Formal Verification of Distributed Algorithms

Philipp Küfner, Uwe Nestmann, Christina Rickmann

Formal Verification of Distributed Algorithms
From Pseudo Code to Checked Proofs

Philipp Küfner, Uwe Nestmann and Christina Rickmann

Technische Universität Berlin

Abstract. We exhibit a methodology to develop mechanically-checkable parameterized proofs of the correctness of fault-tolerant round-based distributed algorithms in an asynchronous message-passing setting. Motivated by a number of case studies, we sketch how to replace often-used informal and incomplete pseudo code by mostly syntax-free formal and complete definitions of a global-state transition system. Special emphasis is put on the required deepening of the level of proof detail to be able to check them within an interactive theorem proving environment.

1 Introduction

Lamport, Shostak and Pease [LSP82] write about an argument concerning the *Byzantine Generals Problem*: "This argument may appear convincing, but we strongly advise the reader to be very suspicious of such nonrigorous reasoning. Although this result is indeed correct, we have seen equally plausible "proofs" of invalid results. We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm." Along these lines, our goal is to develop mechanically checked proofs about distributed algorithms. In this paper, we address "positive" results on the correctness of problem-solving algorithms, as opposed to "negative" results about the impossibility to solve a problem. We strive for experimental answers to:

- Which description techniques fit best to prove the desired properties of a distributed algorithm when using a theorem prover?
- How big is the gap between informal paper proofs and computer-checked proofs? Is the gap merely due to the level of detail that is hidden in words like "straightforward", or is it due to the incompleteness or inadequacy of the model and the used proof techniques?

Fault-Tolerant Distributed Algorithms Any analysis of distributed algorithms is placed within a particular context provided by a system model in which both the algorithm and its specification are to be interpreted.

Basic Process Model. We assume a setting of finitely many processes that behave at independent speeds. The controlled global progress of a synchronous system, often paraphrased by means of a sequence of communication rounds, is not available. Moreover, in our model, processes communicate via asynchronous

messaging, i.e., without any bound on the delay between emission and reception of messages. Processes typically perform atomic steps consisting of three kinds of actions: (1) input of a message, followed by (2) local computation, followed by (3) output of messages. Starting from some initial configuration of processes and the messaging mechanism, system runs are generated by subsequently performing steps, typically (though not necessarily) in an interleaving fashion. In our model, processes may crash, but do not recover. Here, a process is called *correct* (in a run) if it does not crash (in this run). In other models, where processes may recover, the notion of processes' correctness needs to be adapted.

Distributed Consensus [Lyn96] (or short: Consensus) is a well-known problem in the field of distributed algorithms: n processes with symmetric behavior, but possibly different initial data, are to commonly decide on one out of several possible values. The task is to find an algorithm satisfying the following three trace properties: (1) Validity: If a process decides a value v, then v was initially proposed by some process. (2) Agreement: No two correct processes decide differently. (3) Termination: Every correct process (eventually) decides some value. Fischer, Lynch and Paterson ([FLP85]) showed that Consensus is *not* solvable in a fault-prone environment like the above. Essentially, this is due to the fact that in an asynchronous setting, processes that wait for messages to arrive cannot know whether the sender has crashed or whether the message is just late. Often, this is phrased as a lack of synchrony in the communication infrastructure.

Failure Detection and Round-based Solutions. To enhance the setting by some degree of partial synchrony, Chandra and Toueg [CT96] introduced the notion of *failure detectors* (FD). They allow to determine what is needed to make Consensus solvable in an asynchronous crash-failure environment. A failure detector can be regarded as a local instance at every process that monitors the crash status of all [other] processes. The information provided by the failure detector is usually imperfect, i.e., unreliable to a certain degree. In addition to an enhancement of the model, a common modeling idiom is to use *round-based algorithms*, which help to simulate the spirit of synchronized executions of the processes to some extent: every process keeps and controls a local round number in its own state. This round number can be attached to messages, which can thereby be uniquely identified. In asynchronous systems, this enables receiving processes to arrange the messages sent by other processes into their intended sequential order.

Proofs Revisited. Our set of case studies comprises several distributed algorithms that solve the problem of Consensus for various system models and with varying degree of complexity, taken from [CT96, Lam98, FH07]. For all of these algorithms, proofs of correctness are available, though of quite different level of detail and employing rather different techniques. The proofs often use an induction principle based on round numbers. This counters the fact that executions of distributed algorithms in an asynchronous model do usually *not* proceed globally round-by-round—which resembles the reasoning in a synchronous model—but rather locally step-by-step. More precisely, the proof does not follow the step-by-step structure of executions, but rather magically proceeds from round to round, while referring to ("the inspection of") individual statements in the pseudo code.

Own Work. Motivated by the lack of formality in typical published proofs of the properties of distributed algorithms, it is part of our own research programme to provide comprehensive formal models and develop formal—but still intuitive—proofs, best mechanically checked by (interactive) theorem provers.

Previous Work. In [FMN07, Fuz08], one of the algorithms of [CT96] and the related Paxos algorithm [Lam98] were presented with a high degree of detail and formality: (*i*) the previously implicit setting of the networked processes including the communication infrastructure is made explicit by global state structures; (*ii*) the previous pseudo code is replaced by a syntax-free description of actions that transform vectors of state variables; (*iii*) the behavior of these two parts is modeled as a labeled global transition system, on which the three trace properties required for Consensus are defined; (*iv*) all proof arguments are backed up by explict reference to this transition system.

Contributions. (1) We demonstrate a method based on the models of [FMN07, Fuz08] to represent distributed algorithms, including the underlying mechanisms for communication and failure detection, together with their respective fairness assumptions, within the theorem-proving environment Isabelle [NPW02]. It turns out that the representation of the transition system specification is quite similar to the syntactic representation of actions in Lamport's *Temporal Logic of Actions* (TLA) [Lam02], enriched by the notion of global configuration and system runs. (2) We explain how this representation, especially the first-class status of system runs, can be used to verify *both* safety *and*—as a novelty—liveness properties within a theorem prover. We argue that, to prove safety properties, it is more intuitive to replace the mere invariant-based style (as exemplified in Isabelle by [JM05]) by arguments that make explicit reference to system runs. Moreover, also to prove liveness properties, explicit system runs represent a convenient proof vehicle. (3) Accompanied by the concrete running example[1] inspired by [FH07], we present a formal framework for distributed algorithms and explain techniques for verifying different classes of properties of the algorithm. (4) We give a quick overview of our case studies and provide links to the mechanized proofs in Isabelle. To our knowledge, it is the first time that algorithms from [CT96] have been mechanically checked. In the case of the Paxos proof of [Fuz08], we were able to correct some bugs, while developing the formalization in Isabelle. In contrast, the case study drawn from [FH07] was previously subject to a process-algebraic proof; here, we just used it as a reasonably simple example to demonstrate the quick applicability of our method, small enough to be presented within the space limits of this paper.

Overview of the Paper. In §2, we introduce our formal model based on transition rules and explicit traces that is fundamental for this work. In §3, we present the respective formalization of the required properties. In §4, we discuss how the proof techniques in the respective settings differ, referring to round-based induction and history variables. In §5, we summarize the results of our case studies on doing mechanized proofs for Consensus algorithms. For the sake of readability, we only sketch essential details of the simplest algorithm to demonstrate the method and the shape of the model on which to carry out proofs.

[1] full details at http://www.mtv.tu-berlin.de/fileadmin/a3435/Isa/RotCoord.zip

2 Modeling the algorithm

Distributed algorithms are often described in terms of state machines [Lam78, Sch90], which are used to capture the individual processes' behavior. As in automata theory, transitions from one state to the next may be given by abstractly named actions. In a more concrete version of this model, states can be considered as vectors of values of the relevant variables and transitions are directed manipulations of these vectors. An essential part of a distributed system is the interprocess communication (IPC) infrastructure, e.g., shared memory, point-to-point messaging, broadcast, or remote procedure calls ([Lyn96]). Thus, the global collection of local state machines for the individual processes must be accompanied by some appropriate representation of the IPC.

2.1 Pseudo Code for State Machines

Listing 1.1 exemplifies a common style to present distributed algorithms: a piece of pseudecode that indicates what is executed by each process locally. We use it as a running example to illustrate our experience with Isabelle-checked proofs.

The algorithm solves Consensus in the presence of a *strong* failure detector (see [CT96] and below). We use process ids (PIDs) p_1 to p_n to refer to the n processes. The algorithm proceeds in rounds. For every round, there is one process playing the role of a coordinator. For the given algorithm, coordination means to propose a single value to all processes, while the latter wait for this proposal. The function alive contacts the above-mentioned failure detector to detect the possible failure of the coordinator. The (only) assumption about the failure detector is that at least one correct process is not suspected to be crashed, i.e. the function alive does always return 'True' for some process that never crashes. Correct processes that are not suspected are called *trusted-immortal*.

Listing 1.1. The Rotating Coordinator Algorithm for participant p_i

```
1 x_i := Input
2 for r:= 1 to n do {
3       if r = i then broadcast x_i;
4       if alive(p_r) then x_i := input_from_broadcast;
5 }
6 output x_i;
```

As there is no formal semantics and no compiler for pseudo code, formal reasoning is impossible at this level of abstraction. Furthermore, the environment including the IPC-infrastructure (e.g., the various messages in transit) is usually not an explicit part of the pseudo code model. Nevertheless, we choose pseudo code as a starting point for our formalization, as many algorithms are given in this style. Hence, the model is both informal and incomplete; it cannot be transferred into a theorem-proving environment without giving a formal semantics to the usually textual description of actions and not without providing a complete representation of the IPC-infrastructure. We provide a formal model for this

locale *Algorithm* =
fixes
InitPred :: *'configuration* ⇒ *bool* **and**
ProcActionSet :: (*'configuration* ⇒ *'configuration* ⇒ *proc* ⇒ *bool*) *set* **and**
ComActionSet :: (*'configuration* ⇒ *'configuration* ⇒ *bool*) *set* **and**
ProcessState :: *'configuration* ⇒ *proc* ⇒ *'process-state*
assumes
ActionSetsFin:
finite ProcActionSet
finite ComActionSet **and**
StateInv:
⋀*A i*. ⟦*A*∈*ProcActionSet*; *A c c' i*; *ProcessState c j* ≠ *ProcessState c' j*⟧⟹ *i* = *j*
⋀*A*. ⟦*A*∈*ComActionSet*; *A c c'* ⟧⟹ *ProcessState c j* = *ProcessState c' j*

Fig. 1. Locale for Distributed Algorithm

example, mention more general modeling aspects and show the key techniques used for the respective correctness proofs.

2.2 State Machines in Isabelle

Fuzzati et al. [FMN07] define the algorithm in terms of transition rules where the transitions are computation steps between so-called configurations, which represent the global states of the distributed system. In [FMN07], a configuration at time t consists of three components: (1) an array of the local states of the processes, (2) the message history (a set of all point-to-point messages sent until t) and (3) the broadcast history (a set of all broadcast messages sent until t).

The concrete definition of configurations obviously depends on the respective algorithm. To get a general model for distributed algorithms we introduce an abstract type variable *'configuration*. We apply ideas from Merz et al. [CDM11] and model distributed algorithms as interpretations of the locale given in Fig. 1. Locales are used to introduce parameterized modules in Isabelle's theories (see [Bal03]). In the definition of the locale *Algorithm*, *InitPred* is a predicate that returns true if a given configuration is a valid initial configuration. *ProcActionSet* contains all valid actions that a process can execute: examples are local computations or the sending of messages by placing messages into the local outbox. *ComActionSet* contains (communication) actions that can be performed independently from a dedicated process (by the system or the communication infrastructure); as an example, imagine the loss of a message during transmission. There is no sense in assigning such an action to the sender or the receiver of the message. Both action sets have to be finite. *ProcessState* must be instantiated with a mapping that returns a process state for a given configuration and a process. Of course communication actions are not allowed to change the state of any process and performed by some process may not change the state of any other process. This is asserted by *StateInv*. This allows us to formulate some standard lemmas to be used for each interpretation of the locale. An action *A*

definition
LocalStep:: *'configuration* ⇒ *'configuration* ⇒ *bool* **where**
LocalStep c c' ≡ ∃ *i* ∈ *procs*. ∃ *A* ∈ *ProcActionSet*. *A c c' i*
definition
ComStep:: *'configuration* ⇒ *'configuration* ⇒ *bool* **where**
ComStep c c' ≡ ∃ *A* ∈ *ComActionSet*. *A c c'*
definition
Step:: *'configuration* ⇒ *'configuration* ⇒ *bool* (**infixl** → *900*) **where**
Step c c' ≡ *LocalStep c c'* ∨ *ComStep c c'*
definition
deadlock :: *'configuration* ⇒ *bool* **where**
deadlock c ≡ ∀ *c'*. ¬ *c* → *c'*
definition
FinalStuttering :: *'configuration* ⇒ *'configuration* ⇒ *bool* **where**
FinalStuttering s s' ≡ (*s* = *s'*) ∧ *deadlock s*
definition
Run :: (*T* ⇒ *'configuration*) ⇒ *bool* **where**
Run R ≡ *InitPred* (*R 0*) ∧ (∀ *t*::*T*. ((*R t*) → (*R* (*t*+*1*)))
∨ *FinalStuttering* (*R t*) (*R* (*t*+*1*)))

Fig. 2. Definition: Steps and Runs

from *ProcActionSet* is a predicate that takes two configurations c, c' and a proc i and returns true if and only if A is a valid step from c to c' executed by process i. Likewise, an action A from *ComActionSet* is a predicate over configurations.

We call *LocalStep* the execution of an action from *ProcActionSet*, and *ComStep* the execution of an action from *ComActionSet*. A step from c to c' happens if and only if there is a *LocalStep* or a *ComStep* from c to c' (see Fig. 2). To verify properties of an algorithm, all possible executions of the algorithm must be inspected. We use the term *Run* for the execution of the algorithm and define it as an infinite sequence of configurations where *InitPred* holds in the initial configuration and every configuration and its successor is in the step relation. Of course there might be configurations where no further step is possible. In this case the system is deadlocked. For these cases where the actual execution would be finite we allow the system to take *stuttering* steps, i.e. to repeat the last configuration until the end of time. (see Fig. 2).

IPC The given example requires to implement a message passing mechanism to enable the communication between processes. In our model, a message will traverse three states on its way from the sender to the receiver:

- **outgoing**: When a sender wants to send a message (or a set of messages) it puts the message into its outgoing buffer. Messages in the outgoing buffer are still at the senders site, i.e. outgoing messages are lost if the sender crashes.
- **transit**: The message is on its way to the receiver.
  A crash of the sender does no longer concern messages that are in transit.

```
record MsgStatus =
  outgoing :: nat
  transit  :: nat
  received :: nat

record content =
  snd :: nat
  rcv :: nat

definition Msgs :: (('a content-scheme) ⇒ MsgStatus) ⇒ (('a content-scheme) set)
where
  Msgs M ≡ {m. outgoing (M m) > 0 ∨ transit (M m) > 0 ∨ received (M m) > 0}

definition OutgoingMsgs :: (('a content-scheme) ⇒ MsgStatus) ⇒ (('a
content-scheme) set) where OutgoingMsgs M ≡ {m. outgoing (M m) > 0}

definition TransitMsgs :: (('a content-scheme) ⇒ MsgStatus) ⇒ (('a content-scheme)
set) where TransitMsgs M ≡ {m. transit (M m) > 0}

definition ReceivedMsgs :: (('a content-scheme) ⇒ MsgStatus) ⇒ (('a
content-scheme) set) where ReceivedMsgs M ≡ {m. received (M m) > 0}
```

Fig. 3. Messages

– **received**: The message has already arrived on the receiver's site.
  It is now ready to be processed by the receiver.

We use a multiset-like structure to represent messages in our system, i.e. for every message the number of copies that are **outgoing**, (respectively **transit**, **received**) are stored by mapping messages to records of type *MsgStatus*, a record that can store a number for each option. For such a mapping we use the term message history. The message history is part of each configuration of our algorithm and represents the state of the message evolution. In our model, sender and receiver are stored for each message. The type *Message* will later be extended by the payload of the message (depending on the algorithm we want to model). To work with the set of messages (respectively the set of outgoing messages, of transit messages, of received messages) we define functions that return the respective set for a given message history M (see Fig. 3). Now we are ready to define relations between message histories that describe

– the placement of a message into the outgoing buffer
– the sending of a message, i.e. the change of status from outgoing to transit
– the receiving of a message

Our running example requires to put a set of messages in the outbox. Therefore, we define a directive *MultiSend*: it is a predicate that is true for two message histories M,M' and a set of messages *msgs* if and only if M and M' are equal, except for the **outgoing** values for messages in *msgs*, which are incremented by one in M'. The definitions for changing a message status from **outgoing**

definition *MultiSend* :: ((*'a content-scheme*) ⇒ *MsgStatus*) ⇒ ((*'a content-scheme*) ⇒ *MsgStatus*) ⇒ (*'a content-scheme*) *set* ⇒ *bool* **where**
MultiSend M M' msgs ≡ *M'* = (λ*m*. *if* (*m* ∈ *msgs*) *then incOutgoing* (*M m*) *else M m*)

definition *Transmit* :: ((*'a content-scheme*) ⇒ *MsgStatus*) ⇒ ((*'a content-scheme*) ⇒ *MsgStatus*) ⇒ (*'a content-scheme*) ⇒ *bool* **where**
Transmit M M' m ≡ *m* ∈ *OutgoingMsgs M*
∧ *M'* = *M* (*m* := ⦇ *outgoing* = (*outgoing* (*M m*))−(*1*::*nat*), *transit* = *Suc*(*transit* (*M m*)), *received* = *received* (*M m*)⦈)

definition *Receive* :: ((*'a content-scheme*) ⇒ *MsgStatus*) ⇒ ((*'a content-scheme*) ⇒ *MsgStatus*) ⇒ (*'a content-scheme*) ⇒ *bool* **where**
Receive M M' m ≡ *m* ∈ *TransitMsgs M*
∧ *M'* = *M* (*m* := ⦇ *outgoing* = (*outgoing* (*M m*)), *transit* = *transit* (*M m*)−(*1*::*nat*), *received* = *Suc*(*received* (*M m*))⦈)

Fig. 4. Message movements

to **transit** (*Transmit*) and from **transit** to **received** (*Receive*) are for single messages only and straightforward (see Fig. 4). Note that their preconditions '*m* ∈ *OutgoingMsgs M*' ('*m* ∈ *TransitMsgs M*') are necessary to rule out a decrement of zero accidentally generating infinitely many messages.

Rotating Coordinator Model For the concrete model of our algorithm, we need to define a data structure *configuration* that contains all relevant information of the system for one point in time. We already identified message histories as one important part of every configuration. Of course, processes can take steps and change their local state without changing the message history (for example, by doing local computations or processing received messages). Hence, another important component of a configuration must be the local states of processes. In our case, the state of a process p_i can be defined by five variables:

r	- The round of p_i.
phs	- The phase of p_i, i.e. a next-step indicator.
x	- The value of the variable x_i (cp. listing 1.1)
crashed	- A flag showing whether p_i has crashed.
decision	- A value that is set from $\perp$ to v value when p_i decides v.

We use the value *P1* in *phs* if the process is in line 3 and *P2* if it is in line 4 (c.p. listing 1.1). We do not need more values, as being in line 6 can be detected by testing if $r > n$ (where n is the number of processes) and the remaining lines are just initialization of variables that will be modeled by the respective *InitPred*.

As in [FMN07], we use the term *program counter* for the pair (*r*, *phs*). In a configuration, we have to store a process state entry for each process. Hence, we use a mapping from processes to process states as another component of configurations. As explained above, the message type must be extended by the

```
record process-state =
  r         :: nat
  phs       :: Phase
  x         :: Input
  crashed   :: bool
  decision  :: Input option

record msg = Message +
  cnt-v :: Input

record configuration =
  St :: proc ⇒ process-state
  Me  :: msg ⇒ MsgStatus
```

Fig. 5. Definition: *configuration*

```
definition
  p1msgset :: proc ⇒ Input ⇒ msg set where
  p1msgset i v ≡ {m. ∃j ∈ procs.
        m = (|snd = i, rcv = j, cnt-v = v|) }

definition
  MsgGen :: configuration ⇒ configuration ⇒ proc ⇒ bool where
  MsgGen c c' i ≡ crashed (St c i) = False
  ∧ phs (St c i) = P1 ∧ r (St c i) ≤ N
  ∧ St c' = (St c) (i := (|
  r = r (St c i),
  phs = P2,
  x = x (St c i),
  crashed = False,
  decision = decision (St c i)  |))
  ∧ (if (r (St c i) = PID i) then
    MultiSend (Me c) (Me c') (p1msgset i (x (St c i)))
  else
    Me c' = Me c)
```

Fig. 6. Definition: Action *MsgGen*

respective payload. In our case, the content a process has to send is its current x value, here of type *Input*. Hence, the message type is extended by a field *cnt-v*. As a result of these considerations, we get the type for a configuration as a record consisting of an array of process states and a message history (see Fig. 5).

Next, we define the communication and process actions. Regarding Listing 1.1 after the initialization, a process p_i checks whether it is itself the coordinator of its current round. If so, then it sends a set of messages with $p_i i$ as the sender, the current value of x at p_i as the content, and all processes as the receivers. *p1msgset* in Fig. 6 constructs such a set for the respective arguments p_i and x.

Based on this definition, for example, the definition of action *MsgGen* is read as follows: A step from c to c' is a *MsgGen* step taken by p_i if and only if

- p_i is not crashed in c
- *phs* of p_i is *P1* in c and *P2* in c'

- r of p_i is less or equal than N and is not changed to c'
- x, *crashed* and *decision* of p_i do not change from c to c'
- states of all other processes do not change
- a multisend of the respective *p1msgset* happens from c to c' if the current round of the process equals its process id (PID).

We use the Isabelle function update syntax to implement the desired behaviour: $St\ c' = (St\ c)\ (i := (\!|\ X\ |\!)\ )$ denotes the update in $(St\ c)$ at i to X to yield $(St\ c')$. Thanks to currying in Isabelle, *crashed* $(St\ c\ i)$ denotes the *crashed* variable of process p_i's state in configuration c. *MsgRcvTrust* trusts an awaited sender and receives its message, while *MsgRcvSuspect* suspects a sender by no longer waiting for its message. *Finish* implements the decision step of a process and *Crash* disables a process by setting its *crashed* variable. There are also the two communication actions *MsgSend* and *MsgDeliver* which push messages one step further (from **outgoing** to **transit**, respectively from **transit** to **received**).

For runs, *initial* configurations are those where, for all processes p_i, (1) (*r,phs*) is (*1,P1*), (2) x is the input of p_i, (3) (*crashed,decision*) is (false,$\perp$) and (4) for all messages **outgoing**, **transit** and **received** are set to 0.

We define two sets *ProcActions* and *Networkactions* and write down the interpretation of the introduced locale *Algorithm* as *RotCoord* with *St* as the required mapping from configuration and processes to process states (given in Fig. 7). Note that proofs for the finiteness of *ProcActions* and *NetworkActions* and for the assertions about process states are required. Two lemmas *StateInv1* and *StateInv2* show that the locale assumption *StateInv* is satisfied.

type-synonym *procAction* = *configuration* ⇒ *configuration* ⇒ *proc* ⇒ *bool*

definition
ProcActions :: *procAction set* **where**
ProcActions ≡ {*MsgGen*, *MsgRcvTrust*, *MsgRcvSuspect*, *Crash*, *Finish*}

type-synonym *networkAction* = *configuration* ⇒ *configuration* ⇒ *bool*

definition
NetworkActions :: *networkAction set* **where**
NetworkActions ≡ {*MsgSend*, *MsgDeliver*}

interpretation *RotCoord*: *Algorithm*
Init
ProcActions
NetworkActions
St
by (*unfold-locales*,
auto simp add: *ProcActions-def NetworkActions-def StateInv1 StateInv2*)

Fig. 7. Interpretation *RotCoord*

3 Requirement Specification

Validity and Agreement (see the Introduction) are safety properties; as pure invariants of the algorithm they can be formulated as state predicates. Hence, for Validity and Agreement, it would be sufficient to reason about individual states, and whether the properties are preserved by every transition. Termination is a liveness property; it requires us to consider full runs as first-class entities.

We do not include a formulation of Validity in this paper. Instead, next to the formalizations of Agreement and Termination (see Fig. 8), we explicitly mention Irrevocability for decisions, i.e, that decisions cannot be undone or overwritten.

lemma *Irrevocability*: **assumes** R: *Run* R **and**
d: *decision* (*St* (*R* t) i) $\neq$ *None* **and** z: $z \geq t$
shows
the(*decision*(*St* (*R* z) i)) = *the* (*decision*(*St* (*R* t) i))
decision (*St* (*R* z) i) $\neq$ *None*

theorem *Agreement*: **assumes** R: *Run* R **and**
di: *decision*(*St* (*R* t) i) $\neq$ *None* **and**
dj: *decision*(*St* (*R* t) j) $\neq$ *None*
shows *the*(*decision*(*St* (*R* t) i)) = *the* (*decision*(*St* (*R* t) j))

theorem *Termination*: **assumes** R: *Run* R **and** i: $i \in$ *Correct* R
shows $\exists\, t.$ *decision* (*St* (*R* t) i) $\neq$ *None*

Fig. 8. Irrevocability, Agreement and Termination

4 Proof techniques

'Inspection of the Code' To show that an algorithm exhibits certain properties, we need to refer to its 'code'. Since pseudo code has no formal semantics, this kind of reference cannot be formal. The reader has to believe that certain basic assertions are implied by single lines of the code; e.g., if line 27 states $x := 5$ then, after line 27 is executed by p_i, variable x will indeed have value 5. Reasoning is done by 'inspection of the code'. For distributed algorithms, this kind of local reasoning is of course error-prone, because one might assume $x = 5$ when executing line 28, which might be wrong if x is shared and another process changes x while p_i moves from line 27 to 28. In [FMN07], the reference to local pseudo code is replaced by the reference to formally-defined global transition rules. Then, if some rule A is provably the only one that changes the variable x of process p_i and process p_i's variable has changed from time t_x to t then, obviously "by inspection of" the rules, we can infer that A was executed between t_x and t. Such a setup is a useful basis for its application within a theorem prover.

Invariant-Based Reasoning This well-known technique boils down to the preservation of properties from one configuration to another during computation steps: essentially, it requires a proof by case analysis for all possible actions in such a step. Here, the formal version of 'inspection of the code' is pertinent. Finding a proof that an invariant property holds in some initial configuration leads to the standard proof technique of induction over time t, i.e., along the configurations of a run, which we evidently use a lot in our examples.

History-Based Reasoning Reasoning along the timeline gets more difficult if also assertions about the past are made. Showing that p_i received a message m on its way to configuration c would require to inspect every possible prefix of a run, unless there is some kind of bookkeeping implemented in the model. In [FMN07], this problem is solved by the introduction of history variables [Cli73, Cla78, Cli81] that keep track of events during the execution of the algorithm. For verification purposes of concurrent programs, history variables are common (see [GL00, Owi76]). Technically, we make history variables an explicit part of our model that also serves for the needed IPC. This provides access to the entire communication history. Every sent message is stored in the history and will not be deleted during a run. Hence, when inspecting a configuration $(R\ t)$, all messages sent before t are accessible. Therefore, the above-mentioned assertion can be reduced to the simple check that m is in the message history of c.

Application of Proof Techniques As Validity is an invariant, the technique for invariant-based reasoning is applied. For the used induction principle, it is to show that Validity holds in the initial state of every run and every step of the algorithm preserves it. Hence the main part of the proof is a classical example for the most-used proof technique mentioned above: fix a run R and a time t, then perform induction on t. As a consequence, the remaining proof goals are:

- show that for every initial configuration $R(0) \in \mathbf{Init}\ P(R(0))$ holds.
- show that $P(R(t))$ implies $P(R(t{+}1))$.

Mostly, the first goal is implied by the definition of the initial states **Init**. The second goal requires that every defined transition rule preserves P; the induction hypothesis can be strengthened by the knowledge that the step $R(t) \to R(t{+}1)$ is derived by exactly one application of one of the defined transition rules (distinction of cases). Hence, it remains to show that every application of some rule leads to a successor configuration $R(t{+}1)$ with $P(R(t{+}1))$.

The main argument for Agreement in our running example is that processes cannot skip messages of trusted-immortal processes. Let *ti* be a trusted-immortal process. We sketch the proof that every process has the same value stored in x before entering a round higher than *PID ti* (where *PID* is a function that returns a unique process id from $1..N$ for every process): Every process j that decides a value must traverse every round number between 1 and n and, therefore, also the round number *PID ti* where *ti* is the coordinator. Since j cannot skip the message of a trusted-immortal process, j has to assign the value v_{ti} of *ti* to its

state variable x before entering round *(PID ti) + 1*. Therefore, afterwards, all processes j in higher rounds than *PID ti* will send the value v_{ti} or nothing (if they crash before) and, hence, processes can only apply value v_{ti} in such rounds. Formally, this is expressed by the Lemma *uniformRndsAfterTI2* asserting that two processes in rounds higher than the process id of some trusted-immortal process must have the same x value. This implies agreement since every process that decides, decides for its x value and must be in round $n{+}1$ and $n{+}1$ is greater than every process id. Many more invariants must be derived to prove this lemma and both introduced proof techniques are applied in multiple steps.

The proof for Termination appears to be, at first sight, quite obvious: Processes can only block while waiting for messages of trusted-immortal processes. We sketch the proof how mutual waiting is ruled out. Let ti_1 and ti_2 be two trusted-immortal processes waiting for each other's messages. Without loss of generality, let *PID* $ti_1 \leq$ *PID* ti_2. Since ti_1 waits for the message of ti_2, it must be in round *PID* ti_2 and therefore in a round greater than or equal to its own round. Thus, ti_1 already must have sent the message for round *PID* ti_1 and therefore ti_2 must eventually receive this message.

In our formal model, this proof is quite more difficult. The suggested proof relies on the implicit fairness assumption that every possible process step and every possible message transition from **outgoing** to **transit** and from **transit** to **received** will eventually happen, which is implicitly implied by the model.

Nevertheless, in the given example, it is possible to give a formal proof without introducing further fairness assumptions about the execution of *ProcActions*: since we allow *FinalStuttering* only if no further defined actions are possible, in runs with *FinalStuttering*, there can be no infinitely enabled actions. Hence, one can prove that every action that is enabled either gets disabled later or is executed later on. The proof of Termination therefore relies on proving two assertions: every run of the algorithm has *FinalStuttering* and, at the beginning of the *FinalStuttering*, a correct process is in round number $n{+}1$ and therefore has decided (otherwise the action *Finish* would still be enabled).

5 More Case Studies

Our much more complicated case studies are two widely known Consensus algorithms: (1) one by Chandra and Toueg [CT96] (thus, from now on referenced as CT) that uses the failure detector $\diamond S$—known as the weakest that allows to solve Consensus—and (2) Paxos, by Lamport [Lam98]. The latter does not satisfy Termination, but it does not need failure detectors.

A formal review of both algorithms is found in [Fuz08]. Compared to the running example (Listing 1.1), both cases are much more complex caused by the weaker assumptions on the asynchrony of the environment. For each algorithm,

we required approximately 10k LOC in Isabelle/HOL[2]. The basic model and the proof techniques are essentially the same except for a few mentionable details.

In the Rotating Coordinator algorithm all n processes decide in the same round (in round $n+1$), while in Paxos and CT processes might decide in different rounds; also, there is no upper bound for the traversed round numbers. Moreover, processes can decide values broadcast in different rounds. Hence, we need some kind of global view on the system, i.e. to consider whole configurations. For this purpose, already [CT96] introduce the notion of *Locked Values*. A value is locked for a round r if more than the half of all processes acknowledged the value sent by the coordinator of r. A central lemma for both algorithms states that if v_1 and v_2 are locked values in rounds r_1 and r_2 then $v_1 = v_2$. Inspired by the proof sketches in [CT96], Fuzzati et al. [FMN07] use induction on the round number to prove this lemma: To prove a proposition P holds for all rounds r' with $r' \geq r$ the first step is to show P holds for $r = r'$. In the inductive step, P is shown for round k under the assumption that P holds for all r' with $r \leq r' < k$.

Regarding the timeline, this approach dissents from standard temporal reasoning techniques, as the 'global' round number does not proceed consistently with the global clock. In fact, the round number might be different in all local states of the processes and can evolve independently from the global progress as long as it is monotonically increasing; it is possible that a round number r_i of process p_i is greater than the round number r_j of a process j and later in time $r_i < r_j$ holds. Therefore, proofs done by this technique are intricate, hard to follow by a reader, and not preferred for doing formal proofs. This is documented by errors that we found in [FMN07, Fuz08] (see below). Making such errors within a theorem proving environment is not possible and, hence, we were forced to correct them.

Another difference due to the complexity of Paxos and CT is that there are different types of messages within single rounds; hence, we get dependencies of messages. For example, if message m_2 depends on the prior reception of message m_1, we can deduce that, if p_i sent m_2 to p_i, it must have received m_1. Moreover, the sender of m_1 must indeed have sent m_1. Thus, new proof patterns arise for dependencies between and also concerning their contents.

One important contribution of the mechanizing proofs is the awareness that even proofs at such a formal level as [Fuz08] can exhibit severe faults without being noticed. During our work, we found several problems both in the model and the proofs. The major problems we found in the proof for Paxos of [Fuz08] are:

- There was an error in the broadcast mechanism that circumvented a delivery of broadcasts to all processes except for its sender and therefore would render executions, where only the minimal majority of processes are alive when the first process decides, nonterminating. Moreover an assumption about the mechanism claimed that every broadcast will eventually be received by all correct processes. Due to the error mentioned before this is in contradiction

[2] full models and proofs can be found at
http://www.mtv.tu-berlin.de/fileadmin/a3435/Isa/CT.zip
http://www.mtv.tu-berlin.de/fileadmin/a3435/Isa/Paxos.zip

to the transition rules. Of course from this contradiction one could derive any property needed.
- Another problem concerned the basic orderings that are introduced for the reasoning on process states. It turned out that the ordering does not fullfill the required monotoncity in time that was assumed. Since many proofs for the following lemmas relied on this ordering, this problem is serious.
- The proof for one of the central lemmas (Locking Agreement) is wrong. It uses another lemma (Proposal Creation), but its assumptions are not fullfilled. Therefore, we had to find an adequate version of this lemma with weaker assumptions and redo both proofs (a similar error occurs in [FMN07]).

6 Conclusion

Exemplified with Consensus algorithms, we show how to represent their widespread informal and incomplete pseudo-code descriptions instead in a formal and complete way that can be processed within a theorem prover. It is not our intention to suggest algorithm designers shall start with pseudo code; we rather show how given pseudo descriptions can be formalized. Furthermore, we may thus point out alternative algorithm representations that *can* be formalized in theorem proving environments. By intention, our approach (continuing our previous 'pencil-and-paper' work [FMN07]) is close to the well-known abstract state machines from Gurevich[Gur93], and also actions in the TLA-format, as it is our goal to achieve formalizations of algorithms and their proofs that are reasonably close to the intuitions of typical researchers in the field. The formalization usually requires to add details to the pseudo code so it hardly ever correponds one-to-one. However, this can also be seen as an advantage as it forces to clarify potential sources of misinterpretation.

When mechanizing the proofs (or rather: previous proof sketches), we tried to stick to the intuitive arguments and proof techniques as much as possible. Hence automatic tools like Sledgehammer or Quickcheck were not used. But, mechanization requires us to write out *all* the details; thereby, it proves that the intuitive reasoning (also in our own previous work) is often enough too sloppy.

We report on three case studies within this paper. (1) Our running example is very simple, as it is based on strong assumptions about the system model. We chose it just as a convenient representative for this paper, as it is impossible to show the more interesting case studies within the space constraints. Still, most of the method can be exemplified with it. We found this example in a process-algebraic setting [FH07], and also wanted to be able to roughly compare the amount of work needed in the two completely different settings. We now believe that the approach of the current paper is more intuitive—and mechanized!—and thus leads to quicker proofs. (2) The CT-algorithm has now, to our knowledge the first mechanically-checked proofs, including Termination. The latter is only possible, as our formal model includes an explicit representation of runs. (3) The Paxos algorithm can, modulo some changes to the model, be seen as a variant of CT. As mentioned before, the work on Disk Paxos in [JM05] is quite similar

to our work for safety properties. The main difference is based on the different model that allows us to comfortably prove liveness properties in the case of CT.

References

Bal03. C. Ballarin. Locales and locale expressions in isabelle/isar. In *TYPES*, 2003.

CDM11. B. Charron-Bost, H. Debrat, and S. Merz. Formal verification of consensus algorithms tolerating malicious faults. In *Proceedings of SSS 2011*, volume 6976 of *LNCS*, pages 120–134, Grenoble, France, 2011. Springer.

Cla78. E. M. Clarke. Proving the correctness of coroutines without history variables. In *ACM-SE 16*, pages 160–167, New York, NY, USA, 1978. ACM.

Cli73. M. Clint. Program proving: Coroutines. *Acta Informatica*, 2:50–63, 1973.

Cli81. M. Clint. On the use of history variables. *Acta Informatica*, 16:15–30, 1981.

CT96. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, 1996.

FH07. A. Francalanza and M. Hennessy. A Fault Tolerance Bisimulation Proof for Consensus. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 395–410, 2007.

FLP85. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

FMN07. R. Fuzzati, M. Merro, and U. Nestmann. Distributed Consensus, Revisited. *Acta Informatica*, 44(6):377–425, 2007.

Fuz08. R. Fuzzati. *A Formal Approach to Fault Tolerant Distributed Consensus*. PhD thesis, EPFL, Lausanne, 2008.

GL00. E. Gafni and L. Lamport. Disk Paxos. In *Distributed Computing*, pages 330–344, 2000.

Gur93. Y. Gurevich. Evolving algebras: An attempt to discover semantics, 1993.

JM05. M. Jaskelioff and S. Merz. Proving the correctness of disk paxos. In *The Archive of Formal Proofs*. http://afp.sf.net/entries/DiskPaxos.shtml, 2005.

Lam78. L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95 – 114, 1978.

Lam98. L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

Lam02. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

LSP82. L. Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM ToPLaS*, 4(3):382–401, 1982.

Lyn96. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Pub., 1996.

NPW02. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Owi76. S. Owicki. A consistent and complete deductive system for the verification of parallel programs. In *Proceedings of STOC '76*, pages 73–86. ACM, 1976.

Sch90. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22:299–319, 1990.