Electronic Communications of the EASST
Volume XXX (2015)

Proceedings of the
15th International Workshop on
Automated Verification of Critical Systems (AVoCS 2015)

Formal Specification, Verification, and Implementation of Fault-Tolerant Systems

Vincent Rahli and David Guaspari and Mark Bickford and Robert L. Constable

15 pages

Guest Editors: Gudmund Grov, Andrew Ireland
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

June 15, 2015

Formal Specification, Verification, and Implementation of Fault-Tolerant Systems

Vincent Rahli and David Guaspari and Mark Bickford and Robert L. Constable

Cornell University

Abstract:

Distributed programs are known to be extremely difficult to implement, test, verify, and maintain. This is due in part to the large number of possible unforeseen interactions among components, and to the difficulty of precisely specifying what the programs should accomplish in a formal language that is intuitively clear to the programmers. We discuss here a methodology that has proven itself in building a state of the art implementation of Multi-Paxos and other distributed protocols used in a deployed database system. This article focuses on the basic ideas of formal EventML programming illustrated by implementing a fault-tolerant consensus protocol and showing how we prove its safety properties with the Nuprl proof assistant.

Keywords: functional programming formal methods formal verification theorem proving distributed systems fault tolerance event logic event-based programming

1 Introduction

Protocol Specification, Verification, and Synthesis. There is good evidence that appropriate formal methods can substantially improve the reliability of distributed protocols and that such methods are especially valuable for this kind of programming because of its intrinsic complexity. We have invested in this line of work for several years, using *constructive logic* because it supports *provably correct* code synthesis from proofs and because aspects of distributed computing are essentially constructive: agents make decisions according to some local information, and a protocol specifies how that information is acquired. "Provably correct" here means that machine checked proofs guarantee that programs satisfy desired correctness properties.

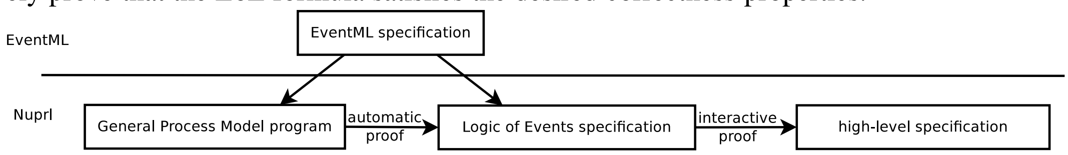
One reason that distributed systems are especially difficult to code correctly and maintain is that there are many intricate failure scenarios to consider. Failure scenarios can be hard to generate and testing them all is not usually possible. Model checkers are often used to verify that distributed systems are correct [29, 1, 19]. However, only *models* of the actual code are verified correct, and such tools may not be able to exhaustively search the space of failure scenarios. Proof assistants, however, allow one to provide definitive arguments.

We use the EventML language to develop protocols. EventML works synergistically with the Nuprl proof assistant [12, 2] which is closely related to the Coq [4, 13] proof assistant. Nuprl is a programing/logical environment based on Constructive Type Theory (CTT) [12, 2], that allows one to both prove mathematical results, and program and prove properties about these programs.

EventML. EventML is a domain-specific ML-like functional programming language for distributed protocols based on asynchronous message passing. It allows programming distributed programs in an event-based style, hence the name "EventML". The language provides *combinators* to implement what can be regarded as event recognizers and event handlers. EventML is based on two formal models of distributed computing implemented in Nuprl: (1) the Logic of Events (LoE) [5, 7] to specify and reason about the information flow of distributed program runs, (2) a General Process Model (GPM) [6] to implement these information flows. The semantic meaning of an EventML program is expressed both by a LoE formula and a GPM program. Because of this dualism we also refer to EventML programs as *constructive specifications*.

Currently, EventML *docks* with Nuprl, but in principle can connect to any prover that implements LoE and GPM. Because every EventML type is a Nuprl type, docking means that any Nuprl expression whose type is an EventML type can be imported into an EventML program.

The diagram below shows the interaction between EventML and Nuprl. Once we have extracted the semantic meaning of an EventML specification in terms of a LoE formula and a GPM program, we automatically prove that the program satisfies the formula. It remains to interactively prove that the LoE formula satisfies the desired correctness properties.



Computation Model. EventML's computation model is based on GPM. A process that takes inputs of type A, and outputs elements of type B, is an element of the following co-recursive type (the definition of the Nuprl `corec` type is outside the scope of this paper): $\texttt{corec}(\lambda P.(A \to P \times \texttt{Bag}(B)) + \texttt{Unit})$. `Unit` is a singleton type and $+$ is the disjoint union type. Therefore, a process is one of two things: a function that given an input of type A, generates a (possibly empty) bag of outputs of type B, and becomes a possibly different process; or a special value, which we call `halt`, that is used to denote a terminated process. Because GPM is implemented in Nuprl, a process is a Nuprl program (i.e., an expression of Nuprl's programming language, an untyped λ-calculus) that can be executed by interpreting it according to the rules of Nuprl's computation system.

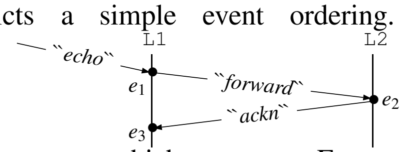
The Logic of Events. The Logic of Events (LoE) [5, 7], related to Lamport's notion of causal order [21], was developed to reason about events occurring in the execution of a distributed system. LoE has been used among other things to verify consensus protocols [28] and cyber-physical systems [3]. In the context of this paper, an event is an abstract entity corresponding to the receipt of a message[1]; the message is called the *primitive information* of the event. An event happens at a specific point in space/time. The space coordinate of an event is called its location, and the time coordinate is given by a well-founded causal ordering on events that totally orders all events at the same location. Using LoE one can describe systems in terms of the causal relations among events and (ultimately) their primitive information.

Event orderings. To reason about a protocol in LoE, we reason about its possible runs. An *event ordering* is an abstract representation of one run of a distributed system; it provides a formal definition of a *message sequence diagram* as used by systems designers. It is a structure consisting of: (1) a set of events; (2) a function `loc` that associates a *location* with each event; (3) a function `info` that associates primitive information with each event; and (4) a well-founded *causal ordering* relation, $<$, on events [21]. We express system properties as predicates on event

[1] Events are formally more general than that in the sense that they might correspond to something else than just the receipt of messages.

orderings. A system satisfies such a property if every execution satisfies the predicate.

The message sequence diagram on the right depicts a simple event ordering. Event e_1 corresponds to the receipt of a message with header ``*echo*`` at location L1. Upon receipt of that ``*echo*`` message, L1 forwards it to L2, which causes e_2. Upon receipt of that ``*forward*`` message, L2 sends an acknowledgment to L1, which causes e_3. Events e_1 and e_3 have same location, and e_1 happens causally before e_2, which happens causally before e_3. We write $e_1 < e_2$, $e_2 < e_3$, and $e_1 <_{\texttt{loc}} e_3$ ($e <_{\texttt{loc}} e'$ is defined as $e < e' \wedge \texttt{loc}(e) = \texttt{loc}(e')$).



Event observers. In LoE, we specify systems by defining and combining *event observers* [5]. An event observer is a function that assigns to any event ordering *eo* and event *e* in that event ordering *eo*, an unordered bag of outputs observed (or produced) at *e*. For example, the following event observer of type Obs(Loc), where Loc is the type of locations, recognizes every event and observes its location: $\lambda eo.\lambda e.\{\texttt{loc}(e)\}$. Event observers can therefore be regarded as combinations of *event recognizers* and *event handlers*. They effectively partition events into those they "recognize" by associating values to those events, and those they do not. For example, the *base observer* denoted vote'base recognizes the arrival of any message with header ``*vote*`` and handles that event by simply returning the content of the message[2]. We may define another observer, call it X, which recognizes that, in the context of some protocol, certain ``*vote*`` messages signify that the protocol has completed and will assign to such an event a value that means "send the 'done' message to Y." X will recognize some but not necessarily all ``*vote*`` events; and the values that it assigns to them differ from the values assigned by vote'base. We specify systems in LoE and EventML by defining and combining such event observers that appropriately classify system events.

We reason about event observers in terms of the *event observer relation*, which relates events, observers, and observations: we say that the event observer X observes v at event e (in an event ordering *eo*), and write $v \in X(e)$, if v is a member of the bag $(X\ eo\ e)$. For readability, our notation suppresses *eo*. X recognizes e when $(X\ eo\ e)$ is nonempty, in which case we also say that e is an X-event.

An EventML specification describes event observers that produce and consume messages (among other values), and especially, it describes a *main observer* that specifies the entire information flow of a system. Main observers output *directed messages* represented by pairs location/message. Given a directed message (l,m), the communication system attempts to deliver message m to location l. This directed message can be seen as the instruction "send message m to location l". This paper assumes that messages are delivered reliably but asynchronously, and may be delivered more than once.

Automation. Formally verifying distributed protocols is not trivial and can be time consuming. However, because we are using a tactic-based proof assistant in the style of LCF [18], there is much room for automation. We have built two main automation tools to assist us in proving properties of distributed systems.

From an EventML specification we automatically generate an *inductive logical form* (ILF), a first order formula that characterizes the observations made by the main observer in terms of the event observer relation. It characterizes the response to any event e in terms of observations

[2] It associates the event with a singleton bag whose value is the message body.

made at some causally prior event $e' < e$. ILFs are the heart of our verification method, providing a powerful way to prove program properties by induction on causal order.

In addition, we have automated some patterns of reasoning on state machines, because typical specifications are composed of several small state machines.

Contributions. This paper introduces basic ideas of EventML, which implements a programming paradigm in which programmers can flexibly use proof assistants to develop verified distributed programs. We show how EventML can be used to (1) define a non-trivial fault-tolerant consensus protocol in Sec. 2, (2) prove the safety properties of this protocol using automation tools in Sec. 3, and (3) generate a verified implementation in Sec. 4. Even though we illustrate our methodology on a simple consensus protocol, we have successfully used this methodology to implement industrial strength fault-tolerant distributed protocols such as Multi-Paxos. See http://nuprl.org/KB/show.php?ID=709 for more details.

2 A Specification of 2/3 Consensus

Consider the following problem: A system has been replicated for fault tolerance. It responds to *commands* identified by values in some type `Cmd`, a parameter of the specification. Commands are issued to any of the system *replicas*, which must come to consensus on the order in which those commands are to be performed, so that all replicas process commands in the same order. Replicas may fail.[3] We assume that all failures are crash failures, that is, a failed replica ceases all communication with its surroundings. The 2/3 consensus protocol [8] tolerates up to `F` failures (another parameter of the specification) by using $3*\mathtt{F}+1$ replicas. (An appealing feature of the protocol is that with a small change, and using $5*\mathtt{F}+1$ replicas, it can tolerate Byzantine failures.) Input events communicate *proposals*, which consist of slot number/command pairs. The slot numbers are modeled by integers: (n,c) proposes that command c be the n^{th} one performed. The protocol is intended to decide which proposals to accept, and to broadcast those decisions to *clients* (whose locations are also a parameter of the specification). Each copy of the replicated system contains a module that carries out the consensus negotiations. To save space, this paper describes only those modules (which we continue to call *Replicas*). An account of how these consensus decisions are used may be found in the description of the Paxos protocol [30].

2.1 A Top-Down Look at the Protocol

This section shows how EventML can organize a top-down description of the protocol, decomposing it to a level at which our remaining task is to define a few event observers that act like state machines. Sec. 2.2 describes one of those state machines, which performs the key computation used to detect consensus. Sec. 2.3 shows how EventML defines an event observer to accomplish that. Figures 1 and 2 provide the full EventML specification.

We begin by describing a structure common to many consensus protocols: Each slot n of an array of commands gets filled whenever a quorum of agents reach consensus on which command to place in n. Decisions result from holding elections, and we spawn a separate process to conduct each one. In this case, for each slot number n, we hold an election to decide which proposals of form $(n,\_)$ to accept. The tally from any particular ballot may be indecisive, so additional rounds of balloting will be spawned as needed. The crucial decisions are when to begin a new round of balloting, what constraints participants must observe in their successive votes, and how

[3] It follows from the FLP impossibility result [14] that consensus might never be reached.

Figure 1 2/3 consensus: Part 1

```
specification two_thirds
(* ============ Parameters ============ *)
parameter Cmd, cmdeq : Type * Cmd Deq (* Command type with equality decider cmdeq *)
parameter F          : Int           (* max number of failures                    *)
parameter reps       : Loc Bag       (* locations of (3 * F + 1) replicas         *)
parameter clients    : Loc Bag       (* locations of the clients to be notified   *)
(* ============ Imported Nuprl declarations ============ *)
import length poss-maj list-diff deq-member from-upto
(* ============ Type definitions ============ *)
type SlotNum  = Int                    type RoundNum    = Int
type Proposal = SlotNum * Cmd          type VotingRound = SlotNum * RoundNum
type Ballot   = VotingRound * Cmd      type Vote        = Ballot * Loc
(* ============ Interface ============ *)
input propose : Proposal      output notify     : Proposal
internal vote : Vote          internal decided : Proposal     internal retry : Ballot
(* ============ Quorum: a state machine ============ *)
(* -- filter -- *)
let new_vote (n,r) (((n',r'),cmd),sender) (cmds,locs) =
  (n,r) = (n',r') & !(deq-member (op =) sender locs);;
(* -- update -- *)
let upd_quorum (n,r) loc ((nr,c),sndr) (cmds,locs) =
  if new_vote (n,r) ((nr,c),sndr) (cmds,locs) then (c.cmds, sndr.locs) else (cmds,locs);;
(* -- output -- *)
let roundout loc (((n,r),cmd),sender) (cmds,locs) =
  if length cmds = 2 * F then let (k,cmd') = poss-maj cmdeq (cmd.cmds) cmd in
    if k = 2 * F + 1 then decided'bcast reps(n, cmd')
    else { retry'send loc ((n,r+1),cmd') }
  else {} ;;
let when_quorum (n,r) loc vt state =
  if new_vote (n,r) vt state then roundout loc vt state else {} ;;
(* -- state machine -- *)
observer QuorumState (n,r) = Memory(\loc.([],[]), upd_quorum (n,r), vote'base) ;;
observer Quorum (n,r) = (when_quorum (n,r)) o (vote'base, QuorumState (n,r)) ;;
```

to detect that consensus has been achieved (complicated by the fact that multiple rounds in the same election may be occurring simultaneously).

Interface. An input event to the protocol is the arrival of a message with header ``*propose*`` whose body is a proposal—i.e., a value of type:

```
type Proposal = Int * Cmd
```

The type of commands is a parameter of the specification:

```
parameter Cmd, cmdeq : Type * Cmd Deq
```

One subtlety: The protocol requires the ability to determine whether two values of `Cmd` are equal. So we require an additional parameter, an "equality decider"—here called `cmdeq`—able to perform that computation. The inputs to the protocol are messages with header ``*propose*`` and body of type `Proposal`

```
input propose : Proposal
```

This declaration implicitly defines the base observer `propose'base` that detects these input events and observes their data.

Outputs of the protocol are directed messages with header ``*notify*``. The data component of an output contains a `Proposal` value that has been accepted.

```
output notify : Proposal
```

This declaration does not introduce a base observer recognizing the arrival of ``*notify*`` messages, because those events occur outside our system. However, it implicitly declares the functions `notify'send` and `notify'bcast` for creating directed messages. If m is the ``*notify*``

Figure 2 2/3 consensus: part 2

```
(* ============ Round ============ *)
observer Round ((n,r),c) =
  Output(\loc.vote'bcast reps (((n,r),c),loc)) || Once(Quorum (n,r)) ;;
(* ============ NewRounds: a state machine ============ *)
(* -- inputs -- *)
observer RoundInfo = retry'base || ((\(((n,r),c),s).{((n,r),c)}) o vote'base);;
(* -- update -- *)
let upd_round n loc ((m,r'),cmd) r = if n = m & r < r' then r' else r ;;
(* -- output -- *)
let when_new_round n loc ((m,r'),cmd) r = if n = m & r < r' then {((m,r'),cmd)} else {} ;;
(* -- state machine -- *)
observer NewRoundsState n = Memory(\loc.0, upd_round n, RoundInfo) ;;
observer NewRounds n = (when_new_round n) o (RoundInfo, NewRoundsState n) ;;
(* ============ Voter ============ *)
let decision n loc (n',c) = if n = n' then notify'bcast clients (n,c) else {};;
observer Notify n = Once((decision n) o decided'base);;
observer Rounds (n,cmd) = Round ((n,0),cmd) || (NewRounds n >>= Round);;
observer Voter (n,cmd) = (Rounds (n,cmd) until (Notify n)) || (Notify n);;
(* ============ NewVoters: a state machine ============ *)
(* -- inputs -- *)
observer Proposal = propose'base || ((\(((n,r),c),s).{(n,c)}) o vote'base);;
(* -- filter -- *)
let new_proposal (n,cmd) (max,missing) = n > max or deq-member (op =) n missing;;
(* -- update -- *)
let upd_replica (n,cmd) (max,missing) =
  if new_proposal (n,cmd) (max,missing) then
     if n > max then (n, missing ++ (from-upto (max + 1) n))
     else (max, list-diff (op =) missing [n])
  else (max,missing) ;;
(* -- output -- *)
let out_proposal loc (n,cmd) state = if new_proposal (n,cmd) state then {(n,cmd)} else {};;
(* -- state machine -- *)
observer ReplicaState = Memory(\loc.(0,[]), upd_replica, Proposal) ;;
observer NewVoters = out_proposal o (Proposal, ReplicaState) ;;
(* ============ Replica & Main program ============ *)
observer Replica = NewVoters >>= Voter;;   main SC where SC = Replica @ reps
```

message with body p, then the expression $($`notify'send` l $p)$ is the directed message (l,m) instructing that m be sent to l; and the expression $($`notify'bcast` $\{l_1,l_2,\ldots\}$ $p)$ is the bag $\{(l_1,m),(l_2,m),\ldots\}$ of such instructions.

Typically, the complete interface of a system is defined in terms of its input messages, its output messages, and its internal messages, i.e., messages that can only be produced and consumed by the participants of the system. The internal messages exchanged by the participants of the protocol presented in this section are as follows: ``*vote*`` messages, by which the replicas cast their votes; ``*decided*`` messages, which inform replicas that consensus has been detected on a particular proposal; and ``*retry*`` messages, which are described below.

Replicas. To characterize top-level agents in the protocol we will define the event observer `Replica`. The main program, `SC`, executes the protocol:

```
main SC where SC = Replica @ reps
```

The bag of locations `reps`, another parameter of the specification, denotes the locations at which the replicas will execute. We may think of `SC` as the restriction of `Replica` to an observer that responds only to events at the locations in `reps`, or as the result of installing an "instance" of `Replica` at each of those locations. `SC` can be implemented by a finite number of instances, while `Replica` cannot because it responds to events at all possible locations.

For each n, the protocol conducts a separate election to vote on proposals for the n^{th} command. `Replica` spawns subprocesses that cast votes in these elections and identify the winners. The spawning (delegation) operator "_>>=_" is an EventML primitive which is used by processes to start sub-processes:[4]

```
observer Replica = NewVoters >>= Voter
```

The event observer NewVoters decides when to spawn a new voting process. Voter is a higher-order function; the values it returns are event observers that do the voting. When some NewVoters-event e occurs and $v \in$ NewVoters(e), Replica spawns a *local instance* of the observer Voter(v). By local instance we mean this: each subprocess spawned at a NewVoters-event e at location *loc* acts only at *loc* and can only react to messages arriving at *loc* after e. For any e there will be at most one v such that $v \in$ NewVoters(e). So a NewVoters-event spawns only one subprocess. (Though it is not required, we typically apply delegation only to such "singled-valued" observers.)

A note on terminology: SC requires several higher-order functions, such as Voter, that return event observers. For convenience we will use "a Voter observer" or "a Voter" as a shorthand for "an event observer returned by Voter."

State machines. Informally, we will call an observer *a state machine* if it defines a distinct state machine at each location. We will say that it reacts to an event if it recognizes the event or if the event can cause its internal state to change.

NewVoters is a state machine. It reacts to ``*proposal*`` (from outside the system) and ``*vote*`` messages (from inside), and it filters those events. At any location *loc*, NewVoters recognizes the first time that *loc* has received a proposal or vote about the n^{th} command and, when it does, outputs (a singleton bag containing) its value. If the value of such an event is (n,c), the effect of (NewVoters >>= Voter) is therefore to spawn a local instance of the event observer Voter(n,c) at location *loc*. The initialization data (n,c) instructs that Voter to vote for (n,c) on the first round.

Voter. Voter observers cast votes and tally the votes they receive to determine whether some proposal has achieved consensus. A Voter will not announce a consensus for proposal (n,c) unless it has received $2*\mathtt{F}+1$ votes for (n,c) from $2*\mathtt{F}+1$ different replicas.

We cannot guarantee that any particular poll of the Voter observers will achieve such a result. Accordingly, for each n we allow arbitrarily many do-over polls: Successive polls for slot number n are assigned consecutive integers called *round numbers*. *Voting rounds* (or just rounds for short) are pairs of the form (n,r)—(slot number/round number). *Ballots* are pairs of the form $((n,r),c)$—(voting round/command). Thus, a Voter casts votes for a particular proposal *in a particular round*. *Votes* are pairs of the form $(((n,r),c),loc)$—(ballot/location). A voter includes its location in each vote. By arranging that replicas ignore duplicate votes, we guarantee that the protocol works even if messages get duplicated.

A Replica spawns Voter subprocesses to conduct separate elections for each slot number. A Rounds observer uses essentially the same idiom to spawn Round observers that handle individual balloting rounds within a single election. A Voter is, essentially, a Rounds process that runs until its election has been decided:

```
observer Rounds (n,c) = Round ((n,0),c) || (NewRounds n >>= Round) ;;
observer Voter (n,c) = (Rounds (n,c) until (Notify n)) || (Notify n) ;;
```

[4] We use the symbol ">>=" because the event observers have the structure of a monad having this combinator as its bind operation.

where "_||_" performs parallel composition. For any event observers A and B, the observer (A until B) acts like A until a B-event occurs, at which point it terminates. We use this to terminate any voting for n once consensus has been reached on n. Rounds, Round, NewRounds, and Notify are also functions that return event observers.

A local instance of Round$((n,r),c)$ conducts the voting for round (n,r) at a particular location. By definition it will cast its vote in round (n,r) for (n,c). Therefore, the first component of Rounds(n,c) ensures that Voter(n,c) votes for proposal (n,c) in round $(n,0)$; other instances of Round, spawned by the second component of Rounds, may cast votes for other proposals in later rounds. Round (detailed in Sec. 2.2) inputs ``*vote*`` messages and outputs directed messages of various kinds: ``*vote*``; ``*decided*``; and ``*retry*``, an internal message calling for a new round when a poll does not achieve consensus.

NewRounds(n) recognizes events that call for new rounds of voting for the n^{th} command. Thus (NewRounds n >>= Round) spawns instances of Round as required.

Notify(n) handles ``*decided*`` message with data (n,c) indicating that consensus has been reached about the n^{th} command, by sending notifications to the clients of the system indicating that slot n has been filled with command c.

2.2 Detecting Consensus

Round$((n,r),c)$ has two components:

```
observer Round ((n,r),c) = Output(\loc.vote'bcast reps (((n,r),c),loc))
                         || Once(Quorum (n,r))
```

The first component multicasts a vote for (n,c) in round (n,r) to all locations in reps and then terminates. The second executes the consensus-detecting process, Quorum(n,r), and terminates once it has either announced a consensus or called for a new round. Once(A) is an observer that acts like A but terminates after the first A-event. Because there is at most one Quorum(n,r) event at any location the use of Once is logically redundant; but effects an optimization that guarantees that a process is cleaned up once it has produced an output.

Quorum(n,r) produces an output as soon as it has received votes in round (n,r) from $2*F+1$ distinct locations. If all of them are votes for the same proposal, call it (n,d), it decides that (n,d) has achieved consensus and sends appropriate ``*decided*`` messages (which will be handled by Notify observers which will send ``*notify*`` messages). If the received votes are not unanimous then it is possible that, however many more votes are tallied, no proposal will receive $2*F+1$ votes on this round. (Note that if F failures have occurred, no more votes will arrive, so Quorum cannot wait for more votes or it might become permanently stuck.) In that case it sends a ``*retry*`` message to call for round $(n,r+1)$. That ``*retry*`` message also tells the Voter that spawned the Quorum how to vote in the new round. If some command d received a majority of the $2*F+1$ votes, the Voter must vote for (n,d). (If no command gets a majority, how it votes does not matter to the logical correctness of the protocol.)

It is possible that a round will occur in which a Quorum(n,i) at one location detects a consensus and a Quorum(n,j) at another location calls for a new round of voting. As a result, multiple notifications may be sent about n, in a single round or in different rounds. Sec. 3 shows that, for any n, all notifications about the n^{th} command will agree on which command has been chosen.

2.3 Implementing `Quorum`

`Quorum(n,r)` is a Mealy state machine: in response to inputs it may change state and produce outputs. Let us factor its definition. We first define `QuorumState(n,r)`, a Moore machine whose state is the collection of votes for round `(n,r)` that the process has received thus far. `Quorum(n,r)` observes `QuorumState(n,r)` and issues directed messages as appropriate. EventML provides primitives for defining Moore machines. We use the primitive `Memory` to define `QuorumState`:

```
observer QuorumState(n,r) = Memory(\loc.([],[]),upd_quorum(n,r),vote'base)
```

A `QuorumState`(n,r) state is a pair of lists `(cmds,locs)`, where `cmds` is a list of commands and `locs` is a list of locations. The state $([c_1;c_2;\ldots],[l_1;l_2;\ldots])$ means that, in round (n,r), the state machine has thus far received a vote from l_1 for c_1, a vote from l_2 for c_2, etc. By maintaining that location list in addition to the command list, `QuorumState` can ignore duplicates; thus, as mentioned above, we need not assume that messages are delivered only once. In the definition of `QuorumState`, the arguments to `Memory` have the following meanings: (a) The expression `(\loc.([],[]))` assigns the initial state to each location, i.e., a pair of empty lists. (b) The transition function `upd_quorum(n,r)` computes the next state from the location and value of the input event and the current state. If an input vote arrives for c from l, and l is not listed in the current state, then `upd_quorum` adds c and l to its state, otherwise the current state stays unchanged. (c) `vote'base` recognizes input ``*vote*`` events and supplies their values.

`Memory` is defined so that `QuorumState` will recognize every ``*vote*`` event, update its internal state, and then return (a singleton bag containing) the value of the internal state *before* performing that update. Had it been more convenient that `QuorumState` return the value of the internal state *after* the update we would have used the primitive combinator `State` instead of `Memory`.

We define the observer `Quorum` from `QuorumState` using the primitive *composition combinator* `(f o (X1,...,Xn))`, which combines the function `f` with the event observers `X1`, ..., `Xn`. This combinator behaves as follows: for all $i \in \{1,\ldots,n\}$, if `Xi` observes `xi` at event e then the event observer `(f o (X1,...,Xn))` observes each value of the bag (`f` `loc`(e) `x1 ... xn`) at event e. `Quorum` is defined as follows:

```
observer Quorum (n,r) = (when_quorum(n,r)) o (vote'base,QuorumState(n,r))
```

This computes the response of `Quorum(n,r)` to event e by applying `when_quorum(n,r)` to `loc`(e), and to the values observed at e by `vote'base` and `QuorumState(n,r)`. Note that `Quorum(n,r)` observes only votes, but not all of them since `when_quorum(n,r)` sometimes returns an empty bag. If an input vote arrives for c from l, and l is listed in the current state, then `when_quorum` does not output anything. Otherwise, it calls `roundout`, which requires the most complex definition:

```
let roundout loc (((n,r),c),sender) (cmds,locs) =
  if length cmds = 2 * F then let (k,c') = poss-maj cmdeq (c.cmds) c in
    if k = 2 * F + 1 then decided'bcast reps (n,c')
    else { retry'send loc ((n,r+1),c') }
  else {}
```

The first argument `loc` is the location of the `Quorum` process calling `roundout` on receipt of a vote; the second argument `(((n,r),c),sender)` matches the data from the input vote; and the third argument `(cmds,locs)` matches the state when the input arrives. Therefore `c.cmds`, where the dot is the cons operation on lists, is the value of the command list that results from processing the input.

We can now understand the outer conditional: If its condition is false then, even after the input event, we have not seen $2*\texttt{F}+1$ votes; so `Quorum` returns an empty bag, and the input event is not a `Quorum(n,r)`-event. Suppose now that the condition is true and consider the inner conditional.

The `poss-maj` function, imported from EventML's library (a snapshot of Nuprl's library), implements the Boyer-Moore majority vote algorithm. The pair `(k,c')` satisfies the following property: If there is a majority entry in the list `c.cmds`, `c'` is its value and `k` is the number of times `c'` occurs in that list. The condition (`k = 2 * F + 1`) therefore tests whether the vote is unanimous. If so, the function returns instructions that the choice of `c'` be broadcast in appropriate ``*decided*`` messages; if not, it returns the instruction to send a ``*retry*`` message. Recall that the declaration of ``*retry*`` messages introduces the operation `retry'send`, for constructing directed messages. Therefore, `retry'send loc ((n,r+1),c')` is the instruction to send to `loc` a ``*retry*`` message with body `((n,r+1),c')`. So `Quorum` sends a message to its own location, which will be observed by `NewRounds`, which will spawn the round `(n,r+1)`. The message data directs the spawned instance of `Round` to vote for `c'` in the new round.

3 The Safety Properties of 2/3 Consensus

From `SC`, our EventML specification of the 2/3 consensus protocol, we generate a LoE specification and a GPM program that express `SC`'s semantic meaning in our two models of distributed computing. We verify `SC`'s correctness using the LoE specification, and we execute it using the GPM program. This section describes the formal verification, in the Nuprl proof assistant, of this protocol using the LoE specification, and Sec. 4 below describes the process of generating the GPM program and automatically verifying that it implements the LoE specification.

3.1 Agreement and Validity

The basic safety properties of any consensus protocol are *agreement* and *validity*. Both these properties have been formally proved by induction on the causal order of events in Nuprl for the 2/3 consensus protocol of Sec. 2. We state them in terms of notifications. Recall that system properties are predicates on event orderings; we must prove that the predicates are true of all possible runs of the system consistent with the `SC` specification[5]. Agreement says that notifications never contradict one another:

$$\begin{array}{l}\forall e_1,e_2{:}\,\mathbf{E}.\ \forall l_1,l_2{:}\,\mathbf{Loc}.\ \forall n{:}\,\mathbb{Z}.\ \forall c_1,c_2{:}\,\texttt{Cmd}.\\ \quad (\texttt{notify'send}\,l_1\,(n,c_1)) \in \mathsf{SC}(e_1) \wedge (\texttt{notify'send}\,l_2\,(n,c_2)) \in \mathsf{SC}(e_2) \Rightarrow c_1 = c_2\end{array}$$

Validity says that any proposal decided on must be one that was proposed:

$$\begin{array}{l}\forall e{:}\,\mathbf{E}.\ \forall l{:}\,\mathbf{Loc}.\ \forall v{:}\,\texttt{Proposal}.\\ \quad (\texttt{notify'send}\,l\,v) \in \mathsf{SC}(e) \Rightarrow \downarrow\exists e'{:}\,\mathbf{E}.\ e' < e \wedge v \in \texttt{propose'base}(e')\end{array}$$

One subtlety: The reader can think of $\downarrow\exists$ as a classical existential. The $\downarrow$ type operator, called "squash", enforces proof erasure, which is necessary here because, generally, there is no *constructive* way to pinpoint the exact ``*propose*`` event that led to a notification being sent. For example, there might have been two such proposals sent, and once we receive them, we have no way to distinguish between them if the content of these messages is identical.

[5] The formal statements of these properties contain a universally quantified variable that the notation suppresses: *eo*, denoting an event ordering.

3.2 Assumptions

For every distributed system we assume that every internal or output message received must have been sent by one of the agents of the system. Formally, we make a separate assumption for each base observer that observes an internal or an output message. For example, if $v \in \texttt{vote'base}(e)$, and e occurs at location loc, there must exist some $e' < e$ such that $(\texttt{vote'send}\ loc\ v) \in \texttt{SC}(e')$. We also assume that $\texttt{reps}$ is a bag of size $3 * \texttt{F} + 1$ without repetitions.

3.3 Automation

We have developed two automation tools that help us prove properties of distributed systems. One is a rewriting tool that consists in using the ILFs mentioned in Sec. 1 in order to prove properties by induction on causal order. The other one consists in the automation of standard patterns of reasoning on state machines.

Inductive Logical Form. ILFs are declarative logical statements that precisely answer questions such as: “What led the process at location l_1 to send a vote to the process at location l_2?”, in terms of input messages’ content and state machines’ states. ILFs are automatically generated from main observers using logical simplifications, and characterizations of the LoE combinators. For example, one of the simplest but subtle such characterizations is the one for “_||_”: $v \in X \mid\mid Y(e) \iff \downarrow(v \in X(e) \lor v \in Y(e))$. This says that v is produced by $X \mid\mid Y$ iff it is produced by either of its components. The $\downarrow$, which enforces proof erasure, is needed because just by knowing that $X \mid\mid Y$ produced v, we cannot in general know whether v was produced by X or Y. For example, if identical replicas run in parallel, and receive the same inputs, then there is no way to distinguish between their outputs if they do not label them with different tags.

Given a main observer X, we wrote a program that starts with a formula of the form $v \in X(e)$, and keeps on rewriting it using equivalences such as the one presented above, to finally generate a formula of the form $v \in X(e) \iff C$, where C is a complete declarative characterization of X’s outputs. In addition, our program also applies various logical simplifications to C. Finally, we have built a proof tactic that automatically proves such double implications.

An ILF provides a characterization of all the messages sent by a system. Because it is often useful to get these characterizations for specific kinds of messages, we also generate ILF instances for all the kinds of messages that the system outputs.

Fig. 3 shows the ILF instance for ˋˋ*vote*ˋˋ messages as generated by Nuprl. The details of this formula are not critical for understanding our methodology. However, let us explain how it characterizes the sending of ˋˋ*vote*ˋˋ messages. This formula says that a vote of the form $\texttt{<<n,r>,c>,sender>}$ ($\texttt{<\_,\_>}$ is Nuprl’s pair constructor) is sent by $\texttt{SC}$ at event $\texttt{e}$ to location $\texttt{i}$ (see box 1) iff: (box 2): $\texttt{e}$ happens at a replica location, which we call R; (box 3): $\texttt{i}$ is also a replica location; (box 4): there exists a proposal $\texttt{<n',c'>}$ that was received by R in a ˋˋ*propose*ˋˋ or ˋˋ*vote*ˋˋ message at a prior event $\texttt{e'}$; (box 5): $\texttt{<n',r'>}$ is such that $\texttt{n'}$ has never been received by R prior to $\texttt{e'}$ (there is no important distinction between $\texttt{ReplicaStateFun}$ and $\texttt{ReplicaState}$, which maintains the list of proposed slot numbers); (box 6): $\texttt{<n',r'>}$ is such that no decision has been made about $\texttt{n'}$ between $\texttt{e'}$ and $\texttt{e}$; finally, (box 7): either $\texttt{<n,c>}$ is $\texttt{<n',c'>}$ and is being voted for at the initial round $\texttt{r=0}$ in response to the ˋˋ*propose*ˋˋ or ˋˋ*vote*ˋˋ message mentioned above (see box 4) that led to a new $\texttt{Voting}$ process being spawned; (box 8): or $\texttt{<n,c>}$ comes from a ˋˋ*retry*ˋˋ or ˋˋ*vote*ˋˋ message, and $\texttt{r}$ is not the initial round, i.e., either some replica believed that consensus could not have been reached at round $\texttt{r-1}$ (in case of a ˋˋ*retry*ˋˋ), or R was still working on a

Figure 3 ILF instance for ``*vote*`` messages

```
∀[Cmd:{T:Type| valueall-type(T)}]. ∀[clients,reps:bag(Id)]. ∀[cmdeq:EqDecider(Cmd)]. ∀[F:ℤ].
∀[f:headers_type{i:l}(Cmd)]. ∀[es:EO]. ∀[e:E]. ∀[i,sender:Id]. ∀[d,n,r:ℤ]. ∀[v:Cmd].
  (<d, i, make-Msg(``vote``;<<<n, r>, c>, sender>)> ∈ main(Cmd;clients;cmdeq;F;reps;f)(e)   1
  ⟺ loc(e) ↓∈ reps  2          ∧ i ↓∈ reps  3          ∧ (d = 0)
      ∧ (↓∃n':ℤ. ∃c':Cmd. ∃e':{e':E| e' ≤loc e }.
            ((((header(e') = ``propose``) ∧ <n', c'> = body(e'))
             ∨ (has-es-info-type(es;e';f;ℤ × ℤ × Cmd × Id)
               ∧ (header(e') = ``vote``)
               ∧ (n' = (fst(fst(fst(msgval(e'))))))                  4
               ∧ (c' = (snd(fst(msgval(e')))))))
           ∧ (((fst(ReplicaStateFun(Cmd;f;es;e'))) < n')
           ∨ (n' ∈ snd(ReplicaStateFun(Cmd;f;es;e'))))           5
           ∧ (no Notify(Cmd;clients;f) n' between e' and e)         6
           ∧ ((((<<<n, r>, c>, sender> = <<<n', 0>, c'>, loc(e)>) ∧ (e = e'))    7
             ∨ (∃r':ℤ. ∃c'':Cmd. ((<<<n, r>, c>, sender> = <<<n', r'>, c''>, loc(e)>)
                  ∧ (∃e1:{e1:E| e1 ≤loc e }
                      ((((header(e1) = ``retry``) ∧ <<n', r'>, c''> = body(e1))
                       ∨ (has-es-info-type(es.e';e1;f;ℤ × ℤ × Cmd × Id)
                         ∧ (header(e1) = ``vote``)
                         ∧ (n' = (fst(fst(fst(msgval(e1))))))               8
                         ∧ (r' = (snd(fst(fst(msgval(e1))))))
                         ∧ (c'' = (snd(fst(msgval(e1)))))))
                      ∧ (NewRoundsStateFun(Cmd;f;n';es.e';e1) < r') ∧ (e = e1)))))))))
```

smaller round number when it received `r` (in case of a ``*vote*``), and is now voting at round `r`.

Using such formulas we can easily trace back the outputs of a distributed system to the states of its state machines, and to its inputs. For example, to prove `SC`'s validity property we start from the characterization of ``*notify*`` messages and trace these messages back to ``*proposals*`` using the various ILF instances.

State Machine Properties. As mentioned in Sec. 2.3, one can define Moore machines in EventML using the `Memory` and `State` keywords. Reasoning about such state machines often turns out to be a large part of the verification effort of a distributed program's correctness. Therefore, our system provides some automation to prove four kinds of local properties of `Memory` and `State` state machines, called: *invariant*, *ordering*, *progress*, and *memory*.

Informally, *a state machine invariant* is a unary property about all possible states of the state machine. *A state machine ordering* property is a binary property about all pairs of states ordered in time. *A state machine progress* property w.r.t. some predicate P is a binary property about all pairs of states ordered in time, such that P is true about at least one of the transitions made between the two states, i.e., such that some progress characterized by P has been made between the two states. *A state machine memory* property is a ternary property between an input, the current state of the machine at the time it received this input, and a later state. Memory properties are used to specify that state machines keep track of some parts of their inputs in their states.

We have proved in Nuprl, by induction on causal order, that `Memory` and `State` state machines satisfy each of these properties if, among other things, they satisfy some transition property regarding consecutive states (in the case of invariants, a base case is also necessary). Therefore, to prove that a state machine satisfies an instance of one of these four properties, we simply have to instantiate the corresponding general lemma and prove the simpler transition property.

We have developed an annotation language to state such properties in EventML, as well as general Nuprl tactics that try to prove these properties automatically (and often succeed) using logical simplifications and simple reasoners on lists, integers, etc. We illustrate invariants using `QuorumState` (the other properties are described in a longer version of this paper [27]): an invariant of a `QuorumState` state of the form `(cmds,locs)` is that `locs` has no repeats and same length as `cmds`. We call that invariant `quorum_inv`, which we state in EventML as follows:

```
import no_repeats length
invariant quorum_inv on (cmds,locs) in (QuorumState ni)
  == no_repeats ::Loc locs /\ length(cmds) = length(locs);;
```

The Nuprl tactic we have designed tries to automatically prove this statement by unfolding `QuorumState`'s definition to a `Memory` observer and by instantiating the corresponding general lemma. It (mainly) remains to prove that the base and induction properties are satisfied, which are trivial to prove in this case. Because we have already proved the general principle by induction on causal order, the tactic does not have to use induction on causal order to prove `quorum_inv`.

3.4 Proof Effort

Thanks to our automation tools and to the rich library of definitions, facts, and proof tactics about LoE and GPM that we have developed over the years, we have specified 2/3 consensus and have proved its two safety properties in Nuprl in merely two days. Proving these two properties involved: automatically generating and proving 8 state machine properties; automatically generating and proving 1 ILF and 4 instances of that ILF; and interactively proving 8 other lemmas (3 of them being trivial, and therefore candidates for future automation).

4 Correct-by-Construction Program Generation

As mentioned in Sec. 1, the semantic meaning of an EventML program is both a LoE event observer and a GPM program. We carry out our correctness proofs on the LoE description of the main event observer. To trust the program we run, we prove that the GPM program implements that LoE description, i.e., that it outputs exactly the same observations. Given an EventML specification, proving that the corresponding GPM program satisfies the corresponding LoE specification is trivial: For each EventML combinator C, there exists a corresponding LoE combinator LC and a corresponding GPM combinator PC which provably implements LC.

For example, let X_1 and X_2 be event observers of type T, implementable by pr_1 and pr_2, respectively. The LoE parallel combinator $X_1 \,||\, X_2$ is defined as $\lambda eo.\lambda e.(X_1\ eo\ e) + (X_2\ eo\ e)$, where $+$ is the append operation on bags. The GPM parallel combinator $pr_1 \,||\, pr_2$ is defined as follows (for simplicity we use the same symbol as for the LoE combinators):

$$\lambda l.\mathtt{fix}\left(\begin{array}{l}\lambda R.\lambda s.\mathtt{let}\ p_1,p_2 = s\ \mathtt{in}\\ \mathtt{if\ halted}(p_1)\wedge_b \mathtt{halted}(p_2)\ \mathtt{then\ halt}\\ \mathtt{else\ run}\left(\begin{array}{l}\lambda m.\mathtt{let}\ p'_1,out_1 = p_1(m)\ \mathtt{in}\\ \quad\ \mathtt{let}\ p'_2,out_2 = p_2(m)\ \mathtt{in}\\ \quad\ (R\ (p'_1,p'_2),out_1 + out_2)\end{array}\right)\end{array}\right)(pr_1\ l,pr_2\ l)$$

This function takes a location l and returns a process that runs p_1 and p_2 in parallel at l. This process maintains a state s composed of two processes: its two components. Its initial state is $(pr_1\ l,pr_2\ l)$. If the current state s of the process is a pair (p_1,p_2), then if both p_1 and p_2 have halted, i.e., they are the special halted process `halt`, then the process becomes `halt`. Otherwise, the process waits for an input message m, and once it has received one, then (1) for $i \in \{1,2\}$, it applies[6] p_i to m to obtain a new process p'_i and a bag of output values out_i; (2) it outputs $out_1 + out_2$ and recursively calls itself on the new state (p'_1,p'_2).

We proved that $pr_1 \,||\, pr_2$ implements $X_1 \,||\, X_2$. The same is true about the other combinators.

[6] The application of a process p to a message m is defined as follows: if $\mathtt{halted}(p)$ then return $(\mathtt{halt},\{\})$, otherwise p is of the form $\mathtt{run}(f)$, and therefore, return $(f\ m)$.

5 Related Work

Much work has been done on specifying and reasoning about distributed systems [24, 17, 10, 11, 9, 19] (to only cite a few).

IOA. IOA [16, 15, 17] is a programming/specification language for describing asynchronous distributed systems as I/O automata (labeled state transition systems) and stating their properties. IOA can interact with a large range of tools such as type checkers, simulators, model checkers, theorem provers, and there is support for synthesis of Java code. Both I/O automata and event observers can specify I/O observations of distributed systems. While IOA is state-based, LoE is event-based (states are implicitly maintained by recursive combinators). Also, our methodology allows us to both prove protocol properties and generate code within Nuprl, and does not require any translation to another language.

TLA. TLA is a temporal logic, based on first-order logic and set theory, that "provides a mathematical foundation for describing systems" [23]. TLA$^+$ [23, 11] is a language for specifying systems described in TLA. TLAPS "is a platform for the development and mechanical verification of TLA$^+$ proofs" [11]. To validate proofs, TLAPS uses a collection of theorem provers, proof assistants, SMT solvers, and decision procedures. One can use a model checker to help catch errors before attempting any proof. At its current stage, TLAPS allows one to prove safety properties (the safety property of a variant of Paxos has been verified using TLAPS) but not liveness/non-blocking properties (we have not yet proved such properties either). TLA$^+$ does not perform program synthesis.

seL4. Our approach is similar to the one taken by Klein et al. to verify the seL4 microkernel [20]. They use Haskell as their specification language, which roughly corresponds to the level of abstraction of EventML in our framework. Then, they translate this code to an Isabelle/HOL version. They prove that this executable specification refines an abstract one, which corresponds to LoE's level. Finally, they generate by hand a C implementation of the specification, which they translate into Isabelle/HOL, in which they defined a model of C, and manually prove that this implementation refines their executable specification. This corresponds to GPM's level. Among other things, our paper shows that a similar methodology can be used to design and implement correct fault-tolerant distributed systems.

Verdi. More recently, Wilcox et al. developed Verdi [31], which is a framework, similar to ours, to develop and reason about distributed systems using Coq. As in our framework they do not have gaps between the code they verify and the code they run: they run OCaml code that they extract from Coq. Verdi provides a compositional way of specifying distributed systems. This is done by applying *verified system transformers*. For example, Paxos transforms a distributed system into a crash-tolerant distributed system (they verified Raft [25] instead of Paxos). One difference between our respective methods is that they verify systems by reasoning about the evolution of the *state of the world*, while our approach relies on the notion of causal order.

6 Conclusion, Current and Future Work

Our methodology scales to more complicated and subtle distributed protocols. For example, we have specified the Multi-Paxos protocol [22, 30] in EventML and proved its safety properties to be correct in Nuprl. We have also built an ordered broadcast service that can switch between various consensus protocols [28]. To get efficient code, we have built in Nuprl a formal tool tuned to automatically optimize GPM programs and prove that the optimized code and the non-optimized program are bisimilar [26]. We are also experimenting with compilers to Lisp and Scala. We are now building support in EventML and Nuprl to: (1) abstract away from implementation details such as specific data structures, (2) automatically prove simple properties such as validity properties, (3) replay large proofs in order to support modifications to specifications. This paper only discussed safety properties. We have started proving progress and non-blocking properties of the 2/3 consensus protocol. However, it turns out that these proofs are tedious. Next, we want to build automation tools to assist us in proving such properties.

Bibliography

[1] Francesco Alberti, Silvio Ghilardi, Elena Pagani, Silvio Ranise, and Gian Paolo Rossi. Automated support for the design and validation of fault tolerant parameterized systems: a case study. *ECEASST*, 35, 2010.

[2] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469, 2006. http://www.nuprl.org/.

[3] Abhishek Anand and Ross Knepper. ROSCoq: Robots powered by constructive reals. ITP 2015 (http://www.cs.cornell.edu/~aa755/ROSCoq/), 2015.

[4] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. http://coq.inria.fr/.

[5] Mark Bickford. Component specification using event classes. In *Component-Based Software Engineering, 12th Int'l Symp.*, volume 5582 of *LNCS*, pages 140–155. Springer, 2009.

[6] Mark Bickford, Robert Constable, and David Guaspari. Generating event logics with higher-order processes as realizers. Technical report, Cornell University, 2010.

[7] Mark Bickford, Robert L. Constable, and Vincent Rahli. Logic of events, a framework to reason about distributed systems. In *Languages for Distributed Algorithms Workshop*, 2012.

[8] B. Charron-Bost and A. Schiper. The Heard-Of model: computing in distributed systems with benign failures. *Distributed Computing*, 22(1):49–71, 2009.

[9] Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. Formal verification of consensus algorithms tolerating malicious faults. In *SSS 2011*, volume 6976 of *LNCS*, pages 120–134. Springer, 2011.

[10] Bernadette Charron-Bost and Stephan Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.

[11] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying Safety Properties With the TLA+ Proof System. In Jürgen Giesl and Reiner Haehnle, editors, *IJCAR 2010*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 142–148. Springer, 2010.

[12] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P.Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[13] The Coq Proof Assistant. http://coq.inria.fr/.

[14] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[15] S. Garland, N. Lynch, J. Tauber, and M. Vaziri. IOA user guide and reference manual. Technical Report MIT/LCS/TR-961, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[16] Stephen J. Garland and Nancy Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of componentbased systems*, pages 285–312. Cambridge University Press, New York, NY, USA, 2000.

[17] Chryssis Georgiou, Nancy Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. Automated implementation of complex distributed algorithms specified in the IOA language. *Int. J. Softw. Tools Technol. Transf.*, 11:153–171, February 2009.

[18] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[19] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD 2013*, pages 201–209. IEEE, 2013.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP 2009*, pages 207–220. ACM, 2009.

[21] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[22] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[23] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2004.

[24] Patrick Lincoln and John M. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *FTCS 1993*, pages 402–411. IEEE Computer Society, 1993.

[25] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC '14*, pages 305–319. USENIX Association, 2014.

[26] Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in Nuprl using computational equivalence and partial types. In *ITP'13*, volume 7998 of *LNCS*, pages 261–278. Springer, 2013.

[27] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems. http://nuprl.org/KB/show.php?ID=709, 2013.

[28] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. Developing correctly replicated databases using formal tools. In *DSN 2014*, pages 395–406. IEEE, 2014.

[29] Tatsuhiro Tsuchiya and André Schiper. Using bounded model checking to verify consensus algorithms. In *DISC 2008*, volume 5218 of *LNCS*, pages 466–480. Springer, 2008.

[30] Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):5:1–5:36, 2015.

[31] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for formally verifying distributed system implementations. PLDI 2015 http://verdi.uwplse.org/, 2015.