




Store Locally, Prove Globally

Nadine Karsten and Uwe Nestmann^(✉) 

Technische Universität Berlin, Berlin, Germany
`{n.karsten,uwe.nestmann}@tu-berlin.de`

Abstract. The use of message-passing process calculi for the verification of distributed algorithms requires support for state-based reasoning that goes beyond their traditional action-based style: knowledge about (local) states is at best provided implicitly. Therefore, we propose a distributed process calculus with locations, the units of distribution, which we equip with explicit state information in the form of memories. On top, we provide a simple formal model for location failure and failure detection such that we can deal with the verification of *fault-tolerant* distributed algorithms. We exhibit the use of our calculus by formalizing a simple algorithm to solve Distributed Consensus and prove its correctness. The proof exploits global invariants by direct access to the local memories.

1 Introduction

Distributed Algorithms. Traditionally [17], distributed algorithms are often described by means of pseudo code for its local processes: sequences of statements may manipulate local variables or trigger the exchange of messages with other participating processes. The following code [23, 9] describes the intended behavior of a single so-called *participant* i (one out of n) which is meant to solve the problem of Distributed Consensus [17] in a system where processes may fail.

```
 $x_i := \text{input};$   
for  $r := 1$  to  $n$  do { if  $r = i$  then broadcast  $x_i$ ;  
                    if  $\text{alive}(p_r)$  then  $x_i := \text{input from broadcast}$  };  
output  $x_i$ ;
```

An understanding of such a distributed algorithm requires to precisely fix the underlying assumptions of the system model, e.g., the meaning of send (broadcast) and receive (input) actions in the context of failures. In the above algorithm, an essential ingredient is the alive-test whose passing is subject to subtle guarantees. In the following, we explain the intuition behind alive-tests in the context of fault tolerance and the correctness of Distributed Consensus in more detail.

Fault Tolerance. In the so-called *fail-stop* model of distributed systems, processes may fail; and when they do so, they do not recover from this state. A failed process does no longer contribute to the system evolution, i.e., it can neither

send nor receive messages. A process that does not fail in a run, is called *correct* (in that run). Failure *detection* provides processes with the permission to *suspect* other processes to *have failed* and, thus, to no longer wait for their messages to arrive. Perfect (i.e., always reliable) failure detection is not implementable in purely asynchronous systems, since it is impossible to distinguish the processes that have failed from those that are just slow. Here, Chandra and Toueg [6] proposed the concept of *unreliable* failure detection, whose degree of reliability is expressed by means of temporal constraints on runs. For example, for the above Consensus algorithm, a property called *Weak Accuracy* suffices: “Some correct process is never suspected by *any* (correct) process”.

Correctness. Specifications for distributed algorithms typically consist of properties of some temporal logic flavor that capture the intended safety and liveness guarantees. For Distributed Consensus, all participants shall agree on the decision for some value, while every participant starts with a private input value as proposal. In the above algorithm, this input value is initially assigned to the local variable x_i , which may then be updated due to knowledge acquired by learning about the values kept by other participants via communication. Three temporal properties then capture in how far an algorithm works correctly. • *Validity*: Every decision must be for some initial proposal. • *Agreement*: No two correct processes decide differently. • *Termination*: Every correct process eventually decides. The verification of these properties is dominated by *state-based reasoning* techniques, often referring to global state invariants about the values that are memorized in the respective local variables x_i of every (alive) participant.

In the above example algorithm, each participant gets its turn to propose a value in the role of the *coordinator* of “its” round. In every other round, each participant is to adopt the value proposed by *that* round’s coordinator . . . unless it cannot detect that it is still “alive”. The algorithm satisfies Termination, as it runs a for-loop and never deadlocks. It satisfies Validity, as values are never invented, but only passed on. It satisfies Agreement, as (at least) in the round of the process that—by Weak Accuracy—is *never* suspected, every other process will have to adopt this proposal. Afterwards, there is no way to decide otherwise.

Using Process Calculi. Process calculi provide a wealth of proof methods and their syntactic nature allows for concise formal models that are nevertheless close to executable code in programming languages. A great variety of process calculi have been developed in the past, most of them for general purposes, some of them rather domain-specific. In the above case, the domain prompts two choices: (i) It is natural to employ *distributed* process calculi [12], where so-called *locations* represent units of distribution, possibly subject to failure. (ii) As message-passing models are prevalent for distributed systems, it is obvious to also use *message passing calculi*, as opposed to distributed process calculi that are based on the migration among and within so-called *ambients* [5].

Most of the existing process calculi (often descendants of CSP [13], CCS [19], ACP [3], or the π -calculus family [20]), however, are based on notions of action, thus essentially supporting just *action-based reasoning*. The main observations

in the above-mentioned attempts to use process calculi to verify distributed algorithms are that (i) even if action-based reasoning—often using bisimulation techniques—is employed on the outside, it still heavily relies on state-based reasoning inside to construct the required bisimulations [9], and (ii) classical process calculi do not at all support state-based reasoning. This was also the main problem in [22, 15], where the respective authors applied process calculus machinery to the specification and verification of fault-tolerant algorithms that solve Distributed Consensus. In [24], the authors propose a method to systematically (re)construct state information for the reachable global states of an example Distributed Consensus algorithm (which was formalized in a tailor-made process calculus) and to capture this information within a dedicated data structure outside of the calculus. The lesson learned from [24] was that this method is too tedious and highly error-prone; it simply did not scale. This is the motivation to, instead of reconstructing implicit state information, make it explicit from the outset and provide linguistic support and structure within the process calculus itself. In this paper, we report on some of our recent results in this endeavor.

Our Approach. We use a reasonably standard and widespread notion of memory: mappings from variables to values. In our calculus, processes are threads that are associated with its local memories. Threads may declare variables and assign the value of complex expressions to them, resulting in updates to their own memory. Threads can be defined recursively, and they may run concurrently. In a fault-tolerant scenario, locations are “named processes” such that failures can be named. Parallel processes, together with “message in transit”, form networks.

In the operational semantics of our calculi, we let transitions operate between structural equivalence classes (equipped with some convenient congruence properties) of states. In a fault-tolerant scenario, global configurations keep track of failures and their detection. Executions of failure-aware networks, and their reachable configurations, can be analyzed via induction on transition sequences.

Related Work. Next to the above-mentioned work [22, 15, 9] using process calculi, we also used a state-machine approach [14], which suffers from the fact its global view on algorithms slightly obfuscating the locality of behaviors.

There are only few other related approaches using process calculi. In several contexts, process calculi have been equipped with notions of location or locality [4, 12, 5], but there they have different meaning; in particular, locations were not equipped with memories. In calculi with reversibility (e.g., [7]), process-local memories are used to store back-tracking information, i.e., a history of steps of a process that led it to the current state, which can be exploited to undo these steps in a causally consistent manner. Closest to our approach is the work of Garavel [10] on LNT, which is a programming language in the spirit of LOTOS that was developed to be easier to use for engineers [11]. Our treatment of write-many variables, which is uncommon for most process calculi, was partly inspired by them. However, the context of LOTOS/LNT is different from our distributed world, as it mainly addresses concurrent algorithms without support for fault-tolerance. More detailed comparisons are found later on in Sect. 3.

Structure and Contributions. In Sect. 2, we introduce memories and expression evaluation. In Sect. 3, we define syntax and semantics of a novel calculus of distributed processes that dispose of local memories. We provide a reasonably simple operational semantics for this calculus and discuss the impact of α -conversion arising from the role of memories as binders for variables. In Sect. 4, we equip the calculus with awareness for locations and failures, which allows for completely new ways to model messages in transit and to deal with failure suspicions. In Sect. 5, we demonstrate the use of the calculus in a case study, where the advantage of direct access to local memories of processes is apparent. In Sect. 6, we summarize our contributions and conclude with a glimpse on future work.

2 Memories

We employ the widely-used idea that *states*, in the simplest possible way, are just variable assignments, which are often also called *memories*. This follows the tradition of research on state-based reasoning (see the ABZ conference series [2]).

We assume a set \mathbb{V} of *values* v , for example, booleans or natural numbers. We also assume a countably infinite set \mathcal{X} of *variables* x . A *memory* is modeled as a total function $M : \mathcal{X} \rightarrow \mathbb{V} \cup \{\top, \perp\}$, by which variables may be associated with values or otherwise have the status of being just *initialized* (\top) or *undefined* (\perp). The set $\text{dom}(M) \triangleq \{x \in \mathcal{X} \mid M(x) \neq \perp\}$ denotes all variables *defined* in M . Accordingly, M_\perp denotes an initial memory, thus without any defined variables.

By their mutable nature, memories may be *updated*, which can be defined as follows: a memory $M\langle x \mapsto w \rangle$, where x is updated to map to $w \in \mathbb{V} \cup \{\top\}$, behaves just like memory M unless we access the entry of the updated variable x :

$$M\langle x \mapsto w \rangle(y) \triangleq \begin{cases} w & \text{if } x = y \\ M(y) & \text{if } x \neq y \end{cases}$$

Note that also the cases with $M(y) \in \{\top, \perp\}$ are properly covered.

We assume a set \mathcal{E} of *expressions* e with $\mathbb{V} \cup \mathcal{X} \subseteq \mathcal{E}$. One may consider arbitrarily complex expressions with vectors and function symbols, as given by:

$$e ::= v \mid x \mid (e, \dots, e) \mid f(e)$$

The intended application will decide the respective range of allowed expressions.

We define the set $\text{fv}(e)$ of (*free*) *variables of* e inductively by $\text{fv}(v) \triangleq \emptyset$, $\text{fv}(x) \triangleq \{x\}$, $\text{fv}((e_1, \dots, e_n)) \triangleq \bigcup_{i \in \{1, \dots, n\}} \text{fv}(e_i)$, and $\text{fv}(f(e)) \triangleq \text{fv}(e)$.

We assume that expressions can be “reduced” to values by terminating computations. As expressions $e \in \mathcal{E}$ may contain variables, we should *evaluate* them within the context of a memory M with $\text{fv}(e) \subseteq \text{dom}(M)$. We let the function $\text{fetch}_M : \mathcal{E} \rightarrow \mathbb{V} \cup \{\perp\}$ for memory M replace the variables in $\text{fv}(e)$ with their M -value; if a variable is only initialized, the result will yield undefined (see Definition 3 in the Appendix). To model the evaluation of expressions that include function symbols f , we assume a homomorphic function $\text{eval}(\cdot) : \mathcal{E} \cup \{\perp\} \rightarrow \mathbb{V} \cup \{\perp\}$

to be employed *after* $\text{fetch}_M(e)$ has fetched from M —if possible—current values for the variables contained in e . The obvious idea then is that eval applies the semantics of each application of the function symbol f . Thus, we define $\text{eval}_M(e) \triangleq \text{eval}(\text{fetch}_M(e))$.

3 A Distributed Process Calculus with Local Memories

As we intend to use this calculus in the context of *distributed* systems, we have to rely on a concept of distributable units. We propose to use *threads* that dispose of their own private memory, which we call *processes*, as the units of distribution. In physically distributed systems, messages take time to travel from one process to another. Therefore, the *asynchronous* variant of message passing is to be preferred, in which send and receive actions are decoupled, as they cannot happen at the same time. Causally evident, send actions must always occur strictly before their corresponding receive action, which we model via a representation of “messages in travel”. All local memory states together with all messages in travel then provide us with the global state of a system.

In this section, we fix all of the these concepts as a calculus with two-level syntax for threads and (networks of distributed) processes.

In our calculus, the standard issues of bindings of variables as well as the notion of α -conversion inevitably pop up and get proper treatment. Note in advance that this treatment is just necessary in order to provide a sound operational semantics for the calculus. When it comes to the use of the calculus for verification, we better avoid the need for α -conversion *during* executions.

Syntax. We assume the set \mathcal{X} of variables, the set \mathbb{V} of values, and the set \mathcal{E} of expressions with $\mathcal{X} \cup \mathbb{V} \subseteq \mathcal{E}$. Let $\mathbb{B} = \{\text{t}, \text{f}\}$ be the set of booleans with $\mathbb{B} \subseteq \mathbb{V}$. Let $\mathbb{C} \subseteq \mathbb{V}$ denote the set of available channels where $c \in \mathcal{E}$ is a metavariable for an expression that has to be evaluated to a channel $c \in \mathbb{C}$.

We use $\{\cdot\}$ to denote multisets/bags and \uplus to denote their disjoint union.

The following figure defines the syntax of our calculus with local memories. The right column represents designators for the respective syntactic categories.

$O ::= \emptyset \mid \{\bar{c}(e)\} \mid O \uplus O$	outgoing bag	
$\mathcal{A} ::= \emptyset \mid \{\bar{c}(v)\} \mid \mathcal{A} \uplus \mathcal{A}$	message aether	\mathcal{A}
$\mu ::= \text{var } x \mid \langle x := e \rangle \mid c(x) \mid O$	actions	\mathcal{A}
$G ::= \mathbf{0} \mid \mu.T \mid G + G$	guards	\mathcal{G}
$T ::= G \mid \text{if } e \text{ then } T \text{ else } T \mid I^{x_1, \dots, x_n} \mid T \mid T$	threads	\mathcal{T}
$P ::= [M \triangleleft T]$	processes	\mathcal{P}
$N ::= P \mid \mathcal{A} \mid N \parallel N$	networks	\mathcal{N}

The syntax defines two layers, threads (\mathcal{T}) and networks (\mathcal{N}).

We first explain threads T , which assemble guards G , which in turn perform actions μ . Action $\text{var } x$ declares variable x , action $\langle x := e \rangle$ assigns (the value of) expression e to variable x , action $c(x)$ receives a value over a channel c to store it in variable x ; messages $\bar{c}\langle e \rangle$ that each send some payload e over some channel c are collected in action O which resembles a multicast operation sending a multiset of messages in one go. A guard G can be $\mathbf{0}$ which does nothing, an action prefix $\mu.T$, or an (external) choice $G + G$. Threads can be guards G , conditionals if e then T else T , or refer to *thread identifiers* $I \in \mathcal{I}$ that are equipped with a list of variables x_1, \dots, x_n for which they need access. We require a defining equation $I^{x_1, \dots, x_n} \stackrel{\text{def}}{=} G$ with $\text{fv}(G) \subseteq \{x_1, \dots, x_n\}$ (see Definition 1) for every used thread identifier. Threads may also run in parallel $T \mid T$.

A *process* $[M \triangleleft T]$ associates a memory M (introduced in Sect. 2) with a thread T . Multisets \mathbb{A} collect messages $\bar{c}\langle v \rangle$ in travel, where c and $v \in \mathbb{V}$ are concrete channels and values, respectively, as determined by expression evaluation. A *network* N is composed of parallel processes together with the message aether.

Our calculus allows for *concurrent* threads *within* processes. This is often required, because concurrent activities support a natural modeling principle for node-local code of distributed algorithms. Unless restrictions are imposed, the memory M is *shared*. For example, in process $[M \triangleleft T_1 \mid T_2]$, both T_1 and T_2 have access to the memory M and can manipulate its variables, i.e., both threads can declare new variables and assign values to them. Thus, we get the usual and well-known problems of potentially competing reads and writes, which we do not intend to repeat in this paper. We also do not intend to discuss potential solutions to race conditions. We do, however, intend to be precise about the semantic implications of such an extension concerning variable bindings.

Binders. Our calculus contains *two* binders for variables. (i) The thread $\text{var } x.T$ acts as a binder for x with scope T . (ii) The process $[M \triangleleft T]$ acts as a binder for the variables in $\text{dom}(M)$ with scope T . As usual, we must carefully deal with free and bound variables. This can be done in a mostly straightforward way.

Definition 1 (Bound and Free Variables). We define the functions bv/fv on actions, threads and processes as follows. For actions:

$$\text{bv}(\mu) \triangleq \begin{cases} \{x\} & \text{if } \mu = \text{var } x \\ \emptyset & \text{otherwise} \end{cases} \quad \text{fv}(\mu) \triangleq \begin{cases} \{x\} \cup \text{fv}(e) & \text{if } \mu = \langle x := e \rangle \\ \{x\} \cup \text{fv}(c) & \text{if } \mu = c(x) \\ \bigcup_{\bar{c}\langle e \rangle \in O} \text{fv}((c, e)) & \text{if } \mu = O \\ \emptyset & \text{otherwise} \end{cases}$$

For threads, the full version can be consulted as Definition 4 in the Appendix. Here, we just point out the case for identifiers:

$$\text{bv}(I^{x_1, \dots, x_n}) \triangleq \emptyset \quad \text{fv}(I^{x_1, \dots, x_n}) \triangleq \{x_1, \dots, x_n\}$$

The other cases are defined homomorphically.

For processes, the interpretation of memories as binders yields:

$$\begin{aligned} \text{bv}([M \triangleleft T]) &\triangleq \text{bv}(T) \cup \text{dom}(M) \\ \text{fv}([M \triangleleft T]) &\triangleq \text{fv}(T) \setminus \text{dom}(M) \end{aligned}$$

A variable x is called *fresh* w.r.t. process P if $x \notin \text{bv}(P) \cup \text{fv}(P)$. An occurrence of a variable is *bound* if it occurs within the scope of a binder for it. (Note that $\text{var } x.T$ is a binder for x , so the x in “ $\text{var } x$ ” itself does not qualify as an occurrence of x .) An occurrence of a variable is *free* if it is not bound.

For example, in thread $(\text{var } x.T_1 \mid \langle x := e \rangle.T_2)$, variable x is both free and bound, as $x \in \text{bv}(\text{var } x.T_1)$ and $x \in \text{fv}(\langle x := e \rangle.T_2)$.

As usual, we may employ the concept of α -conversion to identify processes that only differ in the concrete naming of variables. Likewise, we may rename bound variables, when needed, by consistently replacing all bound occurrences together with the respective binders with appropriately fresh variables. We write $T_1 =_\alpha T_2$, if T_1 and T_2 differ only in consistent renamings of var-bound variables.

Here, we also apply this principle to processes $[M \triangleleft T]$. We may rename variables in T that are bound by M with fresh variables: We do so by consistently replacing them in M —i.e., in $\text{dom}(M)$, as the values associated by M do *not* contain variables—together with all of the respective bound occurrences in T . Formally, replacing a binding for x in M (i.e., with $x \in \text{dom}(M)$) by a binding for a sufficiently fresh y to the M -value of x , can be defined as

$$\{y/x\}M \triangleq (M \upharpoonright_{\text{dom}(M) \setminus \{x\}}) \langle y \mapsto M(x) \rangle$$

by first removing the binding for x ($M \upharpoonright_{\text{dom}(M) \setminus \{x\}}$), then updating $\langle y \mapsto M(x) \rangle$. Let $\{y/x\}T$ denotes the standard substitution of free occurrences of x in T with y . Assuming $x \in \text{dom}(M)$ and y fresh for $[M \triangleleft T]$, we then define:

$$[M \triangleleft T] =_\alpha [\{y/x\}M \triangleleft \{y/x\}T]$$

The reflexive, symmetric and transitive closure of $=_\alpha$ is of course an equivalence. As it just involves consistent in-place renamings of variables, it also satisfies congruence properties. For example, we define $[M \triangleleft T] =_\alpha [M \triangleleft T']$ if $T =_\alpha T'$.

Sanity Conventions. Processes shall provide sufficient knowledge about their local variables. Therefore, a process P is called *closed* if $\text{fv}(P) = \emptyset$. It is practically useful to always require closedness, as the intuitive meaning of an “open” process referring to free variables would be rather dubious: Where should such variables, not bound to their process, refer to? We generalize closedness of processes to networks by stating: A network N is called *legal*, if all its processes are closed.

For verification purposes, we use memories with the intention to access specifically-named local variables. Allowing the application of α -conversion during the course of execution obviously defeats this purpose.¹ Thus, we require that, in any given application, variable names will be chosen such that there is no need to refer to α -conversion when declaring new variables. One ingredient in this respect is that we only permit defining equations $I^{x_1, \dots, x_n} \stackrel{\text{def}}{=} G$ with $\text{bv}(G) = \emptyset$.

¹ The same problem was observed by the authors of [8] when doing invariant proofs.

$$\begin{array}{l}
 \text{(T-ALPHA)} \frac{T_1 =_\alpha T_2}{T_1 \equiv T_2} \qquad \text{(T-OUT)} \frac{}{\emptyset.T \equiv T} \\
 \text{(T-PAR)} \frac{T_1 \equiv T_2 \quad T \in \mathcal{T}}{T_1 \mid T \equiv T_2 \mid T} \\
 \text{(P-ALPHA)} \frac{P_1 =_\alpha P_2}{P_1 \equiv P_2} \qquad \text{(P-MEM)} \frac{T_1 \equiv T_2 \quad M \in \mathcal{M}}{[M \triangleleft T_1] \equiv [M \triangleleft T_2]} \\
 \text{(N-CHEM)} \frac{}{\mathbb{E}_1 \parallel \mathbb{E}_2 \equiv \mathbb{E}_1 \uplus \mathbb{E}_2} \qquad \text{(N-PAR)} \frac{N_1 \equiv N_2 \quad N \in \mathcal{N}}{N_1 \parallel N \equiv N_2 \parallel N}
 \end{array}$$

Fig. 1. Structural Equivalence[s]

$$\text{(STR)} \frac{N \equiv \widehat{N} \quad \widehat{N} \rightarrow \widehat{N}' \quad \widehat{N}' \equiv N'}{N \rightarrow N'} \qquad \text{(PAR)} \frac{N \rightarrow N' \quad \widehat{N} \in \mathcal{N}}{N \parallel \widehat{N} \rightarrow N' \parallel \widehat{N}}$$

Fig. 2. Structure I

Definition 2 (Structural Equivalence). We define the equivalence \equiv for threads (\mathcal{T}), processes (\mathcal{P}) and networks (\mathcal{N}) by the rules in Fig. 1.²

For threads, we assume that both $(\mathcal{G}, +, \mathbf{0})$ and $(\mathcal{T}, \mid, \mathbf{0})$ are commutative monoids. In addition, we include α -conversion by rule T-ALPHA, while rule T-OUT gets rid of empty outgoing bags.

For processes, we also include α -conversion by rule P-ALPHA, while rule P-MEM simply embeds thread congruence.

For networks, we assume that $(\mathcal{N}, \parallel, \emptyset)$ is a commutative monoid. Moreover, rule N-CHEM allows us to combine and separate multisets of traveling messages.

Let \equiv_q denote structural equivalence in which rules T-ALPHA and P-ALPHA are not allowed.

Note that the equivalence \equiv preserves the set of free variables and satisfies some useful congruence properties, due to the inclusion of the rules T-PAR and N-PAR. Note further that we will only consider closed processes in spite of rule P-MEM leaving this aspect open.

Operational Semantics. We define the notion of execution of networks as an unlabeled transition relation on \mathcal{N} . As usual, we exploit the structural equivalence relation \equiv via the rule STR in Fig. 2. Rule PAR allows us to focus on the actions of individual processes: these are captured by the rules in Fig. 3 and 4. Rules DECL, ASSIGN, and RCV are *memory-changing*. Rules SND, TRUE, FALSE, and IDENT are *not memory-changing*. Rules SND and RCV are *global-state-changing*.

² For simplicity, we use the symbol \equiv with heavy overloading. The use of metavariables and the respective context will act like an implicit typing scheme.

$$\begin{array}{c}
\text{(DECL)} \frac{x \notin \text{dom}(M) \cup \text{fv}(\widehat{T})}{[M \triangleleft \text{var } x.T \mid \widehat{T}] \rightarrow [M \langle x \mapsto \top \rangle \triangleleft T \mid \widehat{T}]} \\
\text{(ASSIGN)} \frac{x \in \text{dom}(M) \quad \text{eval}_M(e) = v \in \mathbb{V}}{[M \triangleleft \langle x := e \rangle.T \mid \widehat{T}] \rightarrow [M \langle x \mapsto v \rangle \triangleleft T \mid \widehat{T}]} \\
\text{(RCV)} \frac{\text{eval}_M(c) = c \in \mathbb{C} \quad x \in \text{dom}(M) \quad v \in \mathbb{V}}{[M \triangleleft c(x).T \mid \widehat{T}] \parallel \{\bar{c}\langle v \rangle\} \rightarrow [M \langle x \mapsto v \rangle \triangleleft T \mid \widehat{T}]}
\end{array}$$

Fig. 3. Local Memory-Changing Steps

Rule DECL declares a *new* variable for memory M , so $x \notin \text{dom}(M)$ is clear. We also require $x \notin \text{fv}(\widehat{T})$, as $M \langle x \mapsto \top \rangle$ is a binder for x . Note that for closed processes, $x \notin \text{dom}(M)$ implies $x \notin \text{fv}(\widehat{T})$. Rule ASSIGN evaluates expression e and updates variable x in memory M , but only if it is already defined in M . It is not allowed to reset the variable to \perp (undefined) or \top (initialized).³

Rule RCV defines the reception of message $c \langle v \rangle$. Just like assignment, the received value v updates variable x in memory M ; we only need to further check whether expression c evaluates to channel c . Note, however, that reception may overwrite previous values; this imperative style [10] distinguishes our approach from the “classical” functional style of input, as in CCS [19].

Rule SND selects one of the messages $\bar{c}\langle e \rangle$ in the outgoing bag O ; it then evaluates both c and e and checks whether they fit the requirement of resulting in a channel c and a value v . In case of success, the message is removed from O (where \setminus denotes multiset removal), and its evaluated counterpart $\bar{c}\langle v \rangle$ is placed into the network as “message in travel”.⁴ Rule IDENT describes the insertion of threads via identifiers. The premises ensure that the variables x_1, \dots, x_n of I are captured—as with dynamic scoping—by the associated memory M and that no other variables are accessed from within the defined body G . The rules TRUE and FALSE for evaluating conditionals are standard.

³ Our treatment of variables, the declaration and evaluation is similar to Garavel’s [10] who argues that it is important to have variables not only be declared, but initialized. Garavel [10] suggests to have a static semantics check whether uninitialized variables would be used “too early”. We propose to have the $\text{eval}()$ -function take care of this: for uninitialized variables, it returns \perp and prevents the application of rule ASSIGN.

⁴ In the spirit of asynchronous communication, a thread T shall not be blocked by $O.T$. At least, it shall not be blocked by the non-availability of some matching receiver. Here, the potential blocking is fully caused on the sending side, as the outgoing messages must be evaluated, before the thread T may continue. We consider this OK.

$$\begin{array}{c}
\text{(SND)} \frac{\bar{c}\langle e \rangle \in O \quad O' = O \setminus \{\bar{c}\langle e \rangle\} \quad \text{eval}_M(c) = \mathbf{c} \in \mathbb{C} \quad \text{eval}_M(e) = \mathbf{v} \in \mathbb{V}}{[M \triangleleft O.T \mid \hat{T}] \rightarrow [M \triangleleft O'.T \mid \hat{T}] \parallel \uparrow \bar{c}\langle \mathbf{v} \rangle \downarrow} \\
\text{(IDENT)} \frac{I^{x_1, \dots, x_n} \stackrel{\text{def}}{=} G \quad \text{fv}(G) = \{x_1, \dots, x_n\} \quad \{x_1, \dots, x_n\} \subseteq \text{dom}(M)}{[M \triangleleft I^{x_1, \dots, x_n} \mid \hat{T}] \rightarrow [M \triangleleft G \mid \hat{T}]} \\
\text{(TRUE)} \frac{\text{eval}_M(e) = \mathbf{t}}{[M \triangleleft \text{if } e \text{ then } T_1 \text{ else } T_2 \mid \hat{T}] \rightarrow [M \triangleleft T_1 \mid \hat{T}]} \\
\text{(FALSE)} \frac{\text{eval}_M(e) = \mathbf{f}}{[M \triangleleft \text{if } e \text{ then } T_1 \text{ else } T_2 \mid \hat{T}] \rightarrow [M \triangleleft T_2 \mid \hat{T}]}
\end{array}$$

Fig. 4. Local Non-Memory-Changing Steps

Example. As we require processes to be closed, let $x \in \text{dom}(M)$ for:

$$[M \triangleleft \text{var } x.T_1 \mid \langle x := e \rangle.T_2]$$

With $x \in \text{bv}(\text{var } x.T_1)$ and $x \in \text{fv}(\langle x := e \rangle.T_2)$, the occurrence of x in $\langle x := e \rangle.T_2$ is bound by M , whereas $\text{var } x.T_1$ declares x as a “private” variable with scope T_1 .⁵ Note how the premise of rule DECL ensures to require an α -conversion before the variable can actually be declared.⁶

The following lemma states that α -conversion and transitions get along well.

Lemma 1 (Preservation). *Let N be a legal network. If $N \rightarrow N'$, then N' is legal.*

Proof (Sketch). Variables are never removed from memories M . They can only be changed via α -conversions, but then their bound occurrences in the associated process will be changed accordingly. Otherwise, memories can only grow.

Variables that are bound within the scope of a declaration will remain bound when rule (DECL) is applied, but then by the associated memory M .

⁵ LNT [10] uses var-environments to delimit the scope of variables. Semantically, LNT introduces *stores* (similar to our memories) to keep track of associated values.

⁶ In [10], Garavel suggests to even “prohibit shared variables” and states that in Occam [18] and LOTOS-successor LNT [11] “a parallel composition is considered to be invalid if any of its branches may change the value of a variable used in another branch”. While Occam tries to prevent this at run time, LNT “adds static semantic constraints that forbid at compile time all (syntactically correct) behaviors involving shared variables” [10]. Such considerations can be added on top of our formalization.

$$\begin{array}{c}
\text{(TRIM)} \frac{\text{trim} \in \mathcal{L}}{\emptyset \blacktriangleright N \mapsto \emptyset \blacktriangleright_{\text{trim}} N} \qquad \text{(FAIL)} \frac{\text{trim} \neq k \notin F}{F \blacktriangleright_{\text{trim}} N \mapsto F \cup k \blacktriangleright_{\text{trim}} N}
\end{array}$$

Fig. 5. Failures

4 Location Failures and Their Detection

Syntax. We follow the approach of [21] and introduce a set $\mathbb{L} \subseteq \mathbb{V}$ of *location names*. In our calculus, we then let the processes $[M \triangleleft T]$ of Sect. 3 evolve into *locations* $\ell[M \triangleleft T]$, where $\ell \in \mathbb{L}$, so locations are simply *located processes* [12]. Conveniently, locations may also serve as a natural unit of failure.

We adapt the communication actions of Sect. 3 to become “location-aware”:

- Output $\bar{c}@l\langle e \rangle$ adds the name of the intended target;
- Input $c@l(x)$ adds the name of the intended source;

where l represents an expression that is expected to be evaluated to a location name, so we should require that $\text{eval}_M(l) \in \mathbb{L}$. This has two concrete advantages: (i) Location-aware send actions fit to the intended application domain. (ii) Location-aware receive actions conveniently support suspicions. Message in travel, the elements of bags $\mathbb{A}\mathbb{E}$, now take the form $c_{\text{src} \rightarrow \text{trg}}\langle v \rangle$, with $\text{src}, \text{trg} \in \mathbb{L}$ indicating the source and target of the message.

Structural Equivalence. We adapt the rules of Definition 2 to the extended syntax. The changes from processes to locations and the location-aware forms of communication actions and the messages in transit are orthogonal to the rules.

Operational Semantics. In order to track the failures of locations, we again follow [21] and identify a so-called *trusted-immortal* location trim that cannot fail and will never be suspected. With this abstraction, it is almost trivial to model systems that satisfy *Weak Accuracy* (see Sect. 1). We use global configurations of the form $F \blacktriangleright_{\text{trim}} N$, in which (i) $F \subseteq \mathbb{L}$ indicates which locations *have* failed (so far); (ii) trim is the dynamically determined trusted immortal; (iii) N is a network running in the context of (i) and (ii). For Weak Accuracy, it is required [21] that the very first transition of an execution randomly chooses the trusted immortal from the set of available location names. Rule TRIM in Fig. 5 shows how we represent this behavior starting out from an initial configuration. Rule FAIL then allows any location to fail at any time, unless it has already failed or is immortal. Note that, in case of a location failure, we allow that the associated memory may still be inspected in spite of the location no longer contributing.

Figure 6 embeds the steps of the (adapted) semantics of the location-free calculus into the location-aware setting. Assuming that those steps now carry a label $@\ell$ (see Figs. 8 and 9), rule N-STEP allows such steps only if their responsible location ℓ has not (yet) failed. Rule N-SUSP relies on the label $\text{susp}(k)@\ell$

$$\begin{array}{c}
\text{(N-STEP)} \frac{N \xrightarrow{\textcircled{\ell}} N' \quad \ell \notin F}{F \blacktriangleright_{\text{trim}} N \mapsto F \blacktriangleright_{\text{trim}} N'} \\
\text{(N-SUSP)} \frac{N \xrightarrow{\text{susp}(k)\textcircled{\ell}} N' \quad k \neq \text{trim} \quad \ell \notin F}{F \blacktriangleright_{\text{trim}} N \mapsto F \blacktriangleright_{\text{trim}} N'}
\end{array}$$

Fig. 6. Located Steps

$$\begin{array}{c}
\text{(L-PAR)} \frac{N \xrightarrow{\eta} N' \quad \widehat{N} \in \mathcal{N}}{N \parallel \widehat{N} \xrightarrow{\eta} N' \parallel \widehat{N}} \\
\text{(L-STR)} \frac{N \equiv \widehat{N} \quad \widehat{N} \xrightarrow{\eta} \widehat{N}' \quad \widehat{N}' \equiv N'}{N \xrightarrow{\eta} N'}
\end{array}$$

Fig. 7. Structure II

to govern suspicions: it indicates that (a thread at) location ℓ would like suspect location k to have failed. This is generously permitted, unless it applies to the trusted location `trim` and unless the suspector itself has failed. As a consequence, every run generated with these rules satisfies Weak Accuracy.

Figure 7 is the counterpart to Fig. 2, but now adapted to deal with location-aware labels $\eta \in \{\textcircled{\ell}, \text{susp}(k)\textcircled{\ell} \mid \ell, k \in \mathbb{L}\}$.

Figure 8 contains the location-aware variants of the rules in Fig. 3. Rules L-DECL and L-ASSIGN now take place in locations as opposed to just processes. However, rule L-RCV will now only allow a thread inside a location at ℓ to receive a message $c_{\text{src} \rightarrow \ell}(\mathbf{v})$ if two conditions are satisfied: it must be explicitly addressed to ℓ and it also must originate from the expected source location at `src`.

Figure 9 contains the location-aware variants of the rules in Fig. 4. In addition, rule L-SUSP allows a thread to ignore a reception by launching a suspicion request for the intended source location of the sender. Rule L-SND differs from rule SND of Fig. 4 mainly in the formation of the message in travel: now, mes-

$$\begin{array}{c}
\text{(L-DECL)} \frac{x \notin \text{dom}(M) \cup \text{fv}(\widehat{T})}{\ell[M \triangleleft \text{var } x.T \mid \widehat{T}] \xrightarrow{\textcircled{\ell}} \ell[M \langle x \mapsto \top \rangle \triangleleft T \mid \widehat{T}]} \\
\text{(L-ASSIGN)} \frac{x \in \text{dom}(M) \quad \text{eval}_M(e) = \mathbf{v} \in \mathbb{V}}{\ell[M \triangleleft \langle x := e \rangle.T \mid \widehat{T}] \xrightarrow{\textcircled{\ell}} \ell[M \langle x \mapsto \mathbf{v} \rangle \triangleleft T \mid \widehat{T}]} \\
\text{(L-RCV)} \frac{\text{eval}_M(c) = \mathbf{c} \in \mathbb{C} \quad \text{eval}_M(l) = \text{src} \in \mathbb{L} \quad x \in \text{dom}(M)}{\ell[M \triangleleft c\textcircled{l}(x).T \mid \widehat{T}] \parallel \{c_{\text{src} \rightarrow \ell}(\mathbf{v})\} \xrightarrow{\textcircled{\ell}} \ell[M \langle x \mapsto \mathbf{v} \rangle \triangleleft T \mid \widehat{T}]}
\end{array}$$

Fig. 8. Located Memory-Changing Steps

$$\begin{array}{c}
\text{(L-SUSP)} \frac{\text{eval}_M(l) = \text{src} \in \mathbb{L}}{\ell[M \triangleleft c @ l(x).T \mid \widehat{T}] \xrightarrow{\text{@}\ell} \ell[M \triangleleft T \mid \widehat{T}]} \\
\text{(L-SND)} \frac{\begin{array}{c} \bar{c} @ l(e) \in O \qquad O' = O \setminus \{ \bar{c} @ l(e) \} \\ \text{eval}_M(c) = c \in \mathbb{C} \quad \text{eval}_M(l) = \text{trg} \in \mathbb{L} \quad \text{eval}_M(e) = v \in \mathbb{V} \end{array}}{\ell[M \triangleleft O.T \mid \widehat{T}] \xrightarrow{\text{@}\ell} \ell[M \triangleleft O'.T \mid \widehat{T}] \parallel \{ c_{\ell \rightarrow \text{trg}}(v) \}} \\
\text{(L-IDENT)} \frac{I^{x_1, \dots, x_n} \stackrel{\text{def}}{=} G \quad \text{fv}(G) \subseteq \{x_1, \dots, x_n\} \quad \{x_1, \dots, x_n\} \subseteq \text{dom}(M)}{\ell[M \triangleleft I^{x_1, \dots, x_n} \mid \widehat{T}] \xrightarrow{\text{@}\ell} \ell[M \triangleleft G \mid \widehat{T}]} \\
\text{(L-TRUE)} \frac{\text{eval}_M(e) = \text{t}}{\ell[M \triangleleft \text{if } e \text{ then } T_1 \text{ else } T_2 \mid \widehat{T}] \xrightarrow{\text{@}\ell} \ell[M \triangleleft T_1 \mid \widehat{T}]} \\
\text{(L-FALSE)} \frac{\text{eval}_M(e) = \text{f}}{\ell[M \triangleleft \text{if } e \text{ then } T_1 \text{ else } T_2 \mid \widehat{T}] \xrightarrow{\text{@}\ell} \ell[M \triangleleft T_2 \mid \widehat{T}]}
\end{array}$$

Fig. 9. Located Non-Memory-Changing Steps

sage $c_{\ell \rightarrow \text{trg}}(v)$ explicitly mentions its source ℓ and target trg . Rules L-IDENT, L-TRUE, and L-FALSE now take place in locations as opposed to just processes.

Normalized Derivations. Due to the design of our semantics rules, every derivation of a transition on configurations $F \blacktriangleright_{\text{trim}} N$ can be normalized. Either, the root of the derivation tree is generated by one of the rules in Fig. 5; then nothing else needs to be considered, as the premises do only depend on F and trim . Or, the root is derived by one of the rules in Fig. 6. Then, the transition premise can be always be derived with an application of rule L-STR of Fig. 7. Its purpose is to rearrange the structure of N as well as the internals of its locations such that rule L-PAR can be applied (possibly multiple times). The goal is to identify a *single* location $\ell[M \triangleleft T \mid \widehat{T}]$, possibly together with a suitable *singleton* “travel bag” in order to enable the application of one of the rules in Figs. 8 and 9. An application of rule L-STR can support this by shifting the identified location to the left, if needed (by L-RCV) together with a suitable message, and also shift the intended thread T to the left inside this location.

5 Case Study: Distributed Consensus

In this section, we formalize the algorithm that we presented in the Introduction within our distributed process calculus and prove that it correctly solves Distributed Consensus, i.e., that it satisfies Validity, Agreement and Termination.

As the algorithm uses booleans and natural numbers, we define our sets of expressions and values accordingly: $\mathbb{B} \cup \mathbb{N} \subseteq \mathbb{V}$. We also need operations on numbers and comparisons among them, so \mathcal{E} shall include $e_1 + e_2$, $e_1 = e_2$,

$$\begin{aligned}
 \text{Consensus}_{(\text{input}_1, \dots, \text{input}_n)} &\stackrel{\text{def}}{=} \\
 \prod_{\ell \in \{1, \dots, n\}} \ell &\left[\begin{array}{l}
 M_{\perp} \langle \text{chan} \mapsto c \rangle \\
 \langle x \mapsto \text{input}_{\ell} \rangle \\
 \langle r \mapsto 0 \rangle \\
 \langle \text{output} \mapsto \top \rangle \\
 \triangleleft \textcircled{1} \mathbb{L}_{\ell}^{\text{chan}, x, r, \text{output}} \quad \Big] \\
 \\
 \mathbb{L}_{\ell}^{\text{chan}, x, r, \text{output}} &\stackrel{\text{def}}{=} \\
 \textcircled{2} \langle r := r + 1 \rangle. & \\
 \textcircled{3} \text{ if } r \leq n & \\
 \text{ then } \textcircled{4} \text{ if } \ell = r & \\
 \quad \text{ then } \textcircled{5} \left(\bigoplus_{\ell \neq j \in \{1, \dots, n\}} \overline{\text{chan}}@j(x) \right). \textcircled{1} \mathbb{L}_{\ell}^{\text{chan}, x, r, \text{output}} & \\
 \quad \text{ else } \textcircled{6} \left(\text{chan}@r(x) \right). \textcircled{1} \mathbb{L}_{\ell}^{\text{chan}, x, r, \text{output}} & \\
 \text{ else } \textcircled{7} \langle \text{output} := x \rangle. \textcircled{8} 0 &
 \end{array}
 \end{aligned}$$

Fig. 10. Algorithm

$e_1 \leq e_2$ for $e_1, e_2 \in \mathcal{E}$. We assume that the evaluation function eval (see Sect. 2) takes care for ill-formed and ill-typed cases by then yielding \perp .

In addition, we use a single channel c for the message exchanges; as our calculus fixes source and target location names in communication actions, it will always be unambiguous for which round a message is intended by simply identifying the sender as the respective coordinator of a round: $\mathbb{C} \triangleq \{c\} \subseteq \mathbb{V}$. Likewise, we let $\mathbb{L} \triangleq \{1, \dots, n\} \subseteq \mathbb{N}$, as this is the convention provided by the algorithm. One may (and should!) criticize the abuse of natural numbers for this purpose, which intentionally confuses location names and round numbers, but in order to remain as close to the pseudo code as possible, we follow this convention.

For the vector $(\text{input}_1, \dots, \text{input}_n)$ of initial proposals for the n participants, the code in Fig. 10 represents the algorithm, as formalized in our calculus. We instrument the code with tags $\textcircled{1} \dots \textcircled{8}$ to refer to positions in the code. (Tag $\textcircled{1}$ is used several times, but always with the same thread identifier).

$\text{Consensus}_{(\text{input}_1, \dots, \text{input}_n)}$ defines a network of locations, one for each participant $\ell \in \{1, \dots, n\}$. Each location is equipped with an initial memory, where we directly set the four variables $\text{chan}, x, r, \text{output}$ to their initial values. Note that all participants dispose of the same channel vector. Note also that their initial memories only potentially differ in their initial proposals. We could also use dedicated var-declarations and assignment steps; the effect would be the same, but at the expense of $4 * 2 * n$ additional execution steps. Note that all locations are closed, so the defined network $\text{Consensus}_{(\text{input}_1, \dots, \text{input}_n)}$ is legal.

On each location, the same code is run, as represented by the thread definition for $L_\ell^{\text{chan},x,r,\text{output}}$. Note that the code does not include any variable declarations, so no α -conversion will ever be needed during execution. The increment of the round number (2) together with the break condition (3) simulate the for-loop of the pseudo code. Apart from this deviation, the code only essentially differs from its pseudo variant in that we do not need to check for “alive(p_r)”, as in our calculus suspicion is, except for the trusted immortal, always allowed (6). Note that the command broadcast x_i is explicit in our code as multiple output (5). Here, we deviate from the pseudo code in that we have a coordinator *not* send a message to itself and then wait for its reception; therefore, we use an if-then-else instead of the two if-then constructs in the pseudo code. Finally, note that the thread is completely sequential; there are no parallel threads.

The execution of the algorithm then starts from $\emptyset \blacktriangleright \text{Consensus}_{(\text{input}_1, \dots, \text{input}_n)}$ with no failed processes, and with a trusted immortal yet to be determined. By the design of our semantics, every reachable configuration can be represented in a standard form, up to structural congruence \equiv_{α} , as follows:

$$\emptyset \blacktriangleright \text{Consensus}_{(\text{input}_1, \dots, \text{input}_n)} \longmapsto^+ F \blacktriangleright_{\text{trim}} \mathcal{A} \parallel \prod_{\ell \in \{1, \dots, n\}} \ell [M_\ell \triangleleft \star_\ell T_\ell]$$

where we use $\prod_{\ell \in L} \ell [M_\ell \triangleleft T_\ell]$ for $L \subseteq \mathbb{L}$ as abbreviation for the parallel composition of locations $\ell [M_\ell \triangleleft T_\ell]$ modulo associativity and commutativity.

Therefore, for every reachable configuration, we can now simply inspect (i) the messages in transit (\mathcal{A}), (ii) the individual local states M_ℓ of all participants, and (iii) the “program counters” \star_ℓ (to be understood as a location-specific metavariable) for all participants. Using this direct access, we can now state an informative (global state) invariant. On the one hand, it is very close to the intuitive reasoning that we sketched in Sect. 1. On the other hand, it is formal and can be checked with precise reference to our operational semantics.

Lemma 2 (Invariant). *Let $(\text{input}_1, \dots, \text{input}_n)$ be a valid vector of proposals. Let $\text{Undecided} \triangleq \{\text{input}_1, \dots, \text{input}_n\}$.*

If $\emptyset \blacktriangleright \text{Consensus}_{(\text{input}_1, \dots, \text{input}_n)} \longmapsto^+ F \blacktriangleright_{\text{trim}} \mathcal{A} \parallel \prod_{\ell \in \{1, \dots, n\}} \ell [M_\ell \triangleleft \star_\ell T_\ell]$, then $\forall \ell \in [1, n]$.

$$\left(\begin{array}{ll} M_\ell(r) < \text{trim} & \rightarrow M_\ell(x) \in \text{Undecided} \\ \bigwedge c_{\text{trim} \rightarrow \ell} \langle v \rangle \in \mathcal{A} & \rightarrow v = M_{\text{trim}}(x) \\ \bigwedge M_\ell(r) = \text{trim} \wedge \ell \neq \text{trim} & \rightarrow \left((\star_\ell \in \{3, 4, 6\}) \rightarrow M_\ell(x) \in \text{Undecided} \right) \\ & \wedge (\star_\ell \in \{7, 1, 2\}) \rightarrow M_\ell(x) = M_{\text{trim}}(x) \\ \bigwedge M_\ell(r) > \text{trim} & \rightarrow M_\ell(x) = M_{\text{trim}}(x) \\ \bigwedge c_{\ell \rightarrow k} \langle v \rangle \in \mathcal{A} \wedge \ell > \text{trim} & \rightarrow v = M_{\text{trim}}(x) \\ \bigwedge M_\ell(r) > n \wedge \star_\ell = 3 & \rightarrow M_\ell(\text{output}) = M_\ell(x) \end{array} \right)$$

Note that we use the convention of TLA+ on the use of *conjunction lists* [16], in which the enlisted conjuncts internally have stronger operator precedence.

The invariant of Lemma 2 points out that for every participant ℓ , depending on their respective round $M_\ell(r)$, the content of its current proposal $M_\ell(x)$ can be constrained. Before round trim, not much can be guaranteed (conjunct 1), as suspicions may be applied at will. However, within round trim, it is precisely the passage of *all* non-coordinators from ⑥ to ① that changes the situation (conjuncts 2 and 3), as none of them may suspect the coordinator trim to have failed. Afterwards, this value will be uniformly proposed by all later coordinators (conjuncts 4 and 5). In the invariant, the statements on messages in \mathcal{A} just strengthen the statement in order to make the induction go through, as the information of the decision value is passed on from locations to messages in transit, from where they will be received by the target locations. Finally, note that if the conditions of the constraints are met in a configuration, then this means that a participant was non-failing for long enough in order to reach this state.

Proof (Sketch). We proceed by induction on the length of the sequence \mapsto^+ (note that this induction starts after the first step to determine trim). The invariant is initially trivially satisfied, as all processes are in round 0 and $\mathcal{A} = \emptyset$.

The induction step addresses

$$\begin{aligned} & F \triangleright_{\text{trim}} \mathcal{A} \parallel \prod_{\ell \in \{1, \dots, n\}} \ell [M_\ell \triangleleft \star_\ell T_\ell] \\ \mapsto & F' \triangleright_{\text{trim}} \mathcal{A}' \parallel \prod_{\ell \in \{1, \dots, n\}} \ell [M'_\ell \triangleleft \star'_\ell T'_\ell] \end{aligned}$$

for which we check all possibilities of deriving such a transition. Note that every derivation that results from an application of the rules FAIL (only changes F to F'), L-TRUE, L-FALSE, L-IDENT, L-SUSP (only change $\star_\ell T_\ell$ to $\star'_\ell T'_\ell$) will keep the invariant valid, as they neither change \mathcal{A} nor any of the M_ℓ . The changes to the position must be checked, but are harmless (e.g., ③ to ④ to ⑥, or ① to ②). Rule L-DECL will never be applied, as there are no variable declarations in the code. Otherwise, we only have to deal with applications of the rules L-ASSIGN, L-SND and L-RCV, appearing in the following cases (in which $\ell \notin F$):

1. participant ℓ moving from ② to ③: rule L-ASSIGN
2. participant ℓ moving from ⑤ to (either again ⑤ or) ①: rule L-SND
3. participant ℓ moving from ⑥ to ①: rule L-RCV
4. participant ℓ moving from ⑦ to ③: rule L-ASSIGN

As an example of case 2, consider $\ell = \text{trim}$ with $M_{\text{trim}}(r) = \text{trim}$ for the first time in position ⑤. If the induction step applies L-SND, then the message $c_{\text{trim} \rightarrow j}(\mathbf{v})$ appearing in \mathcal{A} is the first one originating from trim such that for the induction step afterwards conjunct 2 must be checked. It is satisfied, as rule L-SND will use $\text{eval}_{M_{\text{trim}}}(x) = M_{\text{trim}}(x)$ as the payload \mathbf{v} for this message.

As another example, consider a non-coordinator $\ell (\neq \text{trim})$ in just the round $M_\ell(r) = \text{trim}$ in position ⑥. By induction (conjunct 3, subconjunct 1), $M_\ell(x) \in \text{Undecided}$. If the induction step applies L-RCV, then the message $c_{\text{trim} \rightarrow \ell}(\mathbf{v}) \in \mathcal{A}$ must be available. After the step, participant ℓ will be in position ①, so the second subconjunct of conjunct 3 must be satisfied. This holds, as conjunct 2 is true in the hypothesis, because rule L-RCV updates the memory with the received \mathbf{v} .

Note how this proof makes heavy use of the direct access to the values of local variables $M_\ell(r)$ for the various $\ell \in \mathbb{L}$ at each reachable configuration of an execution. After all, it is this possibility of access to local variables in our calculus that makes the proof doable and thus satisfies the title of this paper.

Theorem 1. *The algorithm of Fig. 10 solves Distributed Consensus.*

Proof (Sketch). *Termination* holds, as the only potentially blocking operation is in input position (5). Participants may always suspect, though, unless they wait for trim. In this case, there will eventually be a message, as trim cannot fail.

Validity holds, as the invariant never uses a value that is not in Undecided.

Agreement holds with conjunct 6 of the invariant and *Termination*.

6 Conclusion

We provide linguistic support for state-based reasoning in distributed process calculi. We do so by equipping located processes, the units of distribution in such calculi, with local memories. We develop syntax and operational semantics for this calculus in two steps, starting with a fault-free version. We demonstrate the applicability of our calculus on the formalization of a fault-tolerant algorithm to solve Distributed Consensus. The correctness proof highlights the proximity of our formalization with the widely-used intuitive correctness arguments.

We conjecture that our calculus (or slight extensions of it) is applicable to the large class of fault-tolerant distributed algorithms, which use typical pseudo code with global asynchronous message passing and reference to local variables, next to simple control structures like loops and conditionals. It is rare in this domain that (channel) name passing, as known from the (Applied) Pi Calculus [20, 1], is needed in algorithms. We see, however, no problem at all to also include a restriction operator in our calculus to govern the scope of channel names.

Further work consists of applying our approach to other Distributed Consensus algorithms and mechanizing the reasoning about the invariant and the correctness proof. In [14], we used state machines and checked their correctness in Isabelle. We plan to develop a similar formalization of our calculus.

A Definitions

Definition 3 (Fetching Values for Variables in Memories).

$$\text{fetch}_M(e) \triangleq \begin{cases} e & \text{if } e \in \mathbb{V} \\ M(e) & \text{if } e \in \mathcal{X} \wedge M(e) \in \mathbb{V} \\ (\text{fetch}_M(e_1), \dots, \text{fetch}_M(e_n)) & \text{if } e = (e_1, \dots, e_n) \\ f(\text{fetch}_M(e')) & \text{if } e = f(e') \\ \perp & \text{else} \end{cases}$$

Definition 4 (Bound and Free Variables). *We define the functions bv/fv on threads as follows:*

$$\begin{aligned}
 \text{bv}(\mathbf{0}) &\triangleq \emptyset \\
 \text{bv}(\mu.T) &\triangleq \text{bv}(\mu) \cup \text{bv}(T) \\
 \text{bv}(G_1 + G_2) &\triangleq \text{bv}(G_1) \cup \text{bv}(G_2) \\
 \text{bv}(I^{x_1, \dots, x_n}) &\triangleq \emptyset \\
 \text{bv}(\text{if } e \text{ then } T_1 \text{ else } T_2) &\triangleq \text{bv}(T_1) \cup \text{bv}(T_2) \\
 \text{bv}(T_1 \mid T_2) &\triangleq \text{bv}(T_1) \cup \text{bv}(T_2) \\
 \\
 \text{fv}(\mathbf{0}) &\triangleq \emptyset \\
 \text{fv}(\mu.T) &\triangleq (\text{fv}(\mu) \cup \text{fv}(T)) \setminus \text{bv}(\mu) \\
 \text{fv}(G_1 + G_2) &\triangleq \text{fv}(G_1) \cup \text{fv}(G_2) \\
 \text{fv}(I^{x_1, \dots, x_n}) &\triangleq \{x_1, \dots, x_n\} \\
 \text{fv}(\text{if } e \text{ then } T_1 \text{ else } T_2) &\triangleq \text{fv}(e) \cup \text{fv}(T_1) \cup \text{fv}(T_2) \\
 \text{fv}(T_1 \mid T_2) &\triangleq \text{fv}(T_1) \cup \text{fv}(T_2)
 \end{aligned}$$

References

1. Abadi, M., Blanchet, B., Fournet, C.: The applied pi calculus: mobile values, new names, and secure communication. *J. ACM* **65**(1), 1–41 (2017)
2. ABZ—Rigorous State Based Methods. <https://abz-conf.org/>. A conference series dedicated to the use of state-based formal methods
3. Bergstra, J., Klop, J.W.: Algebra of communicating processes with abstractions. *Theoret. Comput. Sci.* **37**(1), 77–121 (1985)
4. Boudol, G., Castellani, I., Hennessy, M., Kiehn, A.: A theory of processes with localities. *Formal Aspects Comput.* **6**(2), 165–200 (1994)
5. Cardelli, L., Gordon, A.: Mobile ambients. *Theor. Comput. Sci.* **240**(1), 177–213 (2000)
6. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. *J. ACM* **43**(2), 225–267 (1996)
7. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) *CONCUR 2004*. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
8. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV (2013). <http://arxiv.org/abs/1312.7645>. See also ESOP 2012
9. Francalanza, A., Hennessy, M.: A fault tolerance bisimulation proof for consensus (extended abstract). In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 395–410. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_27
10. Garavel, H.: Revisiting sequential composition in process calculi. *J. Log. Algebraic Methods Program.* **84**(6), 742–762 (2015)

11. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) *ModelEd, TestEd, TrustEd*. LNCS, vol. 10500, pp. 3–26. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_1
12. Hennessy, M.: *A Distributed Pi-Calculus*. Cambridge University Press, Cambridge (2007)
13. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall, Hoboken (1985)
14. Küfner, P., Nestmann, U., Rickmann, C.: Formal verification of distributed algorithms. In: Baeten, J.C.M., Ball, T., de Boer, F.S. (eds.) *TCS 2012*. LNCS, vol. 7604, pp. 209–224. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33475-7_15
15. Kühnrich, M., Nestmann, U.: On process-algebraic proof methods for fault tolerant distributed systems. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) *FMOODS/-FORTE -2009*. LNCS, vol. 5522, pp. 198–212. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02138-1_13
16. Lamport, L.: *Specifying Systems*. Addison-Wesley Professional, Boston (2002)
17. Lamport, L., Lynch, N.: Distributed computing: models and methods. In: van Leeuwen, J. (ed.) *Handbook of Theoretical Computer Science*, vol. B: Formal Models and Semantics, chap. 18, pp. 1157–1199. Elsevier (1990)
18. May, D.: Occam. *SIGPLAN Not.* **18**(4), 69–79 (1983). <https://doi.org/10.1145/948176.948183>
19. Milner, R.: *Communication and Concurrency*. Prentice Hall, Hoboken (1989)
20. Milner, R.: *Communicating and Mobile Systems: The π -Calculus*. Cambridge University Press, Cambridge (1999)
21. Nestmann, U., Fuzzati, R.: Unreliable failure detectors via operational semantics. In: Saraswat, V.A. (ed.) *ASIAN 2003*. LNCS, vol. 2896, pp. 54–71. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40965-6_5
22. Nestmann, U., Fuzzati, R., Merro, M.: Modeling consensus in a process calculus. In: Amadio, R., Lugiez, D. (eds.) *CONCUR 2003*. LNCS, vol. 2761, pp. 399–414. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45187-7_26
23. Tel, G.: *Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge (1994)
24. Wagner, C., Nestmann, U.: States in process calculi. In: Borgström, J., Crafa, S. (eds.) *Proceedings of EXPRESS/SOS 2014*. EPTCS, vol. 160, pp. 48–62 (2014). <https://doi.org/10.4204/EPTCS.160.6>