\ \mathsf{then}\ P\ \mathsf{else}\ Q & \textit{(Matching)} & \mid & P[\alpha/\beta] & \textit{(Rename)} \\
& \mid & \mathsf{susp}\ l.P & \textit{(Failure Detector)} & \mid & \mathsf{zero}.P & \textit{(Zero)}
\end{array}
$$

Systems

$$
\begin{array}{lllllll}
S,T \in \textsc{Sys} & ::= & l|[P|] & \textit{(Located Processes)} & \mid & S \mid T & \textit{(Parallel Composition)} \\
& \mid & (\nu\, a)S & \textit{(Channel Scoping)} & \mid & S[\alpha/\beta] & \textit{(Rename)}
\end{array}
$$

Transition Rules

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

$$\textsc{In}\quad \frac{}{\langle \mathcal{L}, n\rangle \triangleright\ l|[a?(\vec{x}).P|] \xrightarrow{a?\vec{v}} l|[P\{\vec{v}/\vec{x}\}|]}$$

$$\textsc{Out}\quad \frac{}{\langle \mathcal{L}, n\rangle \triangleright\ l|[a!(\vec{v}).\mathsf{nil}|] \xrightarrow{a!\vec{v}} l|[\mathsf{nil}|]}$$

$$\textsc{Susp}\quad \frac{}{\langle \mathcal{L}, n\rangle \triangleright\ l|[\mathsf{susp}\ k.P|] \xrightarrow{\tau} l|[P|]}\ k \notin \mathcal{L}$$

$$\textsc{Halt}\quad \frac{}{\langle \mathcal{L}, n+1\rangle \triangleright\ S \xrightarrow{\tau} \langle \mathcal{L} \setminus l, n\rangle \triangleright\ S}$$

$$\textsc{Fork}\quad \frac{}{\langle \mathcal{L}, n\rangle \triangleright\ l|[P \mid Q|] \xrightarrow{\tau} l|[P|] \mid l|[Q|]}$$

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

$$\textsc{RenP}\quad \frac{}{\langle \mathcal{L}, n\rangle \triangleright\ l|[(P)\rho|] \xrightarrow{\tau} (l|[P|])\rho}$$

$$\textsc{Eq}\quad \frac{}{\langle \mathcal{L}, n\rangle \triangleright\ l|[\mathsf{if}\ v = v\ \mathsf{then}\ P\ \mathsf{else}\ Q|] \xrightarrow{\tau} l|[P|]}$$

$$\textsc{Neq}\quad \frac{}{\langle \mathcal{L}, n\rangle \triangleright\ l|[\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ P\ \mathsf{else}\ Q|] \xrightarrow{\tau} l|[Q|]}\ v_1 \neq v_2$$

$$\textsc{Zero}\quad \frac{}{\langle \mathcal{L}, 0\rangle \triangleright l|[\mathrm{zero}.P|] \xrightarrow{\tau} \langle \mathcal{L}, 0\rangle \triangleright l|[P|]}$$

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

$$\textsc{Ren}\quad \frac{\langle \mathcal{L}, n\rangle \triangleright\ S \xrightarrow{\alpha} \langle \mathcal{L}, n\rangle \triangleright\ S'}{\langle \mathcal{L}, n\rangle \triangleright\ (S)\rho \xrightarrow{(\alpha)\rho} \langle \mathcal{L}, n\rangle \triangleright\ (S')\rho}\ \rho = [a/b]$$

$$\textsc{Rest}\quad \frac{\langle \mathcal{L}, n\rangle \triangleright\ S \xrightarrow{\alpha} \langle \mathcal{L}', n'\rangle \triangleright\ S'}{\langle \mathcal{L}, n\rangle \triangleright\ (\nu\, a)S \xrightarrow{\alpha} \langle \mathcal{L}', n'\rangle \triangleright\ (\nu\, a)S'}\ \textsc{Chan}\alpha \in \mathrm{fc}\,((\nu\, a)S)$$

$$\textsc{Par}\quad \frac{\langle \mathcal{L}, n\rangle \triangleright\ S \xrightarrow{\alpha} \langle \mathcal{L}', n'\rangle \triangleright\ S'}{\langle \mathcal{L}, n\rangle \triangleright\ S \mid T \xrightarrow{\alpha} \langle \mathcal{L}', n'\rangle \triangleright\ S' \mid T}$$

$$\textsc{Com}\quad \frac{\langle \mathcal{L}, n\rangle \triangleright\ S \xrightarrow{a?\vec{v}} S' \qquad \langle \mathcal{L}, n\rangle \triangleright\ T \xrightarrow{a!\vec{v}} T'}{\langle \mathcal{L}, n\rangle \triangleright\ S \mid T \xrightarrow{\tau} S' \mid T'}$$

Figure 1: Partial-Failure Calculus (DPC)

whereby it can be marked as a dead location (by being removed from the liveset $\mathcal{L}$) during execution ceasing execution of code residing on it.

- ***Network initialisation*** An output action on some channel c_n will denote the initialisation of the network, where process P_n would be chosen as the designated broadcaster. c channels are not localised to the network, since initialisation of the network is expected to be triggered from the environment outside the network.
- ***Broadcast*** Broadcast in the algorithms that we study happens over *perfect point-to-point* links. We use the constructs of receiving ($b?x$) and sending ($b!m$) messages over channels to encode perfect point-to-point links. Channels may in general have multiple senders and receivers, but we choose to restrict channels to be used in a *linear* fashion, where only one process may send messages on the channel, and only one other process may receive messages from the same channel. More precisely the channel b_n^m is the channel used by process P_n to send messages to P_m. Dually the channel b_m^n is used by process P_m to send messages to P_n. This linear usage is further ensured by the use of scoping to localise these channels to the network which is using

them. Broadcast is thus encoded as the output of a message over the encoded perfect point-to-point links, discussed previously. Dually the receipt of a broadcasted message is encoded as the input action over the same perfect point-to-point links.

- ***Deliver*** Delivery of a message by some participant P_n will be encoded as the output of the message over some channel d_n which is not scoped. The reason why the channel d_n is not scoped is because delivery of a message is the process of passing the received message to another entity external to the broadcast algorithm.
- ***Failure Detection*** Failure detection is immediately expressed using the suspect ($susp\ l.P$) construct of the *partial-failure* calculus that we use for our encoding.

4.2 Encoding Specifications

In Francalanza and Hennessy (2007), Francalanza and Hennessy propose the decomposition of specification during the encoding process, i.e., instead of encoding all the properties of the specification as one process, they use dedicated *wrapper code* or *testing harnesses* for each correctness criteria of the specification. These dedicated harnesses are able to wrap a system and stop any interaction between foreign entities and the network. A harness will then induce a network broadcast and observe the behaviour of the algorithm whilst determining if its behaviour conforms with the behaviour expected by the correctness criteria which is being tested for. This means that now to verify our algorithms we will have a number of equations (one for each correctness criteria in the specification) as follows

$$(\nu\,\tilde{n})(System \mid Harness_1) \approx SimpleSpecification$$
$$\vdots$$
$$(\nu\,\tilde{n})(System \mid Harness_n) \approx SimpleSpecification$$

The structure of the *SimpleSpecification* will be determined on the type of property which is being tested for. If the criteria being tested for is a *safety* property, the respective simple specification would entail an *inert* (nil) process, whilst if testing for a *liveness* property the simple specification would consists of an output on channel OK. Since a *safety* property asserts that *nothing bad* happens, we construct harnesses which wrap and observe systems, triggering a signal on channel *nok* whenever violations are detected. Stated otherwise, if a *safety* property is satisfied no output on *nok* is generated. Conversely, harnesses wrapping systems and testing for *liveness* properties, output OK signals when the expected behaviour is observed. The following set of characteristics are prevalent to the encoding of testing harnesses.

- ***Reliable code*** It is required that wrapper code testing for a property in a distributed algorithm is reliable, that is, it should never fail unlike the participants in the distributed algorithm. Any wrapper code will thus be located at the *immortal location* $\star$.
- ***Initiators*** A harness commences testing by sending the start message on channel c_i to process P_i, the designated broadcaster. This mimics the initiation of a broadcast from an outside environment and allows the harness to induce the network and test for a specific correctness criteria accordingly.
- ***Testing Mechanism*** The observation of message deliveries is modeled by the input on channels $d_1 \ldots d_n$ where n is the number of participants in the network under test. After consuming these deliveries (produced by participants in the network), each testing harness will execute specific code that acts upon the observed delivery behaviour.

$$\mathrm{RRB}_k^n \triangleq \begin{cases} l_0|[\mathsf{nil}|] & k = 0 \\ (\nu\,\tilde{n}) \left(\begin{array}{l} \mathrm{RRB}_{k-1}^n \\ \mid l_k|[P_{\mathrm{BEB}_k^n}[c_k/c'_k,\, d_k/d'_k]|] \\ \mid l_k|[P_{\mathrm{RRB}_k}|] \end{array} \right) & k > 0 \end{cases}$$

where $0 \leq k \leq n$

Best-Effort Broadcast related Processes

$$P_{\mathrm{BEB}_k^n} \triangleq B_{\mathrm{BEB}_k^n} \mid D_{\mathrm{BEB}_k^n}$$
$$B_{\mathrm{BEB}_k^n} \triangleq c_k?\vec{x}.\prod_{j=1}^{n} b_k^j!\vec{x}$$
$$D_{\mathrm{BEB}_k^n} \triangleq \sum_{j=1}^{n} \left(b_j^k?\vec{x}.d_k!\vec{x}\right)$$

Regular Reliable Broadcast related Processes

$$P_{\mathrm{RRB}_k} \triangleq B_{\mathrm{RRB}_k} \mid D_{\mathrm{RRB}_k}$$
$$B_{\mathrm{RRB}_k} \triangleq c_k?\vec{x}.c'_k!(k,\vec{x})$$
$$D_{\mathrm{RRB}_k} \triangleq d'_k?(s,\vec{m}).(d_k!\vec{m} \mid \mathsf{susp}\ l_s.c_k!(s,\vec{m}))$$
$$\tilde{n} = \{b_k^1,\ldots,b_k^k\} \cup \{b_1^k,\ldots,b_k^k\} \cup \{c'_k\} \cup \{d'_k\}$$

Figure 2: Inductive Formal Encoding of (Lazy) Regular Reliable Broadcast

5 Encoding Regular Reliable Broadcast

We present the formal encoding of the regular reliable broadcast algorithm, (which uses the Best Effort Broadcast) and the formal encoding of two *testing harnesses*; one testing for the No Duplication (safety) property and another testing for the Agreement (liveness) property in this same algorithm. The other properties follow the same pattern.

Broadcast specifications quantify over all possible participants in a network, requiring us to provide a proof for *every* instance of participants. We therefore present an inductive formal encoding for the regular reliable broadcast, which will enable us to construct inductive proofs that cover all

$$\text{ND}_n^{i,\vec{m}}[-] \triangleq (\nu\,\tilde{n})(\star|[I_i^{\vec{m}} \mid T_n|] \mid [-])$$

where $0 \leq i \leq n$

$$I_i^{\vec{m}} \triangleq \begin{cases} \mathsf{nil} & i = 0 \\ c_i!\vec{m} & i > 0 \end{cases} \qquad (Initiator)$$

$$T_n \triangleq \begin{cases} \mathsf{nil} & n = 0 \\ T_{n-1} \mid T_n^{'} & n > 0 \end{cases} \qquad (Testers)$$

$$T_n^{'} \triangleq d_n?\vec{x}.d_n?\vec{y}.nok!$$

$$\tilde{n} = \{d_1, \ldots, d_n\} \cup \{c_1, \ldots, c_n\}$$

Figure 3: Formal Encoding of No Duplication Harness

$$A_n^{i,\vec{m}}[-] \triangleq (\nu\,\tilde{n})(\star|[I_i^{\vec{m}} \mid T_n \mid T_0^n|] \mid [-])$$

where $0 < i \leq n$

$$I_i^{\vec{m}} \triangleq c_i!\vec{m} \qquad (Initiator)$$

$$T_0^n \triangleq \mathsf{zero}.(OK! + \sum_{j=1}^{n} d_j?\vec{x}.(d_j!\vec{x} \mid ok_0!))$$

$$T_n \triangleq \begin{cases} T_1^{'} & n = 1 \\ T_{n-1}[^{OK}/_{ok_{n-1}}] \mid T_n^{'} & n > 1 \end{cases} \qquad (Testers)$$

$$T_n^{'} \triangleq ok_{n-1}?.((d_n?\vec{x}.OK!) + (\mathsf{susp}\ l_n.OK!))$$

$$\tilde{n} = \{d_1, \ldots, d_n\} \cup \{ok_0, \ldots, ok_{n-1}\} \cup \{c_1, \ldots, c_n\}$$

Figure 4: Formal Encoding of Agreement Harness

cases of participant numbers.

Figure 2 denotes the formal encoding RRB_k^n of the Regular Reliable Broadcast Algorithm (Algorithm 2), whereby RRB_k^n stands for the network of n ultimate participants, but k actual participants yet in the network. Participants are added inductively in this network until k becomes equal to n. Recall from Algorithm 2, that Regular Reliable Broadcast uses the Best-Effort Broadcast, thus Participants in RRB_k^n are made up of Best-Effort related code $P_{\text{BEB}_k^n}$ and Regular Reliable related code $P_{\text{RRB}_k^n}$.

Best-Effort related code $P_{\text{BEB}_k^n}$ contains broadcast and delivery threads; $B_{\text{BEB}_k^n}$ and $D_{\text{BEB}_k^n}$ respectively. The broadcast thread $B_{\text{BEB}_k^n}$ corresponds to lines 1 to 3 in Algorithm 1 and is only instantiated if there is an input action on the channel c_k. Once instantiated it will broadcast the message to all other entities in the network. The delivery thread $D_{\text{BEB}_k^n}$ on the other hand corresponds to lines 6 to 8 in Algorithm 1, whereby it allows the receipt of one broadcast message.

Regular Reliable related code $P_{\text{RRB}_k^n}$ contain broadcast $B_{\text{RRB}_k^n}$ and delivery $D_{\text{RRB}_k^n}$ threads as well. The broadcast thread $B_{\text{RRB}_k^n}$ corresponds to lines 7 to 9 in Algorithm 2, where once a message is received by some P_i over channel c_i it is forwarded to the *Best-Effort* broadcast underneath. The deliver thread $D_{\text{RRB}_k^n}$ corresponds to lines 11 to 20 and 22 to 27 in Algorithm 2, where if a delivery is received from the Best-Effort broadcast underneath (over channel $d_k^{'}$), the delivery thread delivers (over channel d_k) and releases the code which tests for the liveness of the owner of the received message, and starts a new broadcast with the same message if the process is suspected to have failed.

Figure 3 denotes the formal encoding of the No Duplication *testing harness*. This Harness $\text{ND}_n^{i,\vec{m}}$ tests for the No Duplication property in the Regular Reliable Broadcast algorithm, and follows the conventions outlined in Section 4. Broadcast initiation is done through process $I_i^{\vec{m}}$, which disseminates a message on channel c_i. The testers in this harness $T_n^{'}$, then wait for a delivery on channel d_n, and trigger a *nok* if a second delivery is observed.

Figure 4 presents the formal encoding of the *testing harness* that tests for the Agreement property in the Regular Reliable broadcast algorithm. The testing harness $A_n^{i,\vec{m}}$ which tests for n processes in a network, induces the broadcast network with message $\vec{m}$ by designating participant P_i as the broadcaster. $I_i^{\vec{m}}$ is the initiation code of the harness which induces the broadcast network by outputting message $\vec{m}$ on channel c_i. T_0^n is the code used to initialise testing. The zero construct is used in this harness definition to restrain testing to commence only when no more failures may be induced in the network. The reason for this is that we need to identify which processes are *correct*, that is, which were guaranteed not to fail during execution. T_0^n then either outputs an OK if no participant had delivered, or if it manages to observe a delivery from any participant, it commences the other testers by outputting an ok_0. Each tester T_n where $n > 0$ then waits for an ok_{n-1}, that is, an ok signal from the Tester $n-1$. When this ok_{n-1} is received, the tester checks if the participant it is testing for P_n is alive, if it is not it promptly outputs on ok_n, but if P_n is still alive then the tester outputs on ok_n only if the participant is ready to deliver the broadcasted message.

6 Stating and Proving Correctness

We have till now presented formal encoding for the informal broadcast specifications and informal algorithmic broadcast implementation discussed in Section 2. This constitutes the main contribution of the paper. Using the Harnesses and Algorithms (formally) defined in section 5, together with the bisimulation equivalence of the *partial-failure* calculus (in which the harnesses and algorithms are expressed), we can formulate our correctness criteria in the

following form:

$$\forall n \;\; Harness(n)[System(n)] \approx SimpleSpec$$

where *SimpleSpec* is nil in the case for safety properties and *OK*! in the case of a liveness property. For instance to verify no duplication we are required to prove

$$\forall n \;\; \text{ND}_n^{i,\vec{m}}[\text{RRB}_n^n] \approx \mathsf{nil}$$

whilst to verify agreement we are required to prove

$$\forall n \;\; A_n^{i,\vec{m}}[\text{RRB}_n^n] \approx OK!$$

These are formal statements with an unambiguous semantics expressed as an equivalence amongst two systems within our calculus. Bisimulation equivalence comes equipped with an elegant (coinductive) proof technique whereby we only need to exhibit a relation that includes the afore mentioned pair and observes the transfer property of definition 1.

However these witness relations can be quite large and unwieldy to construct. Moreover, for each correctness criteria we need to exhibit an infinite number of relations, one for every instance of the broadcast network with a specific number of participants.

Francalanza and Hennessy (2007) propose the decomposition of correctness proofs into two phases; the *failure free* phase (or *basic correctness* phase) and the *correctness preservation* phase (or *fault-tolerance* phase). The first phase tests the algorithm under no failures, and equates its behaviour to a *SimpleSpec*

$$\langle \mathcal{L}, 0\rangle \triangleright (\nu\, \tilde{o})(Sys(n) \mid Harness(n)) \approx SimpleSpec(n)$$

The second phase equates the behaviour of the system under failures to the system in a failure free environment

$$\begin{gathered}\langle \mathcal{L}, n\rangle \triangleright (\nu\, \tilde{o})(Sys(n) \mid Harness(n)) \\ \approx \\ \langle \mathcal{L}, 0\rangle \triangleright (\nu\, \tilde{o})(Sys(n) \mid Harness(n))\end{gathered}$$

We adopt this technique in our Correctness proofs to alleviate the cumbersome witness bisimulations.

Furthermore, in order to address the problem of exhibiting witness bisimulations for every possible number of participants, we can adopt an inductive proof approach, whereby we show that the correctness property holds for the base case (one participant) and then the inductive case ($k+1$ participants assuming that the property holds for k participants). In (Zammit, 2013) we show how this can be done for a subset of these properties, using a technique called mocking. We however leave the construction of the full proofs for every broadcast property for future work.

7 Related Work

Kühnrich and Nestmann (2009) use a similar process description language but make use of imperfect failure detectors (Chandra and Toueg, 1996) to encode an algorithmic solution to Distributed Consensus. They point out that the technique of decomposing fault-tolerance proofs (Francalanza and Hennessy, 2007) is not quit helpful in the context of imperfect failure detectors. Nestmann et al. in (Nestmann et al., 2003) verify the Distributed Consensus algorithm developed by (Chandra and Toueg, 1996). To encode their algorithm they use a process calculi as well, but opt to represent reachable states in the consensus algorithm as a message matrix. This "global-view matrix-like representation of reachable states" contains history of messages that have been sent until now in the algorithm, and would help in the formal global reasoning about the contribution of processes to individual rounds of the algorithm.

8 Conclusion and Future Work

This paper presents the following contributions:

1. a formal description for the Best-Effort and Reliable Broadcast specification.
2. a formal description of an algorithmic implementation solving Regular Reliable Broadcast.
3. an outline of how to alleviate the burden of proving equivalences presented in Section 6.

As future work we intend to complete the correctness proofs and extend this encoding and proofing technique to other broadcast algorithms with different forms of reliability requirements.

Acknowledgment

The research work disclosed in this publication is partially funded by the Strategic Educational Pathways Scholarship Scheme (Malta). The scholarship is part-financed by the European Union – European Social Fund

References

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM* 43(2), 225–267.

Francalanza, A. and Hennessy, M. (2007). A fault tolerance bisimulation proof for consesus. In R. D. Nicola (Ed.), *Esop 2007*.

Guerraoui, R. and Rodrigues, L. (2006). *Introduction to reliable distributed programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Kühnrich, M. and Nestmann, U. (2009). On process-algebraic proof methods for fault tolerant distributed systems. (pp. 198–212). FMOODS '09/FORTE '09. Lisboa, Portugal: Springer-Verlag.

Lynch, N. A. (1996). *Distributed algorithms*. The Morgan Kaufmann Series in Data Management Systems Series. Morgan Kaufmann Publishers.

Nestmann, U., Fuzzati, R. and Merro, M. (2003). Modeling consensus in a process calculus. In *In concur: 14th international conference on concurrency theory. lncs* (pp. 393–407). Springer-Verlag.

Tel, G. (1994). *Introduction to distributed algorithms*. New York, NY, USA: Cambridge University Press.

Zammit, M. (2013). *Inductive basic correctness reasoning in formal fault-tolerance proofs for distributed algorithms* (Master's thesis, University of Malta, Malta).