On Process-Algebraic Proof Methods for Fault Tolerant Distributed Systems

Morten Kühnrich[1] and Uwe Nestmann[2]

[1] Department of CS, Aalborg University, Denmark
[2] School of EECS, Berlin Institute of Technology, Germany

Abstract. Distributed Algorithms are hard to prove correct. In settings with process failures, things get worse. Among the proof methods proposed in this context, we focus on process calculi, which offer a tight connection of proof concepts to the actual code representing the algorithm. We use Distributed Consensus as a case study to evaluate recent developments in this field. Along the way, we find that the classical assertional style for proofs on distributed algorithms can be used to structure bisimulation relations. For this, we propose the definition of uniform syntactic descriptions of reachable states, on which state-based assertions can be conveniently formulated. As a result, we get the best of both worlds: on the one hand invariant-style representation of proof knowledge; on the other hand the bisimulation-based formal connection to the code.

1 Introduction

Proof Methods for Distributed Algorithms. The wide-spread technique to describe algorithms in this field is using pseudo code, which is supposed to be self-explanatory, although it usually lacks a precise semantics; this also holds for the underlying communication network that connects distributed participants of the algorithm. Specifications of desired properties are usually expressed in natural language that often refers to terminology and concepts that are well-understood in temporal logics. Proofs in this area usually are in semi-formal style, omitting many details and reasoning steps; often, the involved proof structures are only very loosely connected to the pseudo code that describes the algorithm. Another technique to describe distributed algorithms employs automata (especially *I/O-automata* [Lyn96]). Here, the setting is more formal, although the behavior of the involved automata is still often only described via pseudo code. Proofs are carried out by induction that preserve (global) invariants along system runs; structured and hierarchical proofs are then realized through composition and hierarchical simulation methods.

The loose connection of proofs to the algorithm's description was the starting point for us to try out more syntactic methods, in our case process algebras and process calculi[1]. These come with a large set of compositional proof techniques and a powerful coinductive proof method, known as bisimulation.

[1] We prefer to use the term *process calculus* instead of *process algebra*, when we do not use proper algebraic laws. However, many people use the terms as synonyms.

Proof Methods in Process Algebra. In the context of *process calculi*, verification usually boils down to prove an equation of the form

$$\mathit{System} \approx \mathit{Specification}$$

where *System* represents a (much) more detailed description of what is prescribed by the *Specification*, but where both are described within the same conceptual and linguistic framework. The symbol ≈ denotes some kind of meaningful equivalence or, better, congruence relation; often, notions of bisimilarity are chosen due to their distinctive power and accompanying co-algebraic proof method.

Since *System* usually contains far too many observable details, one often hides those implementation details from the outside observer to make it directly comparable to the *Specification*. The standard restriction operator, usually denoted by $P\backslash a$, hides observations on channel a, which might occur within P, and keeps them internal. Sometimes, even this simple hiding method is not good enough. Then, it may come in handy to have an additional so-called *wrapper code* sit next to the *System* that filters the behavior of the latter more intelligently before it is rendered observable. Equations get the form

$$\underbrace{(\ \mathit{System} \parallel \mathit{Wrapper}\ )\backslash\{a_1,\ldots,a_n\}}_{\mathit{WrappedSystem}} \approx \mathit{SimpleSpecification}$$

where $\parallel$ means that *Wrapper* is run in parallel with *System*, communicating with it, translating its outcome, such that it becomes comparable to the *Specification*. In fact, the actual specification may often be fully encoded within the *Wrapper* such that the *SimpleSpecification* term may become trivial, e.g., just checking for a success signal emanating from the *Wrapper*. While the complexity of the specification seems to be only moved into the wrapper, without gain, one may actually profit from this transfer, because the resulting equation shows much less externally observable behavior; the remaining internal behavior of the *WrappedSystem* is often much easier to deal with. The verification method via wrappers is more or less standard [BH00] and proved helpful in the context of security [SV00] and studies on the expressive power of process calculi [Fou98], where wrappers are called *relays* and even *firewalls*.

A Proof Method for Fault Tolerance. Francalanza and Hennessy have recently proposed a method based on the above approach that, in addition, applies to the domain of fault-tolerant distributed algorithms [FH07]. Concretely, they work in a setting where processes may *fail-stop*, i.e., without recovery, augmented with so-called *perfect failure detectors*. The challenge in this setting is to verify the correctness of distributed algorithms in the context of crashes. Let $\mathit{\Gamma}_k \triangleright \mathit{Sys}$ represent a system configuration; the environment $\mathit{\Gamma}_k$ allows for k different process crashes. Interesting instances of k are $n{-}1$ or $\lceil\frac{n-1}{2}\rceil$, where n is the given number of processes in the distributed system. Then, one contribution of [FH07] is that a typical equation to be verified would be of the form:

$$(\mathit{\Gamma}_k \triangleright \mathit{WrappedSystem}) \ \approx\ (\mathit{\Gamma}_0 \triangleright \mathit{SimpleSpecification})$$

with the side-condition that the *WrappedSystem* is to be composed using a non-crashable wrapper code, while the *SimpleSpecification* is not subject to failures at all. Based on this representation, an essential contribution of [FH07] concerning proof methods is the discovery of a decomposition principle, which allows them to split the right-above equation into two more easily provable parts:

$$(\Gamma_k \triangleright \mathit{WrappedSystem}) \overset{(1)}{\approx} (\Gamma_0 \triangleright \mathit{WrappedSystem}) \overset{(2)}{\approx} (\Gamma_0 \triangleright \mathit{SimpleSpecification})$$

Here, (1) proves the fault tolerance of the *WrappedSystem*, while (2) does the actual verification w.r.t. the *SimpleSpecification.* This approach is appealing since it allows to prove (2), called *basic correctness*, without having to consider process failures. The authors exhibited this method on an arguably simple case study of a round-based Distributed Consensus algorithm in the context of perfect failure detectors ($\mathcal{P}$, in the terminology of [CT96]).

A non-trivial case study in the context of imperfect failure detectors. Our own previous work [NFM03] has been in the context of much weaker imperfect (or: unreliable) failure detectors ($\Diamond\mathcal{S}$, as of [CT96]). Therefore we had to deal with a much more complicated round-based Distributed Consensus algorithm. For its verification, we used a tailor-made distributed process calculus, similar to the one in [FH07], but at that time lacking a bisimulation theory. Moreover, to remain close and comparable to the informal proofs of Chandra and Toueg in [CT96], we did not use the traditional proof method sketched above, but followed the track of reachability analysis, based on inductions as in [Lyn96].

Thus, inspired by Francalanza and Hennessy, we set out to see in how far their bisimulation-based decomposition method also carries over to less trivial algorithms in an imperfect setting. For this, we chose the setting with imperfect failure detector $\mathcal{S}$, whose imperfection lies just between[2] the above-mentioned $\mathcal{P}$ and $\Diamond\mathcal{S}$. As a result, also the required algorithm to solve Distributed Consensus has a complexity in between the ones mentioned above. To verify this algorithm, we had to adapt the calculus of [FH07], mainly to incorporate imperfect failure detection. Then, we tried the proof method of [FH07] on this case study. The results of this undertaking is what the current paper is about.

Contributions. The main insights gained through this work are: (i) with imperfect failure detectors, the decomposition into equations (1) and (2) of [FH07] does not seem to simplify proofs; (ii) to tame the complexity of non-trivial state spaces, syntactic standard forms help characterizing the shape of reachable states[3]; (iii) we observe that invariants, as they are commonly used in traditional proofs on distributed algorithms, can be succinctly defined on the basis of standard forms; (iv) those invariants can be used to conveniently define the bisimulation relations that are used as witnesses in the respective proof method.

[2] $\mathcal{S}$ requires the existence of some non-suspectable process from the beginning, while $\Diamond\mathcal{S}$ just needs to guarantee this eventually, after a phase of uncertainty.

[3] This idea is already more or less visible in the Scheduler example of [Mil89].

Other Proof Methods. Fokkink et.al. [FGR04] pointed out that process algebras need proof methodology, not just methods. Over the years, they developed such a methodology centered on the notions of *cones* and *foci*, whose main use is to tame complicated process behavior by the identification of states where the specification and the system more directly "coincide". This methodology also includes assertional techniques and invariant proofs. However, their methodology has not yet been carried over to contexts with process failure.

On the model-checking side, we just mention two closely related examples. Kühnrich [Küh08] applies model-checking in the context of a model-driven development of an extension of the algorithm studied in the current paper. Tsuchiya and Schiper [TS08] use the model checker Spin to verify asynchronous round-based consensus algorithms. By abstraction, they manage to reduce the state space (with infinite runs) to a finite one that can be model checked. However, they can still only check correctness for fixed network sizes.

2 Distributed Process Calculi for Fault Tolerance

In this section, we introduce a distributed process calculus with process crashes and failure detection, inspired by [NFM03,NF03,FH07,Hen07]. The process model is standard, equipped with the following properties: (i) channel-based synchronous passing of values, (ii) user defined functions, (iii) recursion through parameterized process constants. Since there is no name-passing, the calculus is more like CCS than the Pi Calculus. As a novelty, it incorporates both perfect ($\mathcal{P}$) and imperfect ($\mathcal{S}$) failure detectors, as defined by Chandra and Toueg [CT96].

2.1 Syntax

We use four layers of the syntax (cf. Table 1): data values, guarded processes, processes, and networks. We assume the existence of a countably infinite set of channel, variable, and function names $\mathbf{A} = \{a, b, c, ...\}$ and a finite set $\mathcal{L}oc$ of *location names* that contains the special name $\star$.

Data values and expressions. $\perp$ denotes unknown values; integers are standard. Values can be paired and grouped into sets. $\mathbf{V}$ is the set of values derivable from the non-terminal v in the grammar and $\mathbf{A}$ is disjoint with $\mathbf{V}$. As a consequence there is no name-passing within the calculus. The expression language is composed of data values, variable patterns, pairing, and function application. The meaning of function symbols $\mathtt{f} \in \mathbf{A}$ is defined via a total Turing computable function apply : $\mathbf{A} \times \mathbf{V} \rightarrow \mathbf{V}$ that assigns meaning to function symbols $\mathtt{f} \in \mathbf{A}$.

Guarded processes. A message e is sent via the synchronous channel c by $\overline{c}\langle e \rangle.P$ with continuation P. Message reception on channel c is written $c(X).P$. If a message v is sent on c then $c(X).P$ becomes P with all instances of variables in X instantiated with values from v. Pattern X must be linear. Process $\mathcal{P}(\!|k|\!).P$ contacts a *perfect failure detector* and may only proceed as P when location k is detected to be dead. Process $\mathcal{S}(\!|k|\!).P$ contacts an *imperfect failure detector* and may proceed as P when location k is suspected to have crashed. Since failure

DATA VALUES **V**

$$v \quad ::= \bot, 0, 1, 2, 3, \ldots \ \mid\ (v, v) \ \mid\ \{v, \ldots, v\}$$

VARIABLE PATTERN

$$X \quad ::= x \ \mid\ (X, X), \quad \text{with } x \in \mathbf{A}$$

EXPRESSIONS

$$e \quad ::= v \ \mid\ X \ \mid\ (e, e) \ \mid\ \mathtt{f}(e), \quad \text{with } \mathtt{f} \in \mathbf{A}$$

GUARDED PROCESSES **G**

$$\begin{aligned} G \quad ::= {} & \mathbf{0} \ \mid\ \overline{c}\langle e\rangle.P \ \mid\ c(X).P \ \mid\ \mathcal{S}(\!\vert k\vert\!).P \ \mid\ \mathcal{P}(\!\vert k\vert\!).P \\ \mid {} & G + G \ \mid\ \textbf{if } e \textbf{ then } G \textbf{ else } G \end{aligned}$$

PROCESSES **P**

$$P, Q \ ::= \tau.P \ \mid\ G \ \mid\ K(e) \ \mid\ P \parallel P \ \mid\ P \setminus a$$

NETWORKS **N**

$$M, N \ ::= \mathbf{0} \ \mid\ \ell\,[P] \ \mid\ N \parallel N \ \mid\ N \setminus a$$

PROCESS EQUATIONS

$$D \stackrel{\text{def}}{=} \{K_j(X) = P_j\}_{j \in J} \text{ a finite set of process definitions}$$

Table 1. Syntax

detection is unreliable in this case, the process might incorrectly suspect location k and proceed as P, even though k is actually live. Guarded choice $G + G'$ is the choice between guarded processes G or G'. Branching **if** e **then** G **else** G' evolves to G if e evaluates to an integer greater than zero, otherwise to G'.

Processes. The process $\mathbf{0}$ models inaction; process $\tau.P$ can perform a silent transition and become P. Parallel composition $P \parallel P'$ runs processes P and P' in parallel. Parameterized process constants have the form $K(X)$; are defined w.r.t. to a finite set of process equations D of the form $\{K_j(X) \stackrel{\text{def}}{=} P_j\}_{j \in J}$.

Networks. The network $\star\,[P]$ is a process running at a location $\star$; it has the property that it can never crash. The intention is to use this location for wrapper code. The location $\ell\,[P]\,, \ell \neq \star$ may however fail. An action a may be restricted to N by $N \setminus a$. Networks can be put in parallel $N \parallel N$; we write $\prod_\phi N$ for the parallel composition of a finite set of networks satisfying the logical predicate ϕ.

The substitution of value v for a variable pattern X in expression e or process P is written $e\{v/X\}$ and $P\{v/X\}$ respectively. The operator $\mathrm{fn}(\cdot)$ defined on processes and networks is defined as usual. Notice that only data values can be substituted for names and that all variables of the pattern X must be free in P. We write $\overline{c}\langle e\rangle$ for $\overline{c}\langle e\rangle.\mathbf{0}$ and $c.P$ for $c(x).P$, $x \notin \mathrm{fn}(P)$ and $\overline{c}$ for $\overline{c}\langle\bot\rangle$. Restriction is generalized to sets of names in the obvious way. Lists are defined via right-recursive pairing and we write **let** $X = e$ **in** P for the local binding of X to e in P, formally defined by a process constant $(K(X) \stackrel{\text{def}}{=} P) \in D$. Finally define $a@i(x).P$ to denote $a(x).P + \mathcal{S}(\!\vert i\vert\!).P\{\bot/x\}$, $x \in \mathrm{fn}(P)$ meaning: either receive a value on channel a or suspect location i.

Some notational conventions: $\mathcal{R}_1\mathcal{R}_2$ is the composition of relations $\mathcal{R}_1$ and $\mathcal{R}_2$; $\mathcal{R}^*$ is the transitive closure of a relation $\mathcal{R}$. $|M|$ is the cardinality of the finite multiset M. We occasionally omit binders e.g. if $(x, y) \in S$ where S is a set and y is unused we write $(x, \cdot) \in S$.

2.2 Semantics

The semantics (see Table 2) of our calculus is mostly standard. It is based on configurations consisting of a book-keeping environment and process networks. The terminology of *trusted immortals* was introduced in [NF03,NFM03] to support a simple and direct definition of the failure detector properties of [CT96]. The essence is that the process $\mathsf{ti} \in \mathcal{L}oc$, $\mathsf{ti} \neq \star$ can neither crash (*immortal*) nor be suspected by any other process (*trusted*).

Definition 1 (Configurations). *Configurations C have either of the two forms $(\mathcal{L}, n) \rhd M$ or $(\mathcal{L}, n) \rhd_{\mathsf{ti}} M$, where $\mathcal{L} \subseteq \mathcal{L}oc$ is a finite set of locations, $n \in \mathbb{N}$ and M is a network. We define $\mathbf{C}$ as the set of all configurations.*

We define the projection dead$(\cdot)$ by dead$((\mathcal{L}, n)) = \mathcal{L}oc \setminus \mathcal{L}$ and a predicate live$(\cdot, \cdot)$ in the following way: live$(\star, \Gamma)$ is true for all Γ; live$(\ell, (\mathcal{L}, n))$ is true if $\ell \in \mathcal{L}$. Let $[\![e]\!]$ denote the evaluation of expression e, defined in the standard way.

Definition 2 (Evaluation of networks). *Let $>$ be the evaluation relation defined on configurations (assuming live(ℓ, Γ) everywhere), closed under restriction, parallel composition, reflexivity, transitivity and the following rules:*

$$\begin{array}{ll}
\Gamma \rhd_{\mathsf{ti}} \ell\,[\overline{c}\langle e\rangle.P] & > \Gamma \rhd_{\mathsf{ti}} \ell\,[\overline{c}\langle[\![e]\!]\rangle.P] \\
\Gamma \rhd_{\mathsf{ti}} \ell\,[K(e)] & > \Gamma \rhd_{\mathsf{ti}} \ell\,[P\{[\![e]\!]/X\}]\,, \quad (K(X) \stackrel{\text{def}}{=} P) \in D \\
\Gamma \rhd_{\mathsf{ti}} \ell\,[\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q] & > \Gamma \rhd_{\mathsf{ti}} \ell\,[P]\,, \quad [\![e]\!] > 0 \\
\Gamma \rhd_{\mathsf{ti}} \ell\,[\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q] & > \Gamma \rhd_{\mathsf{ti}} \ell\,[Q]\,, \quad [\![e]\!] = 0.
\end{array}$$

Definition 3. *Structural congruence $\equiv$ is the least equivalence relation defined on configurations, satisfying commutative monoid laws for $(\mathbf{N}, |, \mathbf{0})$, closed under restriction and parallel composition and the rules:*

$$\begin{array}{ll}
\textit{(Nil)}\ \Gamma \rhd_{\mathsf{ti}} \ell\,[\mathbf{0}] \equiv \Gamma \rhd_{\mathsf{ti}} \mathbf{0} & \textit{(New)}\ \Gamma \rhd_{\mathsf{ti}} \ell\,[P \setminus a] \equiv \Gamma \rhd_{\mathsf{ti}} \ell\,[P] \setminus a \\
\multicolumn{2}{l}{\textit{(Location)}\ \Gamma \rhd_{\mathsf{ti}} \ell\,[P \parallel Q] \equiv \Gamma \rhd_{\mathsf{ti}} \ell\,[P] \parallel \ell\,[Q]} \\
\multicolumn{2}{l}{\textit{(Scope)}\ \Gamma \rhd_{\mathsf{ti}} M \parallel (N \setminus a) \equiv \Gamma \rhd_{\mathsf{ti}} (M \parallel N) \setminus a, \quad a \notin \mathrm{fn}(M)}
\end{array}$$

Let $\Rrightarrow$ denote the relation $>\equiv$.
We write $C \Rrightarrow^{\mathsf{T}} C'$, if $C \Rrightarrow^* C'$ and $\nexists C'' \not\equiv C' : C' > C''$.

Actions $\alpha \in$ Act are of the form $\alpha ::= \tau \mid \overline{c}v \mid cv$. The transition relation $\longrightarrow \subseteq \mathbf{C} \times \mathrm{Act} \times \mathbf{C}$ is the smallest relation generated by the rules of Table 2. Rule (TI) non-deterministically selects a trusted immortal. It is the rule that must be applied initially; this is necessary in interplay with (Susp) (see below). Rule (Stop) stops a live process from running if the total number of allowed failures is not zero. Rule (PSusp) models perfect failure detection. Rule (Susp) models

$$\text{(TI)}\ \frac{\mathsf{ti} \in \mathcal{L} \setminus \{\star\}}{(\mathcal{L}, n) \triangleright M \xrightarrow{\tau} (\mathcal{L}, n) \triangleright_{\mathsf{ti}} M} \qquad \text{(Stop)}\ \frac{\ell \neq \mathsf{ti} \wedge \ell \in \mathcal{L}}{(\mathcal{L}, n{+}1) \triangleright_{\mathsf{ti}} M \xrightarrow{\tau} (\mathcal{L}\backslash\{\ell\}, n) \triangleright_{\mathsf{ti}} M}$$

$$\text{(PSusp)}\ \frac{\text{live}(\ell, \mathit{\Gamma}) \wedge \neg\text{live}(k, \mathit{\Gamma})}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [\mathcal{P}(\!|k|\!).P] \xrightarrow{\tau} \mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [P]} \qquad \text{(Susp)}\ \frac{\text{live}(\ell, \mathit{\Gamma}) \wedge k \neq \mathsf{ti} \wedge k \neq \ell}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [\mathcal{S}(\!|k|\!).P] \xrightarrow{\tau} \mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [P]}$$

$$\text{(Tau)}\ \frac{\text{live}(\ell, \mathit{\Gamma})}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell[\tau.P] \xrightarrow{\tau} \mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell[P]} \qquad \text{(SumL)}\ \frac{\text{live}(l, \mathit{\Gamma}) \wedge \mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [G_1] \xrightarrow{\alpha} \mathit{\Gamma}' \triangleright_{\mathsf{ti}} \ell\, [P]}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [G_1 + G_2] \xrightarrow{\alpha} \mathit{\Gamma}' \triangleright_{\mathsf{ti}} \ell\, [P]}$$

$$\text{(Par)}\ \frac{\mathit{\Gamma} \triangleright_{\mathsf{ti}} M \xrightarrow{\alpha} \mathit{\Gamma}' \triangleright_{\mathsf{ti}} M'}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} M \parallel N \xrightarrow{\alpha} \mathit{\Gamma}' \triangleright_{\mathsf{ti}} M' \parallel N} \qquad \text{(SumR)}\ \frac{\text{live}(l, \mathit{\Gamma}) \wedge \mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [G_2] \xrightarrow{\alpha} \mathit{\Gamma}' \triangleright_{\mathsf{ti}} \ell\, [P]}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [G_1 + G_2] \xrightarrow{\alpha} \mathit{\Gamma}' \triangleright_{\mathsf{ti}} \ell\, [P]}$$

$$\text{(Snd)}\ \frac{\text{live}(\ell, \mathit{\Gamma})}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [\overline{c}\langle v\rangle] \xrightarrow{\overline{c}v} \mathit{\Gamma} \triangleright_{\mathsf{ti}} \mathbf{0}} \qquad \text{(Rcv)}\ \frac{\text{live}(\ell, \mathit{\Gamma})}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [c(X).P] \xrightarrow{cv} \mathit{\Gamma} \triangleright_{\mathsf{ti}} \ell\, [P\{v/X\}]}$$

$$\text{(Com)}\ \frac{\mathit{\Gamma} \triangleright_{\mathsf{ti}} M \xrightarrow{\alpha} \mathit{\Gamma} \triangleright_{\mathsf{ti}} M' \qquad \mathit{\Gamma} \triangleright_{\mathsf{ti}} N \xrightarrow{\overline{\alpha}} \mathit{\Gamma} \triangleright_{\mathsf{ti}} N'}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} M \parallel N \xrightarrow{\tau} \mathit{\Gamma} \triangleright_{\mathsf{ti}} M' \parallel N'},\quad \alpha, \overline{\alpha} \neq \tau$$

$$\text{(Red)}\ \frac{C \Rrightarrow C_1 \xrightarrow{\alpha} C_2 \Rrightarrow s'}{C \xrightarrow{\alpha} C'} \qquad \text{(Res)}\ \frac{\mathit{\Gamma} \triangleright_{\mathsf{ti}} M \xrightarrow{\alpha} \mathit{\Gamma}' \triangleright_{\mathsf{ti}} M'}{\mathit{\Gamma} \triangleright_{\mathsf{ti}} M \setminus a \xrightarrow{\alpha} \mathit{\Gamma}' \triangleright_{\mathsf{ti}} M' \setminus a},\quad \alpha \neq \overline{a}v, av$$

Table 2. Structural Operational Semantics

imperfect failure detection: processes never suspect themselves, nor the trusted immortal; every other process may be suspected at any time (see [CT96,NF03]). Rules for communication, sum and parallel composition are all standard. Rule (Red) describes the one way reduction of terms using value evaluations.

On the set of configurations, we define weak bisimilarity "up to". For this, let $\Longrightarrow \overset{\text{def}}{=} \xrightarrow{\tau}{}^{*}$. Then, $C \overset{\hat{\alpha}}{\Longrightarrow} C'$ is $C \Longrightarrow C'$, if $\alpha = \tau$, otherwise $C \Longrightarrow \xrightarrow{\alpha} \Longrightarrow C'$.

Definition 4. *Let $\mathcal{U}$ and $\mathcal{R}$ be binary relations over* $\mathbf{C}$. *We call $\mathcal{R}$ a* weak bisimulation up to $\mathcal{U}$ *if, whenever $C_1\ \mathcal{R}\ C_2$ then*

- *if $C_1 \xrightarrow{\alpha} C_1'$ then there is C_2' with $C_2 \overset{\hat{\alpha}}{\Longrightarrow} C_2'$ and $C_1'\ (\mathcal{URU})\ C_2'$.*
- *if $C_2 \xrightarrow{\alpha} C_2'$ then there is C_1' with $C_1 \overset{\hat{\alpha}}{\Longrightarrow} C_1'$ and $C_1'\ (\mathcal{URU})\ C_2'$.*

Two configurations C_1 and C_2 are said to be weakly bisimilar up to $\mathcal{U}$, written $C_1 \approx_{\mathcal{U}} C_2$, if there is a weak bisimulation (up to $\mathcal{U}$) $\mathcal{R}$ such that $C_1\ \mathcal{R}\ C_2$.

If $\mathcal{U}$ is the identity, then we get the standard bisimilarity $\approx$. If $\mathcal{U} = \equiv$, then we get a well-known proof technique for the standard bisimilarity.

2.3 Proof Methods and Methodology

Referring to the Introduction, the environment $\mathit{\Gamma}_k$ would be represented in our calculus as $(\mathcal{L}, k)$ for some $\mathcal{L} \subseteq \mathcal{L}oc$; likewise $(\mathcal{L}, 0)$ represents a (from now on) failure-free environment. Francalanza and Hennessy [FH07] managed to set up wrapper codes (one for each property to prove) such that *SimpleSpecification* boiled down to the trivial process $\overline{ok}$ running at the immortal location $\star$, the location of the wrapper code. The two equations in their methodology are then:

$$(\mathcal{L}oc, 0) \triangleright (Sys \parallel Wrapper) \setminus R \approx (\mathcal{L}oc, 0) \triangleright \star[\overline{ok}] \tag{1}$$

$$(\mathcal{L}oc, 0) \triangleright (Sys \parallel Wrapper) \setminus R \approx (\mathcal{L}oc, n{-}1) \triangleright (Sys \parallel Wrapper) \setminus R \tag{2}$$

Using transitivity of weak bisimilarity, they may be composed into:

$$(\mathcal{L}oc, n{-}1) \triangleright (Sys \parallel Wrapper) \setminus R \approx (\mathcal{L}oc, 0) \triangleright \star[\overline{ok}] \tag{3}$$

In the context of the chosen case study of [FH07], proving Equation 1 and 2 was easier than proving Equation 3 directly. That context was mainly corresponding to our calculus—except that only perfect failure detection was around. The difference, though, is crucial. With perfect failure detectors ($\mathcal{P}$), there is a gain when the correctness proof is split, as showed above. The proof of basic correctness (i.e., of Equation 1) is much simpler, because all its sub-expressions of the form $\mathcal{P}(\!|k|\!).P + Q$ are then equivalent to Q: no crash failures may occur, which means that no suspicion can be carried out at all. The proof of basic correctness hence eliminates all code after $\mathcal{P}(\!|k|\!)$ prefixes for any $k \in \mathcal{L}$. With imperfect failure detectors ($\mathcal{S}$), this is no longer the case. Expressions of the form $\mathcal{S}(\!|i|\!).P + Q$ cannot simply be rewritten to Q since the failure detector can make mistakes, even if no process crashes may occur! This has the implication that basic correctness (i.e., Equation 1) is hard to prove. We claim that, in the context of imperfect failure detectors, proving Equation 1 is even just as hard as proving Equation 3. So, in the remainder of this paper, we thus tackle Equation 3 directly for our case study.

3 Applying the Methodology to the Case Study

Distributed consensus is the following well-known problem: a fixed number n of agents each initially propose a value v_i, $1 \leq i \leq n$; then, eventually, the agents must agree on a common value $v_i \in \{v_1, \ldots, v_n\}$. The precise specification of the problem comprises three properties with temporal logic flavor: *Termination:* Every live process eventually decides some value. *Agreement:* No two processes decide differently. *Validity:* If a process decides value v, then v was proposed by some process. Table 3 presents an algorithm by Chandra and Toueg [CT96] that is supposed to solve Distributed Consensus in the context of failure detector $\mathcal{S}$.

Definition 5 (Vectors). *A n-vector is a map from set $\{1, \ldots, n\}$ to set $\mathbf{V}$. Let $\widetilde{\bot}$ denote the n-vector $(\bot, \bot, \ldots, \bot)$. Define an order $\leq$ on n-vectors by $V \leq V'$ if for every $\forall 1 \leq i \leq n : V(i) = \bot \vee V(i) = V'(i)$. We read $V \leq V$ as: V' holds at least the knowledge of V.*

The algorithm proceeds in three phases, during which it manipulates two particular vectors of each process. The vector V_p holds the current knowledge of agent p (the *knowledge vector*). If $V_p(i) = v$ then agent p knows that agent i proposed value v. The vector Δ_p is used to relay knowledge from the previous

```
1: **Pseudo code for agent** $p$
2: $V_p \leftarrow \widetilde{\bot},\ V_p(p) \leftarrow v_p$
3: $\Delta_p \leftarrow V_p,\ \ M_p \leftarrow \emptyset$
4:
5: **Phase 1:**
6: **for all** $r_p \leftarrow 1$ to $n{-}1$ **do**
7:    **send** $\mathsf{P1}(p, r_p, \Delta_p)$ to all
8:    $\Delta_p \leftarrow \widetilde{\bot}$
9:    **block until**
10:      **for all** $1 \leq q \leq n$
11:         **receive** $m = \mathsf{P1}(q, r_p, \Delta)$
12:         $M_p \leftarrow M_p \cup \{m\}$
13:         **or** suspect $\mathcal{S}(|q|)$
14:    **for all** $q \leftarrow 1$ to n **do**
15:      **if** $V_p(q) = \bot$ **and** $\exists \Delta' \in M_p$
16:      with $\Delta'(q) \neq \bot$ **then**
17:            $V_p(q) \leftarrow \Delta'(q)$
18:            $\Delta_p(q) \leftarrow \Delta'(q)$
19:
20: **Phase 2:**
21: **send** $\mathsf{P2}(V_p)$ to all
22: **block until**
23:    **for all** $1 \leq q \leq n$ **do**
24:         **receive** $m = \mathsf{P2}(V)$
25:         $M_p \leftarrow M_p \cup \{m\}$
26:         **or** suspect $\mathcal{S}(|q|)$
27: **for all** $q \leftarrow 1$ to n **do**
28:    **if** $\exists V' \in M_p : V'(q) = \bot$
29:       **then** $V_p(q) \leftarrow \bot$
30:
31: **Phase 3:**
32: decide = min $\{q \mid V_p(q) \neq \bot\}$
```

Table 3. Distributed Consensus [CT96]

round (the *relay vector*). Each agent has a round variable r which allows agents to order messages. Variable q is used to iterate through all agent $1 \ldots n$. Variable M_p is a multiset which serves as a store for all received messages. Initially, every agent p knows its own value, i.e. $V_p(p) = v_p$, Δ_p equals V_p and store M_p is empty.

Phase 1 — obtaining knowledge. The agents broadcast and update their knowledge during $n{-}1$ rounds. When a received message contains a previously unknown value then both knowledge and relay vector will be updated. Newly learned values are relayed once because of the boolean predicate in line 15 and the fact that the relay vector is reset in the start of each round. It can be proven that every agent p that completes Phase 1 at least has the same knowledge as ti (the trusted immortal, that is the live and never wrongly suspected agent).

Phase 2 — correcting knowledge. If $V_i(j) = \bot$ for some agent i and j then either agent i has suspected agent j to have crashed or j stopped before sending messages to i. Such "not-known" values are distributed among all the participants. An agent k that receives knowledge vector V_i corrects coordinate j to $\bot$, i.e. $V_k(j) = \bot$. Destruction of knowledge in this fashion happens in line 29. It can be proved that every agent p that reaches the end of Phase 2 has the same $\bot$'s as ti. As an effect it holds that $V_{\mathsf{ti}} = V_p$ for any such p at the end of Phase 2.

Phase 3 — selecting the final value. The two phases above ensure that the knowledge vector of every live agent is equal to V_{ti}. The first non-zero value in the knowledge vector is chosen. Since process ti knows it's own value i.e. $V_{\mathsf{ti}}(\mathsf{ti}) = v_{\mathsf{ti}}$ this value cannot be $\bot$. So, every agent will agree on some number in the end.

3.1 Encoding the Case Study

In Table 5, we formulate system *Sys* in our calculus with a formalization of (i) the behavior of each agent, (ii) the communication between agents and (iii) failure

$$\begin{array}{lcl}
\text{apply}(\texttt{data}, (r, M)) & \overset{\text{def}}{=} & \{\varDelta \mid (\varDelta, r', i) \in M \wedge r = r'\} \\
\text{apply}(\texttt{data}, M) & \overset{\text{def}}{=} & \{V \mid (V, i) \in M\}. \\
\text{apply}(\texttt{senders}, (r, M)) & \overset{\text{def}}{=} & \{i \mid (\varDelta, r', i) \in M \wedge r = r'\} \\
\text{apply}(\texttt{senders}, M) & \overset{\text{def}}{=} & \{i \mid (V, i) \in M\}. \\
\text{apply}(\texttt{update}, (r, M, V, W)) & \overset{\text{def}}{=} & W' \\
\quad \text{where } W'(j) & = & \begin{cases} \varDelta^r & , \varDelta \in \texttt{data}(r, M) \text{ and} \\ & V(j) = \bot^0 \neq \varDelta(j), \\ W(j) & , \text{otherwise} \end{cases} \\
\text{apply}(\texttt{correct}, (M, V)) & \overset{\text{def}}{=} & W \\
\quad \text{where } W(i) & = & \begin{cases} \bot & , V' \in \texttt{data}(M) \text{ and} \\ & V'(i) = \bot^0 \\ V(i) & , \text{otherwise} \end{cases}
\end{array}$$

Table 4. Auxiliary function declarations

detection. We identify agents via numbers, i.e. $\mathcal{L}oc \overset{\text{def}}{=} \{\star, 1, \ldots, n\}$. Agents and each phase of the algorithm are modeled via parameterized process constants.

To ease the correctness proof we need a way of expressing that a value v_i was learned in round r_i. Without changing the algorithm we extend the knowledge vector with round numbers: $(v_1^{r_1}, \ldots, v_n^{r_n})$ where v^r is shorthand for (v, r). We still compare vectors $V \leq V'$ by comparing the unannotated versions of V and V'. The *initial knowledge vector* I_i^0 for agent i is a map defined by $I_i^0(i) = v_i^0$ and $I_i^0(j) = \bot^0$ for $j \neq i$. The *initial relay vector* $I_i(i)$ for agent i is a map defined by $I_i(i) = v_i$ and $I_i(j) = \bot$ for $j \neq i$.

We define functions for the internal computation at each agent. First there are simple functions supporting primitive operations such as multiset manipulation, manipulation of vectors, and operations related to integers such as comparison and addition. The maximum of a finite multiset M of numbers is written $\max(M)$. Function $\texttt{getfst}(V)$ returns first non-zero component of V. If no such entry exists the value $\bot$ is returned. In Table 4 we define more advanced functions. The functions `data` and `senders` are used to project information on sent data and sender identities from a given multiset of messages M. We may call the function with a round number r as filter. The function `update` updates vector W with respect to current knowledge V, round number r and received messages M corresponding to Phase 1, lines 14–18 in Table 3. Function `correct` corrects the knowledge vector V with respect to received messages M, corresponding to Phase 2 in lines 27–29 in Table 3. The process constant $\mathsf{P1}_p(r, V, \varDelta, M)$ corresponds to an agent p in Phase 1 which broadcasts its current knowledge and waits for incoming messages by process constant $\mathsf{C1}_p(r, V, M)$ (that defines the gathering of answers and updates of knowledge in Phase 1). Symbol r is the current round number, V is the current knowledge vector and $\varDelta$ is the current communication vector and M is the (possibly empty) multiset of received messages for round r. Phase 2 is modeled by $\mathsf{P2}_p(V)$ and $\mathsf{C2}_p(V, M)$ corresponds to

```
$Sys \overset{\text{def}}{=}$
    $\left(\prod_{i=1}^{n} i\left[\mathsf{P1}_i(1, I_i^0, I_i, \emptyset)\right]\right)$

$\mathsf{P1}_p(r, V, \Delta, M) \overset{\text{def}}{=}$
    **if** $(r < n)$ **then**
        $\prod_{1 \leq i \leq n} \overline{a_{p,i,r}}\langle \Delta \rangle \parallel \mathsf{C1}_p(r, V, M)$
     **else** $\mathsf{P2}_p(V, M)$

$\mathsf{C1}_p(r, V, M) \overset{\text{def}}{=}$
    **let** $i = 1 + \max(\mathtt{senders}(r, M)))$ **in**
      **if** $i \leq n$ **then**
        $a_{i,p,r}@i(\Delta).$
          $\mathsf{C1}_p\Big(r, \mathtt{update}(r, M, V, V),$
                                    $M + (\Delta, r, i)\Big)$
      **else**
        $\mathsf{P1}_p\Big(r + 1, V,$
          $\mathtt{update}(r, M, V, \widetilde{\bot}), M\Big)$

$\mathsf{P2}_p(V, M) \overset{\text{def}}{=}$
    $\prod_{1 \leq i \leq n} \overline{b_{p,i}}\langle V \rangle \parallel \mathsf{C2}_p(V, M)$

$\mathsf{C2}_p(V, M) \overset{\text{def}}{=}$
    **let** $i = 1 + \max(\mathtt{senders}(M))$ **then**
      **if** $i \leq n$ **then**
        $b_{i,p}@i(V').\mathsf{C2}_p(V, M + (V', i))$
      **else**
        $\mathsf{P3}_p(\mathtt{correct}(M, V), M)$

$\mathsf{P3}_p(V, M) \overset{\text{def}}{=}$
    $\overline{c_p}\langle \mathtt{getfst}(V), V, M \rangle$

$Wrap(i, v) \overset{\text{def}}{=}$
    **if** $(i \leq n)$ **then**
      $\mathcal{P}(\!|i|\!).\ Wrap(i + 1, v)\ +$
      $c_i(v', V, M).$
        **if** $(v{=}\bot \vee v{==}v')$ **then**
          $Wrap(i + 1, v')$ **else** $\mathbf{0}$
     **else if** $(i == n + 1)$ **then** $\overline{ok}$
```

Table 5. Encoding of the algorithm of Table 3

the gathering of information in Phase 2 analogously. Phase 3 is modeled by the process constant $\mathsf{P3}_p(V)$.

Constant $Wrap(i, v)$ is the wrapper code that checks for agreement. It collects all the decision values agent by agent and checks that they agree. If they all agree, then $\overline{ok}$ is released. Otherwise the checker becomes $\mathbf{0}$. The wrap code has to use perfect failure detectors since unreliable failure detectors may cause incorrect answers. For convenience, let $R \overset{\text{def}}{=} \{a_{i,j,k}, b_{i,j}, c_i\}_{1 \leq i,j,k \leq n}$.

Trying to formalize some intuitions about the algorithm, we quickly get to the point where we need to formulate properties that refer to the respective states of the processes, not their actions. Process calculi do not directly support this, except when we refer to the process constants—and their parameters—that we used to write down the code. To enable this kind of reasoning, we define dedicated syntactic forms that also capture the complete message space.

Definition 6. *A standard form C_ξ is a configuration of the form:*

$$\begin{aligned}
&\boxed{\Gamma} \;\triangleright_{\boxed{\mathsf{ti}}} \Bigg(\prod\nolimits_{(p,r,i) \in \boxed{\Pi_1^{\text{out}}}} p\left[\overline{a_{p,i,r}}\langle \boxed{\Delta_{p,r}} \rangle\right] \parallel \prod\nolimits_{(p,i) \in \boxed{\Pi_2^{\text{out}}}} p\left[\overline{b_{p,i}}\langle \boxed{V_p^{\mathsf{P2}}} \rangle\right] \parallel \\
&\prod\nolimits_{p \in \boxed{\Pi_3^{\text{out}}}} p\left[\overline{c_p}\langle \boxed{v_p}, \boxed{V_p^{\mathsf{P3}}}, \boxed{M_p^{\mathsf{P3}}} \rangle\right] \parallel \\
&\prod\nolimits_{(p,r) \in \boxed{\Pi_1^{\text{col}}}} p\left[\mathsf{C1}_p\left(r, \boxed{V_p^{\mathsf{P1}}}, \boxed{M_p^{\mathsf{P1}}}\right)\right] \parallel \prod\nolimits_{p \in \boxed{\Pi_2^{\text{col}}}} p\left[\mathsf{C2}_p\left(\boxed{V_p^{\mathsf{P2}}}, \boxed{M_p^{\mathsf{P2}}}\right)\right] \\
&\prod\nolimits_{c \in \text{dead}(\Gamma)} c\,[Q_c] \quad \parallel \; Wrap\left[Wrap(\boxed{j}, \boxed{w})\right] \Bigg) \setminus R
\end{aligned}$$

where dead($\mathit{\Gamma}$) *is disjoint with* $\mathit{\Pi}_1^{\mathrm{out}}$, $\mathit{\Pi}_2^{\mathrm{out}}$, $\mathit{\Pi}_3^{\mathrm{out}}$, $\mathit{\Pi}_1^{\mathrm{col}}$, $\mathit{\Pi}_2^{\mathrm{col}}$. *Parameter* ξ *is a data structure consisting of all the boxed values above. We refer to its entities "by name", ı.e., using boxed symbols. We will often write* ξ *instead of* C_ξ.

Our standard form is defined w.r.t. process constants. By the semantics, they are not necessarily fully unfolded. Since unfoldings may take place independently in different parts of terms, different process constants may be unfolded at different degrees, some too far, some too little. It requires a subtle definition to precisely relate any reachable configuration to some standard form.

Definition 7. *A configuration* C *with* $C \Rrightarrow^{\tau} C'$ has standard form C_ξ *if there exist a vector family* ξ *such that* $C_\xi \Rrightarrow C'$.

The connection of configurations to standard forms cannot be lost in transition.

Lemma 1 (Preservation of Standard Forms).
If $C \to C'$ *and* C *has a standard form, then* C' *also has a standard form.*

3.2 Weak Bisimulation Relations via Invariants

Definition 6 suggests that the reachable state space of Chandra and Touegs algorithm is reasonably complex. Agents may be in different phases, have different knowledge and different sets of relay vectors. Learning from Chandra and Toueg's proof sketch [CT96], we capture this combinatorial space via invariants.

Definition 8. *An invariant* I *is a boolean predicate defined on configurations such that* $I(C)$ *and* $C \xrightarrow{\alpha} C'$ *imply* $I(C')$ *for* $\alpha \in \{\tau, \overline{ok}\}$.

Invariants provide an abstraction from actual states to classes of states. It is this characteristic that we use when we give witness relations for our weak bisimulation relations. With the convention that $\mathit{Spec} = \mathit{Wrap}[\overline{ok}]$ we require that it accepts the initial configuration and that success eventually is reached:

$$I\Big((\mathcal{L}oc, n{-}1) \triangleright (\mathit{Sys} \parallel \mathit{Wrap}(1,\bot)) \setminus R\Big). \tag{4}$$

$$\text{If } I(\xi) \text{ then } \xi \overset{\overline{ok}}{\Longrightarrow} \mathbf{0} \tag{5}$$

That enables us to construct a witness relation $\mathcal{R} \subseteq \mathbf{C} \times \mathbf{C}$ for Equation 3 (closed under symmetry) of the form:

$$\mathcal{R} = \big\{ \big(\xi, (\mathcal{L}oc, 0) \triangleright \mathit{Wrap}[\overline{ok}]\big) \mid I(\xi) \big\} \tag{6}$$

Lemma 2. $\mathcal{R}$ *is a weak bisimulation up to* $\Rrightarrow$ *if* I *satisfies Equation 4 and 5.*

Equation 6 and the requirements to invariant I prepare for a proof of Equation 3.

Definition 9. *Let predicate* I *be the conjunction of the predicates (all defined below): control, validity, relay, receive*$_1$*, learn, preknow, receive*$_2$ *and know.*

The predicate *control* defines control criteria to the algorithm, e.g. agent p cannot be in Phase 1 and Phase 2 at the same time or agent p cannot be in two different rounds at the same time in Phase 1 etc.

Definition 10 (Control flow). *Define the predicate* $control(\xi)$ *as the conjunction of the expressions below:*

1. *if* $(p_1, \cdot) \in \boxed{\Pi_1^{\mathrm{col}}}$ *and* $p_2 \in \boxed{\Pi_2^{\mathrm{col}}}$ *and* $p_3 \in \boxed{\Pi_3^{\mathrm{out}}}$ *then* $p_1 \neq p_2$ *and* $p_1 \neq p_3$ *and* $p_2 \neq p_3$*.*
2. *if* $(p, r) \in \boxed{\Pi_1^{\mathrm{col}}}$ *and* $(p', r') \in \boxed{\Pi_1^{\mathrm{col}}}$ *and* $p = p'$ *then* $r = r'$*.*
3. $1 \leq \boxed{j} \leq n + 1$
4. $\forall r : |\mathtt{senders}(r, \boxed{M_p^{\mathsf{P1}}})| = \max\left(\mathtt{senders}\left(r, \boxed{M_p^{\mathsf{P1}}}\right)\right)$
5. $|\mathtt{senders}\left(\boxed{M_p^{\mathsf{P2}}}\right)| = \max\left(\mathtt{senders}\left(\boxed{M_p^{\mathsf{P2}}}\right)\right)$
6. *If* $\exists p : (p, \cdot) \notin \boxed{\Pi_1^{\mathrm{col}}}$ *and* $p \notin \boxed{\Pi_2^{\mathrm{col}}}$ *and* $p \notin \boxed{\Pi_3^{\mathrm{out}}}$ *then* $p < \boxed{j}$*.*

The next predicate formally describes what we mean by validity: all values in knowledge and relay vectors have been proposed by someone.

Definition 11 (Validity). *Let* U *be a vector which holds the initially proposed value by participant* i*, i.e.* $U(i) = v_i$ *for* $1 \leq i \leq n$ *and define the predicate* $validity(\xi)$ *as the conjunction of the expressions below:*

1. $\forall p : \boxed{V_p^{\mathsf{P1}}}(p) = v_p^0$*,* 2. $\forall (p, r, i) \in \boxed{\Pi_1^{\mathrm{out}}} : \boxed{\Delta_{p,r}} \leq U$
3. *If* $(p, r) \in \boxed{\Pi_1^{\mathrm{col}}}$ *or* $p \in \boxed{\Pi_2^{\mathrm{col}}}$ *or* $(p, i) \in \boxed{\Pi_2^{\mathrm{out}}}$ *or* $p \in \boxed{\Pi_3^{\mathrm{out}}}$ *then* $\boxed{V_p^{\mathsf{P}}} \leq U$ *for* $\mathsf{P} \in \{\mathsf{P1}, \mathsf{P2}, \mathsf{P3}\}$*.*
4. $\forall (\Delta, r, i) \in \boxed{M_p^{\mathsf{P1}}} \cup \boxed{M_p^{\mathsf{P2}}} \cup \boxed{M_p^{\mathsf{P3}}} : \Delta \leq U$
5. $\forall (V, i) \in \boxed{M_p^{\mathsf{P2}}} \cup \boxed{M_p^{\mathsf{P3}}} : V \leq U,$ *if* $V \neq \bot$
6. *If* $p \in \boxed{\Pi_3^{\mathrm{out}}}$ *then* $\boxed{v_p} = \mathtt{getfst}\left(\boxed{V_p^{\mathsf{P3}}}\right)$
7. $1 \leq \boxed{j} \leq n + 1 \;\wedge\; \exists i : \boxed{w} = v_i \vee \boxed{w} = \bot$*,* 8. *If* $\boxed{j} = 1$ *then* $\boxed{w} = \bot$

The next predicate says that values learned in round r are relayed in round $r+1$.

Definition 12 (Relays of knowledge, Phase 1). *Define the predicate* $relay(\xi)$ *by the following: If* $(p, \cdot) \in \boxed{\Pi_1^{\mathrm{col}}}$ *and* $\boxed{V_p^{\mathsf{P1}}}(j) = v^{r'}$*,* $v \neq \bot$ *for some* j *and* $r' \leq n - 2$ *then it holds that*

1. *if* $(p, r, \cdot) \in \boxed{\Pi_1^{\mathrm{out}}}$ *and* $r \neq r' + 1$ *then* $\boxed{\Delta_{p,r}}(j) = \bot$
2. *if* $(p, r, \cdot) \in \boxed{\Pi_1^{\mathrm{out}}}$ *and* $r = r' + 1$ *then* $\boxed{\Delta_{p,r}}(j) = v$

Knowledge propagates from ti to all live agents of the protocol.

Definition 13 (Received messages, Phase 1). *Define the predicate* $receive_1(\xi)$ *by the following:*

If (a) $(\mathsf{ti}, \cdot) \in \boxed{\Pi_1^{\mathrm{col}}}$ *and (b)* $\boxed{V_{\mathsf{ti}}^{\mathsf{P1}}}(j) = v^r \neq \bot$ *for some* j

then $0 \leq r \leq n-2$ *and* $r' = r+1$ *implies* $(\Delta, r', \mathsf{ti}) \in \boxed{M_p^{\mathsf{P1}}}$ *and* $\Delta(j) = v$.

Maybe the most important property: all agents learn from the trusted immortal.

Definition 14. *Define the predicate learn*(ξ) *by the following:*

If (a) $(\mathsf{ti}, \cdot) \in \boxed{\Pi_1^{\mathrm{col}}}$, *(b)* $\boxed{V_{\mathsf{ti}}^{\mathsf{P1}}}(j) = v^r \neq \bot$ *for some* j, *and (c)* $(p, r') \in \boxed{\Pi_1^{\mathrm{col}}}$
then (a) if $0 \leq r \leq n-2$ *and* $r' \geq r+1$ *then* $\boxed{V_p^{\mathsf{P1}}}(j) = v^{r''}$ *for some* r''.
(b) if $r = n-1$ *then* $\boxed{V_p^{\mathsf{P1}}}(j) = v^{r''}$ *for some* r''.

The property that all agents learn from the trusted immortal also holds at the beginning of Phase 2 where every agent at least has the same knowledge as ti.

Definition 15. *Define predicate preknow*(ξ) *by:* $\forall (p, \cdot) \in \boxed{\Pi_2^{\mathrm{out}}} : \boxed{V_{\mathsf{ti}}^{\mathsf{P2}}} \leq \boxed{V_p^{\mathsf{P2}}}$.

The predicate below states that all agents receive from ti in Phase 2.

Definition 16. *Define the predicate receive*$_2(\xi)$ *as follows: if all of 1.* $p \in \boxed{\Pi_2^{\mathrm{col}}}$, *2.* $i := \max(\texttt{senders}(\boxed{M_p^{\mathsf{P2}}}))$, *and 3.* $\mathsf{ti} < i$, *then* $\exists V : (V, \mathsf{ti}) \in M_p^{\mathsf{P2}}$.

The effect is that everyone has the same knowledge as ti at the end of Phase 2 (or beginning of Phase 3), which is stated in the following predicate:

Definition 17. *Define the predicate know*(ξ) *by 1.* $\forall p \in \boxed{\Pi_3^{\mathrm{out}}} : \boxed{V_{\mathsf{ti}}^{\mathsf{P3}}} = \boxed{V_p^{\mathsf{P3}}}$ *and 2. If* $p \in \boxed{\Pi_3^{\mathrm{out}}}$ *and* $0 < \boxed{j} < n+1$ *then* $\boxed{w} = \boxed{v_p}$.

The following important theorem tells that I is an invariant.

Theorem 1. *I is an invariant which satisfies Equation 4.*

The proof that "$I(\xi)$ implies that $\xi \overset{\overline{ok}}{\Longrightarrow} \mathbf{0}$" uses a progress measure as temporal distance of any agent i to termination. Our main theorem follows directly.

Theorem 2. *The relation in Equation 6 is a weak bisimulation up to* $\Rrightarrow$ *with invariant I defined as in Definition 9.*

In summary, we have proved the required Consensus properties: Validity holds since it is part of the global invariant; Termination follows from the above-mentioned progress analysis ending up in a state where the wrapper code comes to an end; from the argument that the wrapped system is weakly bisimilar to ok, we get Agreement, obviously due to the design of the wrapper code.

4 Conclusion and Future Work

Our case study may offer several insights. The strategy, or: methodology, that worked quite nicely in our case, may be summarized as follows. The usage of a process calculus helps to keep a tight connection to the algorithm's code, so our proofs are meaningful. The formulation of the proof goal by wrappers is, although not a new idea, very useful in the context of fault tolerance. A novelty in our approach was the combination of imperfect failure detectors (as necessarily assumed by the case study) and perfect failure detectors (as idealistically assumed to have the wrapper code function properly). The introduction of standard forms to manage the complexity of state spaces is not a new idea either; it has mostly been used implicitly and often only in toy examples, but it seems to scale quite well. The reason we propose to turn standard forms explicitly into a method is that they provide a well-suited means to express the typical assertional state-based proof knowledge as invariants. Again, the mere use of invariants is not at all a new idea. However, their systematic integration within the bisimulation method seems novel.

Future work on this case study may involve confluence-oriented proof methods, as employed in [FH07,PM06], and to investigate in what flavor they appear in our invariant-oriented method. Likewise, it might support our claims to also carry out our proof case study on a non-wrapped equation, that is, to contrast our approach of this paper with a bisimulation-based proof of an equation without hiding that much external behavior in wrapper code.

References

[BH00] Martin Berger and Kohei Honda. The two-phase commitment protocol in an extended pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 39(1), 2000.

[CT96] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), 1996.

[FGR04] Wan Fokkink, Jan Friso Groote, and Michel Reniers. Process algebra needs proof methodology. *EATCS Bulletin*, 82:109–125, February 2004.

[FH07] Adrian Francalanza and Matthew Hennessy. A fault tolerance bisimulation proof for consensus. In *Proceedings of ESOP*, 2007.

[Fou98] Cédric Fournet. *The Join-Calculus: A Calculus for Distributed Mobile Programming.* PhD thesis, École Polytechnique, Paris, France, 1998.

[Hen07] Matthew Hennessy. *A Distributed Pi-Calculus.* Cambridge University Press, 2007. ISBN: 0-521-87330-4.

[Küh08] Morten Kühnrich. Formal model-driven design of distributed algorithms. Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. November 2008.

[Lyn96] Nancy Lynch. *Distributed Algorithms.* Kaufmann Publishers, 1996.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[NF03] U. Nestmann and R. Fuzzati. Unreliable failure detectors via operational semantics. In Vijay A. Saraswat, editor, *ASIAN*, volume 2896 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2003.

[NFM03] U. Nestmann, R. Fuzzati, and M. Merro. Modeling consensus in a process calculus. In *Proceedings of CONCUR*, 2003.

[PM06] Anna Philippou and George Michael. Verification techniques for distributed algorithms. In *Proceedings of OPODIS 2006*, volume 4305. 2006.

[SV00] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. In *CSFW*, pages 269–284, 2000.

[TS08] Tatsuhiro Tsuchiya and André Schiper. Using bounded model checking to verify consensus algorithms. LNCS, 2008.